MODULAR LOGIC GRAMMARS

Michael C. McCord
IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

ABSTRACT

This report describes a logic grammar formalism, **Modular Logic Grammars**, exhibiting a high degree of modularity between syntax and semantics. There is a syntax rule compiler (compiling into Prolog) which takes care of the building of analysis structures and the interface to a clearly separated semantic interpretation component dealing with scoping and the construction of logical forms. The whole system can work in either a one-pass mode or a two-pass mode. In the one-pass mode, logical forms are built directly during parsing through interleaved calls to semantics, added automatically by the rule compiler. In the two-pass mode, syntactic analysis trees are built automatically in the first pass, and then given to the (one-pass) semantic component. The grammar formalism includes two devices which cause the automatically built syntactic structures to differ from derivation trees in two ways: (1) There is a **shift** operator, for dealing with left-embedding constructions such as English possessive noun phrases while using right-recursive rules (which are appropriate for Prolog parsing). (2) There is a distinction in the syntactic formalism between **strong** non-terminals and **weak** non-terminals, which is important for distinguishing major levels of grammar.

1. INTRODUCTION

The term **logic grammar** will be used here, in the context of natural language processing, to mean a logic programming system (implemented normally in Prolog), which associates semantic representations (normally in some version of predicate logic) with natural language text. Logic grammars may have varying degrees on modularity in their treatments of syntax and semantics. There may or may not be an isolatable syntactic component.

In writing metamorphosis grammars (Colmerauer, 1978), or definite clause grammars, DCG's, (a special case of metamorphosis grammars, Pereira and Warren, 1980), it is possible to build logical forms directly in the syntax rules by letting non-terminals have arguments that represent partial logical forms being manipulated. Some of the earliest logic grammars (e.g., Dahl, 1977) used this approach. There is certainly an appeal in being direct, but there are some disadvantages in this lack of modularity. One disadvantage is that it seems difficult to get an adequate treatment of the scoping of quantifiers (and more generally **focalizers**, McCord, 1981) when the building of logical forms is too closely bonded to syntax. Another disadvantage is just a general result of lack of modularity: it can be harder to develop and understand syntax rules when too much is going on in them.

The logic grammars described in McCord (1982, 1981) were three-pass systems, where one of the main points of the modularity was a good treatment of scoping. The first pass was the syntactic component, written as a definite clause grammar, where syntactic structures were explicitly built up in the arguments of the non-terminals. Word sense selection and slot-filling were done in this first pass, so that the output analysis trees were actually partially semantic. The second pass was a preliminary stage of semantic interpretation in which the syntactic analysis tree was **reshaped** to reflect proper scoping of modifiers. The third pass took the reshaped tree and produced logical forms in a straightforward way by carrying out **modification** of nodes by their daughters using a modular system of rules that manipulate **semantic items** -- consisting of logical forms together with terms that determine how they can combine.

The CHAT-80 system (Pereira and Warren, 1982, Pereira, 1983) is a three-pass system. The first pass is a purely syntactic component using an **extraposition grammar** (Pereira, 1981) and producing syntactic analyses in **rightmost normal form**. The second pass handles word sense selection and slot-filling, and the third pass handles some scoping phenomena and the final semantic interpretation. One gets a great deal of modularity between syntax and semantics in that the first component has no elements of semantic interpretation at all.

In McCord (1984) a one-pass semantic interpretation component, SEM, for the EPISTLE system (Miller, Heidorn and Jensen, 1981) was described. SEM has been interfaced both to the EPISTLE NLP grammar (Heidorn, 1972, Jensen and Heidorn, 1983), as well as to a logic grammar, SYNT, written as a DCG by the author. These grammars are purely syntactic and use the EPISTLE notion (op. cit.) of **approximate parse**, which is similar to Pereira's notion of **rightmost normal form**, but was developed independently. Thus SYNT/SEM is a two-pass system with a clear modularity between syntax and semantics.

In DCG's and extraposition grammars, the building of analysis structures (either logical forms or syntactic trees) must be specified explicitly in the syntax rules. A certain amount of modularity is then lost, because the grammar writer must be aware of manipulating these structures, and the possibility of using the grammar in different ways is reduced. In Dahl and McCord (1983), a logic grammar formalism was described, **modifier structure grammars** (MSG's), in which structure-building (of annotated derivation trees) is implicit in the formalism. MSG's look formally like extraposition grammars, with the additional ingredient that semantic items (of the type used in McCord (1981)) can be indicated on the left-hand sides of rules, and contribute automatically to the construction of a syntactico-semantic tree much like that in McCord (1981). These MSG's were used interpretively in parsing, and then (essentially) the two-pass semantic interpretation system of McCord (1981) was used to get logical forms. So, totally there were three passes in this system.

In this report, I wish to describe a logic grammar system, **modular logic grammars** (MLG's), with the following features:

- There is a syntax rule compiler which takes care of the building of analysis structures and the interface to semantic interpretation.
- There is a clearly separated semantic interpretation component dealing with scoping and the construction of logical forms.
- The whole system (syntax and semantics) can work optionally in either a one-pass mode or a two-pass mode.
- In the one-pass mode, no syntactic structures are built, but logical forms are built directly during parsing through interleaved calls to the semantic interpretation component, added automatically by the rule compiler.
- In the two-pass mode, the calls to the semantic interpretation component are not interleaved, but are made in a second pass, operating on syntactic analysis trees produced (automatically) in the first pass.
- The syntactic formalism includes a device, called the **shift** operator, for dealing with left-embedding constructions such as English possessive noun phrases ("my wife's brother's friend's car") and Japanese relative clauses. The shift operator instructs the rule compiler to build the structures appropriate for left-embedding. These structures are not derivation trees, because the syntax rules are right-recursive, because of the top-down parsing associated with Prolog.
- There is a distinction in the syntactic formalism between **strong** non-terminals and **weak** non-terminals, which is important for distinguishing major levels of grammar and which simplifies the working of semantic interpretation. This distinction also makes the (automatically produced) syntactic analysis trees much more readable and natural linguistically. In the absence of shift constructions, these trees are like derivation trees, but only with nodes corresponding to strong non-terminals.
- In an experimental MLG, the semantic component handles all the scoping phenomena handled by that in McCord (1981) and more than the semantic component in McCord (1984). The logical form language is improved over that in the previous systems.

The MLG formalism allows for a great deal of modularity in natural language grammars, because the syntax rules can be written with very little awareness of semantics or the building of analysis structures, and the very same syntactic component can be used in either the one-pass or the two-pass mode described above.

Three other logic grammar systems designed with modularity in mind are Hirschman and Puder (1982), Abramson (1984) and Porto and Filgueiras (1984). These will be compared with MLG's in Section 6.

2. THE MLG SYNTACTIC FORMALISM

The syntactic component for an MLG consists of a declaration of the **strong non-terminals**, followed by a sequence of MLG syntax rules. The declaration of strong non-terminals is of the form

strongnonterminals(NT1.NT2. ... .NTn.nil).

where the NTi are the desired strong non-terminals (only their principal functors are indicated). Non-terminals that are not declared strong are called **weak**. The significance of the strong/weak distinction will be explained below.

MLG syntax rules are of the form

A ==> B

where A is a **non-terminal** and B is a **rule body**. A **rule body** is any combination of **surface terminals**, **logical terminals**, **goals**, **shifted non-terminals**, **non-terminals**, the symbol 'nil', and the cut symbol '/', using the sequencing operator ':' and the 'or' symbol '|'. (We represent left-to-right sequencing with a colon instead of a comma, as is often done in logic grammars.) These rule body elements are Prolog terms (normally with arguments), and they are distinguished formally as follows.

- A **surface terminal** is of the form +A, where A is any Prolog term. Surface terminals correspond to ordinary terminals in DCG's (they match elements of the surface word string), and the notation is often [A] in DCG's.
- A **logical terminal** is of the form Op-LF, where Op is a **modification operator** and LF is a logical form. Logical terminals are special cases of **semantic items**, the significance of which will be explained below. Formally, the rule compiler

recognizes them as being terms of the form A-B. There can be any number of them in a rule body.

- A **goal** is of the form $A, where A is a term representing a Prolog goal. (This is the usual provision for Prolog procedure calls, which are often indicated by enclosure in braces in DCG's.)

- A **shifted non-terminal** is either of the form %A, or of the form F%A, where A is a weak non-terminal and F is any term. (In practice, F will be a list of features.) As indicated in the Introduction, the shift operator '%' is used to handle left-embedding constructions in a right-recursive rule system.

- Any rule body element not of the above four forms and not 'nil' or the cut symbol is taken to be a **non-terminal**.

A **terminal** is either a surface terminal or a logical terminal. Surface terminals are building blocks for the word string being analyzed, and logical terminals are building blocks for the analysis structures.

A syntax rule is called **strong** or **weak**, according as the non-terminal on its left-hand side is strong or weak.

It can be seen that on a purely formal level, the only differences between MLG syntax rules and DCG's are (1) the appearance of logical terminals in rule bodies of MLG's, (2) the use of the shift operator, and (3) the distinction between strong and weak non-terminals. However, for a given linguistic coverage, the syntactic component of an MLG will normally be more compact than the corresponding DCG because structure-building must be explicit in DCG's. In this report, the arrow '-->' (as opposed to '==>') will be used for for DCG rules, and the same notation for sequencing, terminals, etc., will be used for DCG's as for MLG's.

What is the significance of the strong/weak distinction for non-terminals and rules? Roughly, a strong rule should be thought of as introducing a new **level** of grammar, whereas a weak rule defines analysis within a level. Major categories like **sentence** and **noun phrase** are expanded by strong rules, but auxiliary rules like the recursive rules that find the postmodifiers of a verb are weak rules. An analogy with ATN's (Woods, 1970) is that strong non-terminals are like the start categories of subnetworks (with structure-building POP arcs for termination), whereas weak non-terminals are like internal nodes.

In the one-pass mode, the MLG rule compiler makes the following distinction for strong and weak rules. In the Horn clause translation of a strong rule, a call to the semantic interpretation component is compiled in at the end of the clause. The non-terminals appearing in rules (both strong and weak) are given extra arguments which manipulate semantic structures used in the call to semantic interpretation. No such call to semantics is compiled in for weak rules. Weak rules only gather information to be used in the call to semantics made by the next higher strong rule. (Also, a shift generates a call to semantics.)

In the two-pass mode, where syntactic analysis trees are built during the first pass, the rule compiler builds in the construction of a tree node corresponding to every strong rule. The node is labeled essentially by the non-terminal appearing on the left-hand side of the strong rule. (A shift also generates the construction of a tree node.) Details of rule compilation will be given in the next section.

As indicated above, logical terminals, and more generally semantic items, are of the form

```
Operator-LogicalForm.
```

The Operator is a term which determines how the semantic item can combine with other semantic items during semantic interpretation. (In this combination, new semantic items are formed which are no longer logical terminals.) Logical terminals are most typically associated with lexical items, although they are also used to produce certain non-lexical ingredients in logical form analysis. An example for the lexical item "each" might be

```
Q/P - each(P,Q).
```

Here the operator Q/P is such that when the "each" item modifies, say, an item having logical form man(X), P gets unified with man(X), and the resulting semantic item is

```
@Q - each(man(X),Q)
```

where @Q is an operator which causes Q to get unified with the logical form of a further modificand. Details of the use of semantic items will be given in Section 4.

Now let us look at the syntactic component of a sample MLG which covers the same ground as a well-known DCG. The following DCG is taken essentially from Pereira and Warren (1980). It is the sort of DCG that builds logical forms directly by manipulating partial logical forms in arguments of the grammar symbols.

```
sent(P) --> np(X,P1,P): vp(X,P1).
np(X,P1,P) --> det(P2,P1,P): noun(X,P3):
                relclause(X,P3,P2).
np(X,P,P) --> name(X).
vp(X,P) --> transverb(X,Y,P1): np(Y,P1,P).
vp(X,P) --> intransverb(X,P).
relclause(X,P1,P1&P2) --> +that: vp(X,P2).
relclause(*,P,P) --> nil.
det(P1,P2,P) --> +D: $dt(D,P1,P2,P).
noun(X,P) --> +N: $n(N,X,P).
name(X) --> +X: $nm(X).
transverb(X,Y,P) --> +V: $tv(V,X,Y,P).
intransverb(X,P) --> +V: $iv(V,X,P).

/*  Lexicon  */

n(man,X,man(X)).  n(woman,X,woman(X)).
nm(john).  nm(mary).
```

```
dt(every,P1,P2,all(P1,P2)).
dt(a,P1,P2,ex(P1,P2)).
tv(loves,X,Y,love(X,Y)).
iv(lives,X,live(X)).
```

The syntactic component of an analogous MLG is as follows. The lexicon is exactly the same as that of the preceding DCG. For reference below, this grammar will be called MLGRAM.

```
strongnonterminals(sent.np.relclause.det.nil).

sent ==> np(X): vp(X).
np(X) ==> det: noun(X): relclause(X).
np(X) ==> name(X).
vp(X) ==> transverb(X,Y): np(Y).
vp(X) ==> intransverb(X).
relclause(X) ==> +that: vp(X).
relclause(*) ==> nil.
det ==> +D: $dt(D,P1,P2,P): P2/P1-P.
noun(X) ==> +N: $n(N,X,P): 1-P.
name(X) ==> +X: $nm(X).
transverb(X,Y) ==> +V: $tv(V,X,Y,P): 1-P.
intransverb(X) ==> +V: $iv(V,X,P): 1-P
```

This small grammar illustrates all the ingredients of MLG syntax rules except the shift operator. The shift will be illustrated below. Note that 'sent' and 'np' are strong categories but 'vp' is weak. A result is that there will be no call to semantics at the end of the 'vp' rule. Instead, the semantic structures associated with the verb and object are passed up to the 'sent' level, so that the subject and object are "thrown into the same pot" for semantic combination. (However, their surface order is not forgotten.)

There are only two types of modification operators appearing in the semantic items of this MLG: '1' and P2/P1. The operator '1' means 'left-conjoin'. Its effect is to left-conjoin its associated logical form to the logical form of the modificand (although its use in this small grammar is almost trivial). The operator P2/P1 is associated with determiners, and its effect has been illustrated above.

The semantic component will be given below in Section 4. A sample semantic analysis for the sentence "Every man that lives loves a woman" is

```
all(man(X1)&live(X1),ex(woman(X2),love(X1,X2))).
```

This is the same as for the above DCG. We will also show a sample parse in the next section.

A fragment of an MLG illustrating the use of the shift in the treatment of possessive noun phrases is as follows:

```
np  ==> det: np1.
np1 ==> premods: noun: np2.
np2 ==> postmods.
np2 ==> poss: %np1.
```

The idea of this fragment can be described in a rough procedural way, as follows. In parsing an np, one reads an ordinary determiner (det), then goes to np1. In np1, one reads several premodifiers (premods), say adjectives, then a head noun, then goes to np2. In np2, one may either finish by reading postmodifiers (postmods), OR one may read an apostrophe-s (poss) and then SHIFT back to np1. Illustration for the noun phrase, "the old man's dusty hat":

```
    the     old     man      's
np det np1 premods noun np2 poss %np1

dusty   hat      (nil)
premods noun np2 postmods
```

When the shift is encountered, the syntactic structures (in the two-pass mode) are manipulated (in the compiled rules) so that the initial np ("the old man") becomes a left-embedded sub-structure of the larger np (whose head is "hat"). But if no apostrophe-s is encountered, then the structure for "the old man" remains on the top level.

3. COMPILATION OF MLG SYNTAX RULES

In describing rule compilation, we will first look at the two-pass mode, where syntactic structures are built in the first pass, because the relationship of the analysis structures to the syntax rules is more direct in this case.

The syntactic structures manipulated by the compiled rules are represented as **syntactic items**, which are terms of the form

```
syn(Features,Daughters)
```

where Features is a **feature list** (to be defined), and Daughters is a list consisting of syntactic items and terminals. Both types of terminal (surface and logical) are included in Daughters, but the displaying procedures for syntactic structures can optionally filter out one or the other of the two types. A **feature list** is of the form nt:Arg1, where nt is the principal functor of a strong non-terminal and Arg1 is its first argument. (If nt has no arguments, we take Arg1=nil.) It is convenient, in large grammars, to use this first argument Arg1 to hold a list (based on the operator ':') of grammatical features of the phrase analyzed by the non-terminal (like **number** and **person** for noun phrases).

In compiling DCG rules into Prolog clauses, each non-terminal gets two extra arguments treated as a difference list representing the word string analyzed by the non-terminal. In compiling MLG rules, exactly the same thing is done to handle word strings. For handling syntactic structures, the MLG rule compiler adds additional arguments which manipulate 'syn' structures. The number of additional arguments and the way they are used depend on whether the non-terminal is strong or weak. If the original non-terminal is strong and has the form

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

then in the compiled version we will have

```
nt(X1, ..., Xn, Syn, Str1,Str2).
```

Here there is a single syntactic structure argument, Syn, representing the syntactic structure of the phrase associated by nt with the word string given by the difference list (Str1, Str2).

On the other hand, when the non-terminal nt is weak, four syntactic structure arguments are added, producing a compiled predication of the form

```
nt(X1, ..., Xn, Syn0,Syn, Mods1,Mods2, Str1,Str2).
```

Here the pair (Mods1, Mods2) holds a difference list for the sequence of structures analyzed by the weak non-terminal nt. These structures could be 'syn' structures or terminals, and they will be daughters (modifiers) for a 'syn' structure associated with the closest higher call to a strong non-terminal -- let us call this higher 'syn' structure the **matrix** 'syn' structure. The other pair (Syn0, Syn) represents the changing view of what the matrix 'syn' structure actually should be, a view that may change because a shift is encountered while satisfying nt. Syn0 represents the version before satisfying nt, and Syn represents the version after satisfying nt. If no shift is encountered while satisfying nt, then Syn will just equal Syn0. But if a shift is encountered, the old version Syn0 will become a daughter node in the new version Syn.

In compiling a rule with several non-terminals in the rule body, linked by the sequencing operator ':', the argument pairs (Syn0, Syn) and (Mods1, Mods2) for weak non-terminals are linked, respectively, across adjacent non-terminals in a manner similar to the linking of the difference lists for word-string arguments. Calls to strong non-terminals associate 'syn' structure elements with the modifier lists, just as surface terminals are associated with elements of the word-string lists.

Let us look now at the compilation of a set of rules. We will take the noun phrase grammar fragment illustrating the shift and shown above in Section 2, and repeated for convenience here, together with declarations of strong non-terminals.

```
strongnonterminals(np.det.noun.poss.nil).

np  ==> det: np1.
np1 ==> premods: noun: np2.
np2 ==> postmods.
np2 ==> poss: %np1.
```

The compiled rules are as follows:

```
np(Syn, Str1,Str3) <-
  det(Mod, Str1,Str2) &
  np1(syn(np:nil,Mod:Mods),Syn,
      Mods,nil, Str2,Str3).

np1(Syn1,Syn3, Mods1,Mods3, Str1,Str4) <-
  premods(Syn1,Syn2, Mods1,Mod:Mods2,
          Str1,Str2) &
  noun(Mod, Str2,Str3) &
  np2(Syn2,Syn3, Mods2,Mods3, Str3,Str4).

np2(Syn1,Syn2, Mods1,Mods2, Str1,Str2) <-
  postmods(Syn1,Syn2, Mods1,Mods2, Str1,Str2).

np2(syn(Feas,Mods0),Syn, Mod:Mods1,Mods1,
      Str1,Str3) <-
  poss(Mod, Str1,Str2) &
  np1(syn(Feas,syn(Feas,Mods0):Mods2),Syn,
      Mods2,nil, Str2,Str3).
```

In the first compiled rule, the structure Syn to be associated with the call to 'np' appears again in the second matrix structure argument of 'np1'. The first matrix structure argument of 'np1' is

```
syn(np:nil,Mod:Mods).
```

and this will turn out to be the value of Syn if no shifts are encountered. Here Mod is the 'syn' structure associated with the determiner 'det', and Mods is the list of modifiers determined further by 'np1'. The feature list np:nil is constructed from the leading non-terminal 'np' of this strong rule. (It would have been np:Arg1 if np had a (first) argument Arg1.)

In the second and third compiled rules, the matrix structure pairs (first two arguments) and the modifier difference list pairs are linked in a straightforward way to reflect sequencing.

The fourth rule shows the effect of the shift. Here syn(Feas,Mods0), the previous "conjecture" for the matrix structure, is now made simply the first modifier in the larger structure

```
syn(Feas,syn(Feas,Mods0):Mods2)
```

which becomes the new "conjecture" by being placed in the first argument of the further call to 'np1'. If the shift operator had been used in its binary form F0%np1, then the new conjecture would be

```
syn(NT:F,syn(NT:F0,Mods0):Mods2)
```

where the old conjecture was syn(NT:F,Mods0). In larger grammars, this allows one to have a completely correct feature list NT:F0 for the left-embedded modifier.

To illustrate the compilation of terminal symbols, let us look at the rule

```
det ==> +D: $dt(D,P1,P2,P): P2/P1-P.
```

from the grammar MLGRAM in Section 2. The compiled rule is

```
det(syn(det:nil,+D:P2/P1-P:nil), D.Str,Str) <-
  dt(D,P1,P2,P).
```

Note that both the surface terminal +D and the logical terminal P2/P1-P are entered as modifiers of the 'det' node. The semantic interpretation component looks only at the logical terminals, but in certain applications it is useful to be able to see the surface terminals in the syntactic structures. As mentioned above, the display procedures for syntactic structures can optionally show only one type of terminal.

The display of the syntactic structure of the sentence "Every man loves a woman" produced by MLGRAM is as follows.

```
sentence:nil
    np:X1
        det:nil
            X2/X3-all(X3,X2)
        1-man(X1)
    1-love(X1,X4)
    np:X4
        det:nil
            X5/X6-ex(X6,X5)
        1-woman(X4)
```

Note that no 'vp' node is shown in the parse tree; 'vp' is a weak non-terminal. The logical form produced for this tree by the semantic component given in the next section is

```
all(man(X1), ex(woman(X2),love(X1,X2))).
```

Now let us look at the compilation of syntax rules for the one-pass mode. In this mode, syntactic structures are not built, but semantic structures are built up directly. The rule compiler adds extra arguments to non-terminals for manipulation of semantic structures, and adds calls to the top-level semantic interpretation procedure, 'semant'.

The procedure 'semant' builds complex semantic structures out of simpler ones, where the original building blocks are the logical terminals appearing in the MLG syntax rules. In this process of construction, it would be possible to work with semantic items (and in fact a subsystem of the rules do work directly with semantic items), but it appears to be more efficient to work with slightly more elaborate structures which we call **augmented semantic items**. These are terms of the form

```
sem(Feas,Op,LF),
```

where Op and LF are such that Op-LF is an ordinary semantic item, and Feas is either a feature list or the list terminal:nil. The latter form is used for the initial augmented semantic items associated with logical terminals.

As in the two-pass mode, the number of analysis structure arguments added to a non-terminal by the compiler depends on whether the non-terminal is strong or weak. If the original non-terminal is strong and has the form

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

then in the compiled version we will have

```
nt(X1, ..., Xn, Sems1,Sems2, Str1,Str2).
```

Here (Sems1, Sems2) is a difference list of augmented semantic items representing the list of semantic structures for the phrase associated by nt with the word string given by the difference list (Str1, Str2). In the syntactic (two-pass) mode, only one argument (for a 'syn') is needed here, but now we need a list of structures because of a raising phenomenon necessary for proper scoping, which we will discuss in Sections 4 and 5.

When the non-terminal nt is weak, five extra arguments are added, producing a compiled predication of the form

```
nt(X1, ..., Xn, Feas, Sems0,Sems, Sems1,Sems2,
                                   Str1,Str2).
```

Here Feas is the feature list for the matrix strong non-terminal. The pair (Sems0, Sems) represents the changing "conjecture" for the complete list of daughter (augmented) semantic items for the matrix node, and is analogous to first extra argument pair in the two-pass mode. The pair (Sems1, Sems2) holds a difference list for the sequence of semantic items analyzed by the weak non-terminal nt. Sems1 will be a final sublist of Sems0, and Sems2 will of course be a final sublist of Sems1.

For each strong rule, a call to 'semant' is added at the end of the compiled form of the rule. The form of the call is

```
semant(Feas, Sems, Sems1,Sems2).
```

Here Feas is the feature list for the non-terminal on the left-hand side of the rule. Sems is the final version of the list of daughter semantic items (after all adjustments for shifts) and (Sems1, Sems2) is the difference list of semantic items resulting from the semantic interpretation for this level. (Think of Feas and Sems as input to 'semant', and (Sems1, Sems2) as output.) (Sems1, Sems2) will be the structure arguments for the non-terminal on the left-hand side of the strong rule. A call to 'semant' is also generated when a shift is encountered, as we will see below. The actual working of 'semant' is the topic of the next section.

For the shift grammar fragment shown above, the compiled rules are as follows.

```
np(Sems,Sems0, Str1,Str3) <-
  det(Sems1,Sems2, Str1,Str2) &
  np1(np:nil, Sems1,Sems3, Sems2,nil, Str2,Str3) &
  semant(np:nil, Sems3, Sems,Sems0).

np1(Feas, Sems1,Sems3, Sems4,Sems7, Str1,Str4) <-
  premods(Feas, Sems1,Sems2, Sems4,Sems5,
                             Str1,Str2) &
  noun(Sems5,Sems6, Str2,Str3) &
  np2(Feas, Sems2,Sems3, Sems6,Sems7, Str3,Str4).

np2(Feas, Sems1,Sems2, Sems3,Sems4, Str1,Str2) <-
  postmods(Feas, Sems1,Sems2, Sems3,Sems4,
                              Str1,Str2).

np2(Feas, Sems1,Sems4, Sems5,Sems6, Str1,Str3) <-
  poss(Sems5,Sems6, Str1,Str2) &
  semant(Feas, Sems1, Sems2,Sems3) &
  np1(Feas, Sems2,Sems4, Sems3,nil, Str2,Str3).
```

In the first compiled rule (a strong rule), the pair (Sems, Sems0) is a difference list of the semantic items analyzing the noun phrase. (Typically there

will just be one element in this list, but there can be more when modifiers of the noun phrases contain quantifiers that cause the modifiers to get promoted semantically to be sisters of the noun phrase.) This difference list is the output of the call to 'semant' compiled in at the end of the first rule. The input to this call is the list Sems3 (along with the feature list np:nil). We arrive at Sems3 as follows. The list Sems1 is started by the call to 'det'; its first element is the determiner (if there is one), and the list is continued in the list Sems2 of modifiers determined further by the call to 'np1'. In this call to 'np1', the initial list Sems1 is given in the second argument of 'np1' as the "initial version" for the final list of modifiers of the noun phrase. Sems3, being in the next argument of 'np1', is the "final version" of the np modifier list, and this is the list given as input to 'semant'. If the processing of 'np1' encounters no shifts, then Sems3 will just equal Sems1.

In the second compiled rule (for 'np1'), the "versions" of the total list of modifiers are linked in a chain

```
(Sems1, Sems2, Sems3)
```

in the second and third arguments of the weak non-terminals. The actual modifiers produced by this rule are linked in a chain

```
(Sems4, Sems5, Sems6, Sems7)
```

in the fourth and fifth arguments of the weak non-terminals and the first and second arguments of the strong non-terminals. A similar situation holds for the first of the 'np2' rules.

In the second 'np2' rule, a shift is encountered, so a call to 'semant' is generated. This is necessary because of the shift of levels; the modifiers produced so far represent all the modifiers in an np, and these must be combined by 'semant' to get the analysis of this np. As input to this call to 'semant', we take the list Sems1, which is the current version of the modifiers of the matrix np. The output is the difference list (Sems2, Sems3). Sems2 is given to the succeeding call to 'np1' as the new current version of the matrix modifier list. The tail Sems3 of the difference list output by 'semant' is given to 'np1' in its fourth argument to receive further modifiers. Sems4 is the final version of the matrix modifier list, determined by 'np1', and this information is also put in the third argument of 'np2'. The difference list (Sems5, Sems6) contains the single element produced by 'poss', and this list tails off the list Sems1.

When a semantic item Op-LF occurs in a rule body, the rule compiler inserts the augmented semantic item sem(terminal:nil,Op,LF). As an example, the weak rule

```
transverb(X,Y) ==> +V: $tv(V,X,Y,P): 1-P.
```

compiles into the clause

```
transverb(X,Y, Feas, Sems1,Sems1,
          sem(terminal:nil,1,P):Sems2,Sems2,
          V.Str,Str) <-
  tv(V,X,Y,P).
```

The strong rule

```
det  ==> +D: $dt(D,P1,P2,P): P2/P1-P.
```

compiles into the clause

```
det(Sems1,Sems2, D.Sems4,Sems4)<-
  dt(D,P1,P2,P) &
  semant(det:nil,
         sem(terminal:nil,P2/P1,P):nil,
         Sems1,Sems2).
```

4. SEMANTIC INTERPRETATION FOR MLG'S

The semantic interpretation schemes for both the one-pass mode and the two-pass mode share a large core of common procedures; they differ only at the top level. In both schemes, augmented semantic items are combined with one another, forming more and more complex items, until a single item is constructed which represents the structure of the whole sentence. In this final structure, only the logical form component is of interest; the other two components are discarded. We will describe the top levels for both modes, then describe the common core.

The top level for the one-pass mode is simpler, because semantic interpretation works in tandem with the parser, and does not itself have to go through the parse tree. The procedure 'semant', which has interleaved calls in the compiled syntax rules, essentially is the top-level procedure, but there is some minor cleaning up that has to be done. If the top-level non-terminal is 'sentence' (with no arguments), then the top-level analysis procedure for the one-pass mode can be

```
analyze(Sent) <-
  sentence(Sems,nil,Sent,nil) &
  semant(top:nil,Sems,sem(*,*,LF):nil,nil) &
  outlogform(LF).
```

Normally, the first argument, Sems, of 'sentence', will be a list containing a single augmented semantic item, and its logical form component will be the desired logical form. However, for some grammars, the additional call to 'semant' is needed to complete the modification process. The procedure 'outlogform' simplifies the logical form and outputs it.

The definition of 'semant' itself is given in a single clause:

```
semant(Feas,Sems,Sems2,Sems3) <-
  reorder(Sems,Sems1) &
  modlist(Sems1,sem(Feas,id,t),
          Sem,Sems2,Sem:Sems3).
```

Here, the procedure 'reorder' takes the list Sems of augmented semantic items to be combined and re-

orders it (permutes it), to obtain proper (or most likely) scoping. This procedure belongs to the common core of the two methods of semantic interpretation, and will be discussed further below. The procedure 'modlist' does the following. A call

```
modlist(Sems,Sem0,Sem,Sems1,Sems2)
```

takes a list Sems of (augmented) semantic items and combines them with (lets them modify) the item Sem0, producing an item Sem (as the combination), along with a difference list (Sems1, Sems2) of items which are promoted to be sisters of Sem. The leftmost member of Sems acts as the outermost modifier. Thus, in the definition of 'semant', the result list Sems1 of reordering acts on the trivial item sem(Feas,id,t) to form a difference list (Sems2, Sem:Sems3) where the result Sem is right-appended to its sisters. 'modlist' also belongs to the common core, and will be defined below.

The top level for the two-pass system can be defined as follows.

```
analyze2(Sent) <-
  sentence(Syn,Sent,nil) &
  synsem(Syn,Sems,nil) &
  semant(top:nil,Sems,sem(*,*,LF):nil,nil) &
  outlogform(LF).
```

The only difference between this and 'analyze' above is that the call to 'sentence' produces a syntactic item Syn, and this is given to the procedure 'synsem'. The latter is the main recursive procedure of the two-pass system. A call

```
synsem(Syn,Sems1,Sems2)
```

takes a syntactic item Syn and produces a difference list (Sems1, Sems2) of augmented semantic items representing the semantic structure of Syn. (Typically, this list will just have one element, but it can have more if modifiers get promoted to sisters of the node.)

The definition of 'synsem' is as follows.

```
synsem(syn(Feas,Mods),Sems2,Sems3) <-
  synsemlist(Mods,Sems) &
  reorder(Sems,Sems1) &
  modlist(Sems1,sem(Feas,id,t),
          Sem,Sems2,Sem:Sems3).
```

Note that this differs from the definition of 'semant' only in that 'synsem' must first recursively process the daughters Mods of its input syntactic item before calling 'reorder' and 'modlist'. The procedure 'synsemlist' that processes the daughters is defined as follows.

```
synsemlist(syn(Feas,Mods0):Mods,Sems1) <- /&
  synsem(syn(Feas,Mods0),Sems1,Sems2) &
  synsemlist(Mods,Sems2).
synsemlist((Op-LF):Mods,
       sem(terminal:nil,Op,LF):Sems) <- /&
  synsemlist(Mods,Sems).
synsemlist(Mod:Mods,Sems) <-
  synsemlist(Mods,Sems).
synsemlist(nil,nil).
```

The first clause calls 'synsem' recursively when the daughter is another 'syn' structure. The second clause replaces a logical terminal by an augmented semantic item whose feature list is terminal:nil. The next clause ignores any other type of daughter (this would normally be a surface terminal).

Now we can proceed to the common core of the two semantic interpretation systems. The procedure 'modlist' is defined recursively in a straightforward way:

```
modlist(Sem:Sems, Sem0, Sem2, Sems1,Sems3) <-
  modlist(Sems, Sem0, Sem1, Sems2,Sems3) &
  modify(Sem, Sem1, Sem2, Sems1,Sems2).
modlist(nil, Sem, Sem, Sems,Sems).
```

Here 'modify' takes a single item Sem and lets it operate on Sem1, giving Sem2 and a difference list (Sems1, Sems2) of sister items. Its definition is

```
modify(Sem, Sem1, Sem1, Sem2:Sems,Sems) <-
  raise(Sem,Sem1,Sem2) &/.
modify(sem(*,Op,LF),
       sem(Feas,Op1,LF1),
       sem(Feas,Op2,LF2), Sems,Sems) <-
  mod(Op-LF, Op1-LF1, Op2-LF2).
```

Here 'raise' is responsible for raising the item Sem1 so that it becomes a sister of the item Sem1; Sem2 is a new version of Sem1 after the raising, although in most cases, Sem2 equals Sem1. Raising occurs for a noun phrase like "a chicken in every pot", where the quantifier "every" has higher scope than the quantifier "a". The semantic item for "every pot" gets promoted to a left sister of that for "a chicken". 'raise' is defined basically by a system of unit clauses which look at specific types of phrases. For the small grammar MLGRAM of Section 2, no raising is necessary, and the definition of 'raise' can just be omitted.

The procedures 'raise' and 'reorder' are two key ingredients of **reshaping** (the movement of semantic items to handle scoping problems), which was discussed extensively in McCord (1982, 1981). In those two systems, reshaping was a separate pass of semantic interpretation, but here, as in McCord (1984), reshaping is interleaved with the rest of semantic interpretation. In spite of the new top-level organization for semantic interpretation of MLG's, the low-level procedures for raising and reordering are basically the same as in the previous systems, and we refer to the previous reports for further discussion.

The procedure 'mod', used in the second clause for 'modify', is the heart of semantic interpretation.

```
mod(Sem, Sem1, Sem2)
```

means that the (non-augmented) semantic item Sem modifies (combines with) the item Sem1 to give the item Sem2. 'mod' is defined by a system consisting basically of unit clauses which key off the modification operators appearing in the semantic items.

In the experimental MLG described in the next section, there are 22 such clauses. For the grammar MLGRAM of Section 2, the following set of clauses suffices.

```
mod(id-*, Sem, Sem) <- /.
mod(Sem, id-*, Sem) <- /.
mod(1-P, Op-Q, Op-R) <- and(P,Q,R).
mod(P/Q-R, Op-Q, @P-R).
mod(@P-Q, Op-P, Op-Q).
```

The first two clauses say that the operator 'id' acts like an identity. The second clause defines '1' as a left-conjoining operator (its corresponding logical form gets left-conjoined to that of the modificand). The call and(P,Q,R) makes R=P&Q, except that it treats 't' ('true') as an identity. The next clause for 'mod' allows a quantifier semantic item like P/Q-each(Q,P) to operate on an item like 1-man(X) to give the item @P-each(man(X),P). The final clause then allows this item to operate on 1-live(X) to give 1-each(man(X),live(X)).

The low-level procedure 'mod' is the same (in purpose) as the procedure 'trans' in McCord (1981), and has close similarities to 'trans' in McCord (1982) and 'mod' in McCord (1984), so we refer to this previous work for more illustrations of this approach to modification.

For MLGRAM, the only ingredient of semantic interpretation remaining to be defined is 'reorder'. We can define it in a way that is somewhat more general than is necessary for this small grammar, but which employs a technique useful for larger grammars. Each augmented semantic item is assigned a precedence number, and the reordering (sorting) is done so that when item B has higher precedence number than item A, then B is ordered to the left of A; otherwise items are kept in their original order. The following clauses then define 'reorder' in a way suitable for MLGRAM.

```
reorder(A:L,M) <-
   reorder(L,L1) & insert(A,L1,M).
reorder(nil,nil).

insert(A,B:L,B:L1) <-
  prec(A,PA) & prec(B,PB) & gt(PB,PA) &/&
  insert(A,L,L1).
insert(A,L,A:L).

prec(sem(terminal:*,*,*),2) <- /.
prec(sem(relclause:*,*,*),1) <- /.
prec(*,3).
```

Thus terminals are ordered to the end, except not after relative clauses. In particular, the subject and object of a sentence are ordered before the verb (a terminal in the sentence), and this allows the straightforward process of modification in 'mod' to scope the quantifiers of the subject and object over the material of the verb. One can alter the definition of 'prec' to get finer distinctions in scoping, and for this we refer to McCord (1982, 1981).

For a grammar as small as MLGRAM, which has no treatment of scoping phenomena, the total complexity of the MLG, including the semantic interpretation component we have given in this section, is certainly greater than that of the comparable DCG in Section 2. However, for larger grammars, the modularity is definitely worthwhile -- conceptually, and probably in the total size of the system.

5. AN EXPERIMENTAL MLG

This section describes briefly an experimental MLG, called MODL, which covers the same linguistic ground as the grammar (called MOD) in McCord (1981). The syntactic component of MOD, a DCG, is essentially the same as that in McCord (1982). One feature of these syntactic components is a systematic use of slot-filling to treat complements of verbs and nouns. This method increases modularity between syntax and lexicon, and is described in detail in McCord (1982).

One purpose of MOD, which is carried over to MODL, is a good treatment of scoping of modifiers and a good specification of logical form. The logical form language used by MODL as the target of semantic interpretation has been improved somewhat over that used for MOD. We describe here some of the characteristics of the new logical form language, called LFL, and give sample LFL analyses obtained by MODL, but we defer a more detailed description of LFL to a later report.

The main predicates of LFL are **word-senses** for words in the natural language being analyzed, for example, believe1(X,Y) in the sense "X believes that Y holds". Quantifiers, like 'each', are special cases of word-senses. There are also a small number of **non-lexical predicates** in LFL, some of which are associated with inflections of words, like 'past' for past tense, or syntactic constructions, like 'yesno' for yes-no questions, or have significance at discourse level, dealing for instance with topic/comment. The arguments for predicates of LFL can be conscants, variables, or other logical forms (expressions of LFL).

Expressions of LFL are either predications (in the sense just indicated) or combinations of LFL expressions using the conjunction '&' and the **indexing operator** ':'. Specifically, if P is a logical form and E is a variable, then P:E (read "P indexed by E") is also a logical form. When an indexed logical form P:E appears as part of a larger logical form Q, and the index variable E is used elsewhere in Q, then E can be thought of roughly as standing for P together with its "context". Contexts include references to time and place which are normally left implicit in natural language. When P specifies an event, as in see(john,mary), writing P:E and subsequently using E will guarantee that E refers to the same event. In the logical form language used in McCord (1981), event variables (as arguments of verb and noun senses) were used for indexing. But the indexing operator is more powerful because it can index complex logical forms. For some applications, it is sufficient to ignore contexts, and in such cases we just think of P:E as verifying P and binding E to an instantiation

of P. In fact, for PROLOG execution of logical forms without contexts, ':' can be defined by the single clause: P:P <- P.

A specific purpose of the MOD system in McCord (1981) was to point out the importance of a class of predicates called **focalizers**, and to offer a method for dealing with them in semantic interpretation. Focalizers include many determiners, adverbs, and adjectives (or their word-senses), as well as certain non-lexical predicates like 'yesno'. Focalizers take two logical form arguments called the **base** and the **focus**:

```
focalizer(Base,Focus).
```

The Focus is often associated with sentence stress, hence the name. The pair (Base, Focus) is called the **scope** of the focalizer.

The adverbs 'only' and 'even' are focalizers which most clearly exhibit the connection with stress. The predication only(P,Q) reads "the only case where P holds is when Q also holds". We get different analyses depending on focus.

```
John only buys books at Smith's.
only(at(smith,buy(john,X1)), book(X1)).

John only buys books at Smith's.
only(book(X1)&at(X2,buy(john,X1)), X2=smith).
```

Quantificational adverbs like 'always' and 'seldom', studied by David Lewis (1975), are also focalizers. Lewis made the point that these quantifiers are properly considered **unselective**, in the sense that they quantify over all the free variables in (what we call) their bases. For example, in

```
John always buys books at Smith's.
always(book(X1)&at(X2,buy(john,X1)), X2=smith).
```

the quantification is over both X1 and X2. (A paraphrase is "Always, if X1 is a book and John buys X1 at X2, then X2 is Smith's".)

Quantificational determiners are also focalizers (and are unselective quantifiers); they correspond closely in meaning to the quantificational adverbs ('all' - 'always', 'many' - 'often', 'few' - 'seldom', etc.). We have the paraphrases:

```
Leopards often attack monkeys in trees.
often(leopard(X1)&tree(X2)&in(X2,attack(X1,X3)),
      monkey(X3)).

Many leopard attacks in trees are (attacks)
   on monkeys.
many(leopard(X1)&tree(X2)&in(X2,attack(X1,X3)),
     monkey(X3)).
```

Adverbs and adjectives involving comparison or degree along some scale of evaluation (a wide class) are also focalizers. The base specifies the base of comparison, and the focus singles out what is being compared to the base. This shows up most clearly in the superlative forms. Consider the adverb "fastest":

```
John ran fastest yesterday.
fastest(run(john):E, yesterday(E)).

John ran fastest yesterday.
fastest(yesterday(run(X)), X=john).
```

In the first sentence, with focus on "yesterday", the meaning is that, among all the events of John's running (this is the base), John's running yesterday was fastest. The logical form illustrates the indexing operator. In the second sentence, with focus on "John", the meaning is that among all the events of running yesterday (there is an implicit location for these events), John's running was fastest.

As an example of a non-lexical focalizer, we have yesno(P,Q), which presupposes that a case of P holds, and asks whether P & Q holds. (The pair (P, Q) is like Topic/Comment for yes-no questions.) Example:

```
Did John see Mary yesterday?
yesno(yesterday(see(john,X)), X=mary).
```

It is possible to give Prolog definitions for most of the focalizers discussed above which are suitable for extensional evaluation and which amount to model-theoretic definitions of them. This will be discussed in a later report on LFL.

A point of the grammar MODL is to be able to produce LFL analyses of sentences using the modular semantic interpretation system outlined in the preceding section, and to arrive at the right (or most likely) scopes for focalizers and other modifiers. The decision on scoping can depend on heuristics involving precedences, on very reliable cues from the syntactic position, and even on the specification of foci by explicit underlining in the input string (which is most relevant for adverbial focalizers). Although written text does not often use such explicit specification of adverbial foci, it is important that the system can get the right logical form after having some specification of the adverbial focus, because this specification might be obtained from prosody in spoken language, or might come from the use of discourse information. It also is an indication of the modularity of the system that it can use the same syntactic rules and parse path no matter where the adverbial focus happens to lie.

Most of the specific linguistic information for semantic interpretation is encoded in the procedures 'mod', 'reorder', and 'raise', which manipulate semantic items. In MODL there are 22 clauses for the procedure 'mod', most of which are unit clauses. These involve ten different modification operators, four of which were illustrated in the preceding section. The definition of 'mod' in MODL is taken fairly directly from the corresponding procedure 'trans' in MOD (McCord, 1981), although there are some changes involved in handling the new version of the logical form language (LFL),

especially the indexing operator. The definitions of 'reorder' and 'raise' are essentially the same as for procedures in MOD.

An illustration of analysis in the two-pass mode in MODL is now given. For the sentence "Leopards only attack monkeys in trees", the syntactic analysis tree is as follows.

```
sent
    nounph
        1-leopard(X)
    avp
        (P<Q)-only(P,Q)
    1-attack(X,Y)
    nounph
        1-monkey(Y)
    prepph
        @@R-in(Z,R)
        nounph
            1-tree(Z)
```

Here we display complete logical terminals in the leaf nodes of the tree. An indication of the meanings of the operators (P<Q) and @@R will be given below.

In the semantic interpretation of the prepositional phrase, the 'tree' item gets promoted (by 'raise') to be a left-sister of the the 'in' item, and the list of daughter items (augmented semantic items) of the 'sent' node is the following.

```
nounph 1 leopard(X)
avp P<Q only(P,Q)
terminal 1 attack(X,Y)
nounph 1 monkey(Y)
nounph 1 tree(Z)
prepph  @@R in(Z,R).
```

Here we display each augmented semantic item sem(nt:Feas,Op,LF) simply in the form nt Op LF. The material in the first field of the 'monkey' item actually shows that it is stressed. The reshaping procedure 'reorder' rearranges these items into the order:

```
nounph 1 leopard(X)
nounph 1 tree(Z)
prepph  @@R in(Z,R)
terminal 1 attack(X,Y)
avp P<Q only(P,Q)
nounph 1 monkey(Y)
```

Next, these items successively modify (according to the rules for 'mod') the matrix item, sent id t, with the rightmost daughter acting as innermost modifier. The rules for 'mod' involving the operator (P<Q) associated with only(P,Q) are designed so that the logical form material to the right of 'only' goes into the focus Q of 'only' and the material to the left goes into the base P. The material to the right is just monkey(Y). The items on the left ('leopard', 'tree', 'in', 'attack') are allowed to combine (through 'mod') in an independent way before being put into the base of 'only'. The operator @@R associated with in(Z,R) causes R to be bound to the logical form of the modificand -- attack(X,Y). The combination of items on the left of 'only' is

```
leopard(X)&tree(Z)&in(Z,attack(X,Y))
```

This goes into the base, so the whole logical form is

```
only(leopard(X)&tree(Z)&in(Z,attack(X,Y)),
     monkey(Y)).
```

For detailed traces of logical form construction by this method, see McCord (1981).

An illustration of the treatment of left-embedding in MODL in a two-pass analysis of the sentence "John sees each boy's brother's teacher" is as follows.

```
sent
    nounph
        1-(X=john)
    1-see(X,W)
    nounph
        nounph
            nounph
                determiner
                    Q/P-each(P,Q)
                1-boy(Y)
                1-poss
            1-brother(Z,Y)
            1-poss
        1-teacher(W,Z)

Logical form...

each(boy(Y),the(brother(Z,Y),
                the(teacher(W,Z),see(john,W)))).
```

The MODL noun phrase rules include the shift (in a way that is an elaboration of the shift grammar fragment in Section 2), as well as rules for slot-filling for nouns like 'brother' and 'teacher' which have more than one argument in logical form. Exactly the same logical form is obtained by MODL for the sentence "John sees the teacher of the brother of each boy". Both of these analyses involve raising. In the first, the 'poss' node resulting from the apostrophe-s is raised to become a definite article. In the second, the prepositional phrases (their semantic structures) are promoted to be sisters of the "teacher" node, and the order of the quantifiers is (correctly) reversed.

The syntactic component of MODL was adapted as closely as possible from that of MOD (a DCG) in order to get an idea of the efficiency of MLG's. The fact that the MLG rule compiler produces more structure-building arguments than are in the DCG would tend to lengthen analysis times, but it is hard to predict the effect of the different organization of the semantic interpreter (from a three-pass system to a one-pass and a two-pass version of MODL). The following five sentences were used for timing tests.

Who did John say that the man introduced Mary to?
Each book Mary said was given to Bill

was written by a woman.
Leopards only attack monkeys in trees.
John saw each boy's brother's teacher.
Does anyone wanting to see the teacher know whether there are any books left in this room?

Using Waterloo Prolog (an interpreter) on an IBM 3081, the following average times to get the logical forms for the five sentences were obtained (not including time for I/O and initial word separation):

MODL, one-pass mode - 40 milliseconds.
MODL, two-pass mode - 42 milliseconds.
MOD - 35 milliseconds.

So there was a loss of speed, but not a significant one. MODL has also been implemented in PSC Prolog (on a 3081). Here the average one-pass analysis time for the five sentences was improved to 30 milliseconds per sentence.

On the other hand, the MLG grammar (in source form) is more compact and easier to understand. The syntactic components for MOD and MODL were compared numerically by a Prolog program that totals up the **sizes** of all the grammar rules, where the size of a compound term is defined to be 1 plus the sum of the sizes of its arguments, and the size of any other term is 1. The total for MODL was 1433, and for MOD was 1807, for a ratio of 79%.

So far, nothing has been said in this report about semantic constraints in MODL. Currently, MODL exercises constraints by unification of semantic types. Prolog terms representing type requirements on slot-fillers must be unified with types of actual fillers. The types used in MODL are **type trees**. A **type tree** is either a variable (unspecified type) or a term whose principal functor is an atomic type (like 'human'), and whose arguments are **subordinate** type trees. A type tree T1 is **subordinate** to a type tree T2 if either T1 is a variable or the principal functor of T1 is a subtype (**ako**) of the principal functor of T2. Type trees are a generalization of the **type lists** used by Dahl (1981), which are lists of the form T1:T2:T3:..., where T1 is a supertype of T2, T2 is a supertype of T3, ..., and the tail of the list may be a variable. The point of the generalization is to allow cross-classification. Multiple daughters of a type node cross-classify it. The lexicon in MODL includes a preprocessor for lexical entries which allows the original lexical entries to specify type constraints in a compact, non-redundant way. There is a Prolog representation for type-hierarchies, and the lexical preprocessor manufactures full type trees from a specification of their leaf nodes.

In the one-pass mode for analysis with MLG's, logical forms get built up during parsing, so logical forms are available for examination by semantic checking procedures of the sort outlined in McCord (1984). If such methods are arguably best, then there may be more argument for a one-pass system (with interleaving of semantics). The general question of the number of passes in a natural language understander is an interesting one. The MLG formalism makes this easier to investigate, because the same syntactic component can be used with one-pass or two-pass interpretation.

In MODL, there is a small dictionary stored directly in Prolog, but MODL is also interfaced to a large dictionary/morphology system (Byrd, 1983, 1984) which produces syntactic and morphological information for words based on over 70,000 lemmata. There are plans to include enough semantic information in this dictionary to provide semantic constraints for a large MLG.

Alexa McCray is working on the syntactic component for an MLG with very wide coverage. I wish to thank her for useful conversations about the nature of the system.

6. COMPARISON WITH OTHER SYSTEMS

The **Restriction Grammars** (RG's) of Hirschman and Puder (1982) are logic grammars that were designed with modularity in mind. Restriction Grammars derive from the Linguistic String Project (Sager, 1981). An RG consists of context-free phrase structure rules to which **restrictions** are appended. The rule compiler (written in Prolog and compiling into Prolog), sees to it that derivation trees are constructed automatically during parsing. The restrictions appended to the rules are basically Prolog procedures which can walk around, during the parse, in the partially constructed parse tree, and can look at the words remaining in the input stream. Thus there is a modularity between the phrase-structure parts of the syntax rules and the restrictions. The paper contains an interesting discussion of Prolog representations of parse trees that make it easy to walk around in them.

A disadvantage of RG's is that the automatically constructed analysis tree is just a derivation tree. With MLG's, the shift operator and the declaration of strong non-terminals produce analysis structures which are more appropriate semantically and are easier to read for large grammars. In addition, MLG analysis trees contain logical terminals as building blocks for a modular semantic interpretation system. The method of walking about in the partially constructed parse tree is powerful and is worth exploring further; but the more common way of exercising constraints in logic grammars by parameter passing and unification seems to be adequate linguistically and notationally more compact, as well as more efficient for the compiled Prolog program.

Another type of logic grammar developed with modularity in mind is the Definite Clause Translation Grammars (DCTG's) of Abramson (1984). These were inspired partially by RG's (Hirschman and Puder, 1982), by MSG's (Dahl and McCord, 1983), and by Attribute Grammars (Knuth, 1968). A DCTG rule is like a DCG rule with an appended list of clauses which compute the semantics of the node resulting from use of the rule. The non-terminals on the right-hand side of the syntactic portion of the rule can be indexed by variables, and these index variables can be used in the semantic portion to link to the syntactic portion. For example, the DCG rule

```
sent(P) --> np(X,P1,P): vp(X,P1).
```

from the DCG in Section 2 has the DCTG equivalent:

```
sent ::= np@N: vp@V <:>
   logic(P) ::- N@logic(X,P1,P) & V@logic(X,P1).
```

(Our notation is slightly different from Abramson's and is designed to fit the Prolog syntax of this report.) Here the indexing operator is '@'. The syntactic portion is separated from the semantic portion by the operator '<:>'. The non-terminals in DCTG's can have arguments, as in DCG's, which could be used to exercise constraints (restrictions), but it is possible to do everything by referring to the indexing variables. The DCTG rule compiler sees to the automatic construction of a derivation tree, where each node is labeled not only by the expanded non-terminal but also by the list of clauses in the semantic portion of the expanding rule. These clauses can then be used in computing the semantics of the node. When an indexed non-terminal NT@X appears on the right-hand side of a rule, the indexing variable X gets instantiated to the tree node corresponding to the expansion of NT.

There is a definite separation of DCTG rules into a syntactic portion and a semantic portion, with a resulting increase of modularity. Procedures involving different sorts of constraints can be separated from one another, because of the device of referring to the indexing variables. However, it seems that once the reader (or writer) knows that certain variables in the DCG rule deal with the construction of logical forms, the original DCG rule is just as easy (if not easier) to read. The DCTG rule is definitely longer than the DCG rule. The corresponding MLG rule:

```
sent ==> np(X): vp(X).
```

is shorter, and does not need to mention logical forms at all. Of course, there are relevant portions of the semantic component that are applied in connection with this rule, but many parts of the semantic component are relevant to several syntax rules, thus reducing the total size of the system.

A claimed advantage for DCTG's is that the semantics for each rule is listed locally with each rule. There is certainly an appeal in that, because with MLG's (as well as the methods in McCord (1982, 1981)), the semantics seems to float off more on its own. Semantic items do have a life of their own, and they can move about in the tree (implicitly, in some versions of the semantic interpreter) because of raising and reordering. This is not as neat theoretically, but it seems more appropriate for capturing actual natural language.

Another disadvantage of DCTG's (as with RG's) is that the analysis trees that are constructed automatically are derivation trees.

The last system to be discussed here, that in Porto and Filgueiras (1984), does not involve a new grammar formalism, but a methodology for writing DCG's. The authors define a notion of **intermediate semantic representation** (ISR) including **entities** and **predications**, where the predications can be viewed as logical forms. In writing DCG rules, one systematically includes at the end of the rule a call to a semantic procedure (specific to the given rule) which combines ISR's obtained in arguments of the non-terminals on the right-hand side of the rule. Two DCG rules in this style (given by the authors) are as follows:

```
sent(S) --> np(N): vp(V): $ssv(N,V,S).
vp(S) --> verb(V,trans): np(N): $svo(V,N,S).
```

Here 'ssv' and 'svo' are semantic procedures that are specific to the 'sent' rule and the 'vp' rule, respectively. The rules that define 'ssv' and 'svo' can include some general rules, but also a mass of very specific rules tied to specific words. Two specific rules given by the authors for analyzing "All Viennese composers wrote a waltz" are as follows.

```
svo(wrote,M:X,wrote(X)) <- is_a(M,music).
ssv(P:X,wrote(Y),author_of(Y,X)) <-
   is_a(P,person).
```

Note that the verb 'wrote' changes from the surface form 'wrote', to the intermediate form wrote(X), then to the form author_of(Y,X). In most logic grammar systems (including MOD and MODL), some form of argument filling is done for predicates; information is added by binding argument variables, rather than changing the whole form of the predication. The authors claim that it is less efficient to do argument filling, because one can make an early choice of a word sense which may lead to failure and backtracking. An intermediate form like wrote(X) above may only make a partial decision about the sense.

The value of the "changing" method over the "adding" method would appear to hinge a lot on the question of parse-time efficiency, because the "changing" method seems more complicated conceptually. It seems simpler to have the notion that there are word-senses which are predicates with a certain number of arguments, and to deal only with these, rather than inventing intermediate forms that help in discrimination during the parse. So it is partly an empirical question which would be decided after logic grammars dealing semantically with massive dictionaries are developed.

There is modularity in rules written in the style of Porto and Filgueiras, because all the semantic structure-building is concentrated in the semantic procedures added (by the grammar writer) at the ends of the rules. In MLG's, in the one-pass mode, the same semantic procedure call, to 'semant', is added at the ends of strong rules, automatically by the compiler. The diversity comes in the ancillary procedures for 'semant', especially 'mod'. In fact, 'mod' (or 'trans' in McCord, 1981) has something in common with the Porto-Filgueiras procedures in that it takes two intermediate representations (semantic items) in its first two arguments and produces a new intermediate representation in its third argument. However, the

changes that 'mod' makes all involve the modification-operator components of semantic items, rather than the logical-form components. It might be interesting and worthwhile to look at a combination of the two approaches.

Both a strength and a weakness of the Porto-Filgueiras semantic procedures (compared with 'mod') is that there are many of them, associated with specific syntactic rules. The strength is that a specific procedure knows that it is looking at the "results" of a specific rule. But a weakness is that generalizations are missed. For example, modification by a quantified noun phrase (after slot-filling or the equivalent) is often the same, no matter where it comes from. The method in MLG's allows semantic items to move about and then act by one 'mod' rule. The reshaping procedures are free to look at specific syntactic information, even specific words when necessary, because they work with augmented semantic items. Of course, another disadvantage of the diversity of the Porto-Filgueiras procedures is that they must be explicitly added by the writer of syntax rules, so that there is not as much modularity as in MLG's.

REFERENCES

Abramson, H. (1984) "Definite clause translation grammars," Proc. 1984 International Symposium on Logic Programming, pp. 233-240, Atlantic City.

Byrd, R. J. (1983) "Word formation in natural language processing systems," Proc. 8th International Joint Conference on Artificial Intelligence, pp. 704-706, Karlsruhe.

Byrd, R. J. (1984) "The Ultimate Dictionary Users' Guide," IBM Research Internal Report.

Colmerauer, A. (1978) "Metamorphosis grammars," in L. Bolc (Ed.), Natural Language Communication with Computers, Springer-Verlag.

Dahl, V. (1977) "Un systeme deductif d'interrogation de banques de donnees en espagnol," Groupe d'Intelligence Artificielle, Univ. d'Aix-Marseille.

Dahl, V. (1981) "Translating Spanish into logic through logic," American Journal of Computational Linguistics, vol. 7, pp. 149-164.

Dahl, V, and McCord, M. C. (1983) "Treating coordination in logic grammars," American Journal of Computational Linguistics, vol. 9, pp. 69-91.

Heidorn, G. E. (1972) Natural Language Inputs to a Simulation Programming System, Naval Postgraduate School Technical Report No. NPS-55HD72101A.

Hirschman, L. and Puder, K. (1982) "Restriction grammar in Prolog," Proc. First International Logic Programming Conference, pp. 85-90, Marseille.

Jensen, K. and Heidorn, G. E. (1983) "The fitted parse: 100% parsing capability in a syntactic grammar of English," IBM Research Report RC 9729.

Knuth, D. E. (1968) "Semantics of context-free languages," Mathematical Systems Theory, vol. 2, pp. 127-145.

Lewis, D. (1975) "Adverbs of quantification," In E.L. Keenan (Ed.), Formal Semantics of Natural Language, pp. 3-15, Cambridge University Press.

McCord, M. C. (1982) "Using slots and modifiers in logic grammars for natural language," Artificial Intelligence, vol 18, pp. 327-367. (Appeared first as 1980 Technical Report, University of Kentucky.)

McCord, M. C. (1981) "Focalizers, the scoping problem, and semantic interpretation rules in logic grammars," Technical Report, University of Kentucky. To appear in Logic Programming and its Applications, D. Warren and M: van Caneghem, Eds.

McCord, M. C. (1984) "Semantic interpretation for the EPISTLE system," Proc. Second International Logic Programming Conference, pp. 65-76, Uppsala.

Miller, L. A., Heidorn, G. E., and Jensen, K. (1981) "Text-critiquing with the EPISTLE system: an author's aid to better syntax," AFIPS Conference Proceedings, vol. 50, pp. 649-655.

Pereira, F. (1981) "Extraposition grammars," American Journal of Computational Linguistics, vol. 7, pp. 243-256.

Pereira, F. (1983) "Logic for natural language analysis," SRI International, Technical Note 275.

Pereira, F. and Warren, D. (1980) "Definite clause grammars for language analysis - a survey of the formalism and a comparison with transition networks," Artificial Intelligence, vol. 13, pp. 231-278.

Pereira, F. and Warren, D. (1982) "An efficient easily adaptable system for interpreting natural language queries," American Journal of Computational Linguistics, vol. 8, pp. 110-119.

Porto, A. and Filgueiras, M. (1984) "Natural language semantics: A logic programming approach," Proc. 1984 International Symposium on Logic Programming, pp. 228-232, Atlantic City.

Sager, N. (1981) Natural Language Information Processing: A Computer Grammar of English and Its Applications, Addison-Wesley.

Woods, W. A. (1970) "Transition network grammars for natural language analysis," C. ACM, vol. 13, pp. 591-606.