REVERSIBLE LOGIC GRAMMARS FOR MACHINE TRANSLATION.

Marc DYMETMAN
Pierre ISABELLE

Centre Canadien de Recherches sur
L'Informatisation du Travail
1575 Boul. Chomedey
Laval,Quebec
CANADA H7V 2X2

ABSTRACT

The CRITTER translation system makes use of a single grammar to perform analysis and synthesis tasks. The formalism used is a variant of DCG (Definite Clause Grammars), in which annotations have been added to allow for dual compilations of the grammar into analysis and synthesis Prolog programs sharing the same declarative content.

These annotations are of two types: 1) annotations separating the declarative content of rules *(logic)* from goal-processing order *(control),* and 2) annotations which act as directives for the compiler(s) to perform "optimization" transformations on groups of rules making the target Prolog procedures better adapted to the - *analysis* or *synthesis* - task at hand.

1. THE TRANSLATION MODEL.

1.1 CRITTER

CRITTER is an experimental system that we are currently developing as a test-bed for our translation model. It is designed to translate from English to French (and conversely) reports concerning the meat trade market produced on a weekly basis by the Canadian Department of Agriculture.

The following sentences provide a short sample of the language of these reports:

Imports of slaughter cattle from the United States last week dropped 62% compared to the previous week, totalling 334 steers and 50 heifers.

La semaine dernière, les importations de bovins d'abattage ont chuté de 62% en regard de la semaine précédente, totalisant 334 bouvillons et 50 taures.

1.2 Principal features of the translation model.

We will briefly summarize the main features of the translation model. For a more detailed description, see <Isabelle et al., 1988>.

The translation model can be viewed as the composition of three relations:

- the source analysis/synthesis relation: *anasynt_s(T_S, SurfSyn_S, Sem_S)* which defines a set of well-formed triples where T_S is a source language text, SurfSyn_S and and Sem_S are respectively a surface syntactic structure and a semantic structure for this text, both being source language dependent;

- the target analysis/synthesis relation: *anasynt_t(T_T, SurfSyn_T, Sem_T)* which is the analogue of anasynt_s for the target language.

- the transfer relation: *tr(Sem_S, Sem_T)* which defines a set of couples where Sem_S and Sem_T are respectively source and target semantic structures which are considered to be translationally equivalent.

Formally, the *anasynt* relations are described in the framework of *definite clause grammars* (DCGs) <Pereira, Warren 1980>, to which *control annotations* have been added (see section 2.2).

As for the *tr* relation, it is defined through a set of clauses, obtained from the compilation of a *transfer dictionary* into Prolog.

1.3 Semantic Structures.

Consider as an example the (rather artificial) English sentence:

In Edmonton, John gave hogs to Mary.

Ignoring for the moment the surface syntactic structure, anasynt_s will assign the following semantic structure to this sentence:

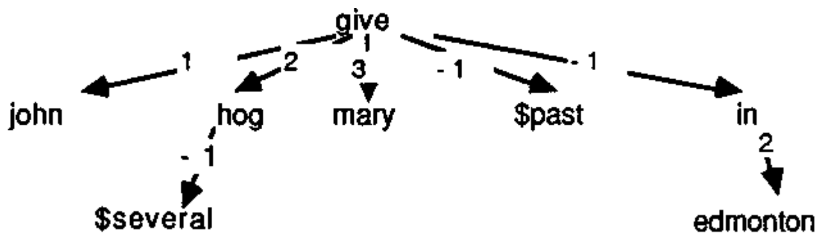


Fig.1

The labels 1,2,3 can be read as argument positions relative to the predicate 'give'. As for the (-i) labels ("inverse arguments"), they are a notational device that allows us to simultaneously represent *predicate-argument* relations as well as *subordination* relations.

For instance the 'in' node has two arguments: the first (-1) being the giving event, the second (2) being the location of this event. The "inverse" notation for the first argument is a way of making the semantic structure reminiscent of the fact that "in Edmonton" is syntactically a dependent of 'give'. (For a justification of this approach, see <Isabelle et al., 1988>).

2. REVERSIBLE LOGIC GRAMMARS.

2.1 Goals.

Consider once again the *anasynt* relation (for specificity, we shall discuss the *source* analysis/synthesis relation, but everything will apply mutatis mutandis to the target relation). The *declarative* reading of the anasynt_s relation is just that it is a set of triples of the form: <T,SurfSyn,Sem >. From a *computational* perspective, the challenge posed by reversibility is to write a *program* P having the following properties:

- inputting a specific text T to the program will result in the output of every Sem such that
  ∃SurfSyn anasynt(T,SurfSyn,Sem)
- inputting a specific Sem to the program will result in the output of every text T such that
  ∃SurfSyn anasynt(T,SurfSyn,Sem).

For instance, providing the program with the input *"In Edmonton, John gave hogs to Mary"* should result in the output of the semantic structure Sem of Fig. 1, and giving to the program Sem as input should result in the output of each of the paraphrases of Fig.2 :

In Edmonton, John gave hogs to Mary.
In Edmonton, John gave Mary hogs.
In Edmonton, hogs were given to Mary by John.
In Edmonton, hogs were given by John to Mary.
In Edmonton, Mary was given hogs by John.
John gave hogs to Mary in Edmonton.
John gave Mary hogs in Edmonton.
Hogs were given to Mary by John in Edmonton.
Hogs were given by John to Mary in Edmonton.
Mary was given hogs by John in Edmonton.

Fig.2

2.2 Some Basic Assumptions.

In CRITTER, the *anasynt* relation can be seen as an interface between the reversible logic grammar and the transfer component. If *s* is the main-sentence nonterminal - and assuming for the time being a DCG-type translation of nonterminals into clauses - it can be defined in the following way:

```
anasynt(String,SurfSyn,Sem) :-
        s(S,String,[]),
        syn(S,SurfSyn),
        sem(S,Sem).
```

Here, *syn* (resp. *sem)* are access predicates, which assign to *linguistic object S* some structure representing its surface syntax (resp. its semantic representation).

Generally speaking, nonterminals in the grammar appear under the uniform format:

nt(NT) --> etc...

where NT is some *linguistic object,* which contains information of different types accessed through "selectors" like *sem, syn, cat, number,* etc...

An important property of the grammar is its syntactic and semantic compositionality: all NTs are assigned one *syn* and one *sem,* and these are built out of the *syns* and *sems* of the daughters of NT in the derivation tree.

This dual syntactic/semantic aspect of linguistic objects is of basic importance for reversibility. It then makes sense e.g. to divide the task of synthesis into subtasks a) and b):

a) from the semantics of NT, find the possible semantics of its daughters.
b) from the semantics of the daughters, find their possible syntax.

In a grammar which lacked this property, for instance a grammar which performed a one-shot mapping from the global syntax of a sentence into a semantic representation, the computational problem of synthesis would be much more difficult

2.3 DCGs and Reversibility.

2.3.1 Reversibility through goal-ordering.

Consider the following example of a DCG rule, and its standard translation into a definite clause:

```
s(S)->                ───────────────────►   s(S,L1,L3):-
 np(NP),                                      np(NP,L1,L2),
 vp(VP),                                      vp(VP,L2,L3),
 {combine1(NP,VP,S)}.                         combine1 (NP,VP,S).
       (a)                                          (b)
```
Fig.3

In this rule, the *combinel* predicate condenses the relation between the three linguistic objects NP, VP and S. Leaving a more detailed description of the operation of this rule for later discussion (section 3), suffice it to say here that the *combinel* predicate:

- builds the syntax of S out of those of NP and VP,
- builds the semantics of S out of those of NP and VP in such a way that i) the semantics of S and that of VP are immediately related (in fact they are unifiable), and ii) the semantics of NP is determined as soon as the semantics and the syntax of VP are known.
- performs agreement checking between NP and VP.

As it stands in Fig. 3, the *s* rule is fine for analysis purposes:

In analysis, when s is called, L1 is known (it is the string of characters representing the sentence to be parsed), S and L3 being uninstantiated. Thus *np* is called with L1 known and returns with L2 and NP known. Therefore *vp* is called with L2 known and returns with L3. and VP known. Then *combine1* returns with S known, and finally s returns with L3 and S known [1].

On the other hand, attempting to use (b) for *synthesis* purposes leads to catastrophic computational behavior: in synthesis when *s* is called, only the semantics part Sem_S of S is known, L1 and L3 being uninstantiated. Then *np* is called with NP, L1 and L2 completely unknown. The best that can be hoped for then, barring looping, is for *np* to enumerate all noun phrases complete with syntax and semantics, waiting until *vp* and *combine1* to check these against the semantics of VP and S.

In fact, what we need in the synthesis case is simply to arrange for the calling order of *np, vp* and *combinel* to be as follows:

- i.) the call to *combine 1(NP,VP,S)* should come first. *At the time of call, Sem_S is known.* On return, Sem_VP will be known (it is unified to Sem_S), as well as the fact that the subject (suj) of VP is NP.
- ii.) the call to *vp(VP,L2,L3)* should come second. *At the time of call, Sem_VP is known.* On return, VP will be fully instantiated; its syntax will be known, and, furthermore, the semantics of each of the groups subcategorized for by the lexical head of VP (the syntactic arguments of the verb) will be known (see section 3 for details). In particular, the semantics of the subject of VP will be known, which is to say (by i.) that Sem_NP will be known. Also, L2 will be known: it has for prefix the string generated by the VP.
- iii.) the call to *np(NP,L1,L2)* should come last. *At the time of call, Sem_NP is known.* On return, NP will be fully instantiated, and L1 will be known: it has for prefix the string generated by the NP.

The fact that efficiency, or even termination, of a Prolog program (at least when using the standard depth-first search interpreter) is strongly dependent on goal calling order and instantiation state will hardly come as a revelation to anyone. However, a refresher of some of the horrors to be expected is given in Fig. 4.

[1] we are here really sketching an informal inductive proof that analysis will "work" as soon as any nonterminal goal is called with input list instantiated, and a "combination" call comes after nonterminal subgoals.

```
correct(L) :-            correct(L) :-            correct(L) :-
   ok(L),                   ok(L),                   peano(L),
   peano(L).                peano(L).                ok(L).


peano([]).               peano([]).               peano([]).
peano(L) :-              peano(L) :-              peano(L) :-
   L = [a|L'],              peano(L'),               L = [a|L'],
   peano(L').               L = [a|L'].              peano(L').

ok([foo]).               ok([foo]).               ok([foo]).
ok([a]).                 ok([a]).                 ok([a]).

    A                        B                        C
```

Behaviour of A, B, C on invocation of: *correct(L) ?* , where L is a variable:

- in A , the solution L = [a] is found, and there is finite failure on backtrack, as desired.
- in B, a loop occurs and no solution is found.
- in C, the solution L = [a] is found, but a loop occurs on backtrack.

Fig. 4

The *s* rule above is just one among several. What is general in the foregoing discussion is an *invariant* property of the *synthesis* process which has to be maintained:

> When some nonterminal goal *nt(NT,L1,L2)* is called during *synthesis,* the semantics part NT_Sem of NT is known at the time of call. *nt* finitely returns, and at that time NT is fully instantiated and (the string prefix generated by *nt* in) L1 is known.

In fact, the previous discussion of analysis shows that there is a corresponding invariant which is maintained during the *analysis* process:

> When some nonterminal goal *nt(NT,L1,L2)* is called during *analysis,* the input list L1 is known at the time of call. *nt* finitely returns, and at that time NT is fully instantiated and the output list L2 is known.

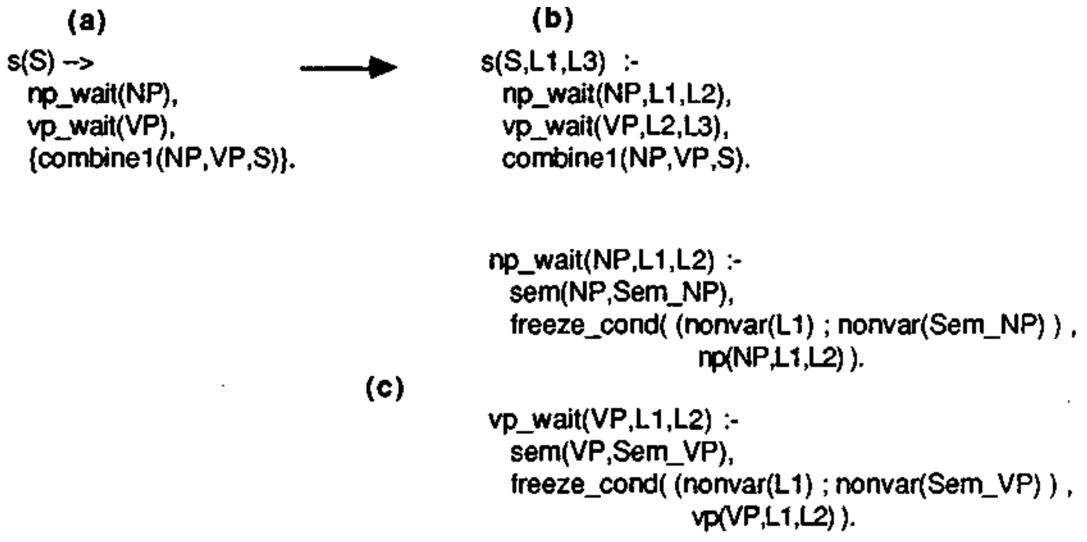
A more thorough analysis of the grammars already written for the analysis of French and the analysis of English convinced us that goal ordering was indeed the main obstacle to reversibility and that if we found a way to guarantee that *nt(NT,L1,L2)* was called only with L1 known (analysis) or when Sem_NT is known (synthesis), we would then get a realistic reversible grammar.

Two ways of ensuring that ordering were tested, and are presented below (section 2.2.2 and 2.2.3). For reasons explained in the sequel, the second alternative was eventually retained.

2.3.2 Dynamic goal-ordering through instantiation-driven control.

The first alternative was to keep the DCG formalism intact, and to compile the grammar into a Prolog program in the standard way. This Prolog program, however, makes heavy use of *goal-freezing,* an idea introduced in PrologII which permits asynchronous evaluation of goals *(demon firing)* when some precondition is met[1].

Fig. 5 gives an example of how this is made to work:



```
(a)                                  (b)
s(S) -->              ------>        s(S,L1,L3) :-
  np_wait(NP),                         np_wait(NP,L1,L2),
  vp_wait(VP),                         vp_wait(VP,L2,L3),
  {combine1(NP,VP,S)}.                 combine1(NP,VP,S).


                                      np_wait(NP,L1,L2) :-
                                        sem(NP,Sem_NP),
                                        freeze_cond( (nonvar(L1) ; nonvar(Sem_NP) ) ,
                                                     np(NP,L1,L2) ).
                          (c)
                                      vp_wait(VP,L1,L2) :-
                                        sem(VP,Sem_VP),
                                        freeze_cond( (nonvar(L1) ; nonvar(Sem_VP) ) ,
                                                     vp(VP,L1,L2) ).
```

Fig. 5

In the grammar, every nonterminal appears under two aspects: *nt* and *nt_wait. nt* appears as the head of rules, and *nt_wait* appears in the body of rules as in (a) of Fig. 5.

A rule is translated by the standard DCG compiler as in (b) of Fig. 5.

Finally instances of the following "clause schema":

```
nt_wait(NT,L1,L2) :-
sem(NT,Sem_NT),
    freeze_cond(  (nonvar(L1); nonvar(Sem_NT)),
                  nt(NT,Ll,L2)
               )
```

are added to the Prolog program, as in (c) of Fig. 5.

The meaning of such clauses is the following: when *nt_wait* is called, wait before calling *nt* itself (which implies making some nondeterministic rule choice) until some precondition is met, namely the condition that either L1 be known, or Sem_NT be known (where Sem_NT is the semantics part of NT). As soon as the precondition is met, "fire" *nt.*

The consequences of such a formulation are the following in the case of Fig. 5 :

[1] For an introduction to goal-freezing, the reader is referred to <Colmerauer 1982>. Unfortunately, most commercial Prologs do not as yet offer this facility, and we have had to implement it in the dialect of Prolog we use (Quintus Prolog) as efficiently as we could, i.e. *not* by meta-interpretation!

- in *analysis,* when *s* is called - with L1 known -, *np* will be ready to fire immediately, for its precondition is met, but *vp* will have to wait for L2 to be instantiated before firing. As for *combinel* it will be called immediately [1].

- in *synthesis,* when *s* is called - with Sem_S known -, neither *np* nor *vp* will be ready to fire, for neither Sem_NP nor Sem_VP will be known at that time. On the other hand, *combinel* will be called immediately and return with Sem_VP known. Then *vp* will fire, and when it finally returns (meaning here that all pending subgoals of *vp* will have been fired) VP will be completely known and L2 partially known (the "string of VP" will be completely known). As a consequence, Sem_NP will be known, so *np* will fire. When it finally returns, NP will be completely known, and L1 partially known (the "string of NP" will be completely known). Thus the string corresponding to *s* will have been generated.

We have done some experiments in this framework which have shown it to be a workable idea. Indeed we were able to analyze and synthesize simple sentences. However, we encountered some problems:

- debugging is difficult with asynchronous evaluation because it is difficult to trace a goal to its caller. This problem might perhaps be alleviated if one had tracing facilities adapted to goal-freezing.

- the runtime overhead to be paid for implementing goal-freezing in a language not specifically designed with this facility in mind is not negligible. Also, extended goal-freezing almost precludes optimized compilation of the Prolog program obtained from a grammar.

- More fundamental, although not well understood, is the fact that *chronological backtracking* (the standard backtracking scheme in Prolog) does not mesh very well with asynchronous goal evaluation: backtracking on some goal can undo a lot of useful work which was done on *previously called* but *logically unrelated* goals.

2.3.3 Static goal-ordering through double compilation.

• Order Annotations.

The problems mentioned in the preceding section led us to look for some other way to achieve the proper goal orderings for analysis and synthesis.

A natural alternative is to extend the DCG formalism in such a fashion that the grammar rule notation separates the *declarative content* of the rule from the *control information* corresponding to the order in which goals should be executed in different evaluation contexts. Indeed, if one reconsiders Fig. 3 above, one notices that the order in which nonterminals *np* and *vp* appear in rule (a) really plays two different roles:

- a specification of the order in which *np* and *vp* appear in the string, which is reflected in translation (b) by the respective associations of differential lists L1, L2 and L3 to *np* and *vp.*
- a specification of the order in which goals are going to be called by the Prolog interpreter.

In order to separate between these two types of ordering, we add *annotations* into the syntax of DCG rules. An example of such annotations is given in Fig. 6:

```
s(S) ->
   np(NP),                    #(3),
   vp(VP),                    #(2),
    {combine1 (NP.VP.S)},   #(1).
```

Fig. 6

[1] "immediately" here is to be understood in a particular way: the exact order of firing events will depend on the details of implementation of the goal-freezing mechanism. What is important in such an asynchronous scheme is not the exact order of calls, but rather the fact that a goal will not be fired before its precondition is met.

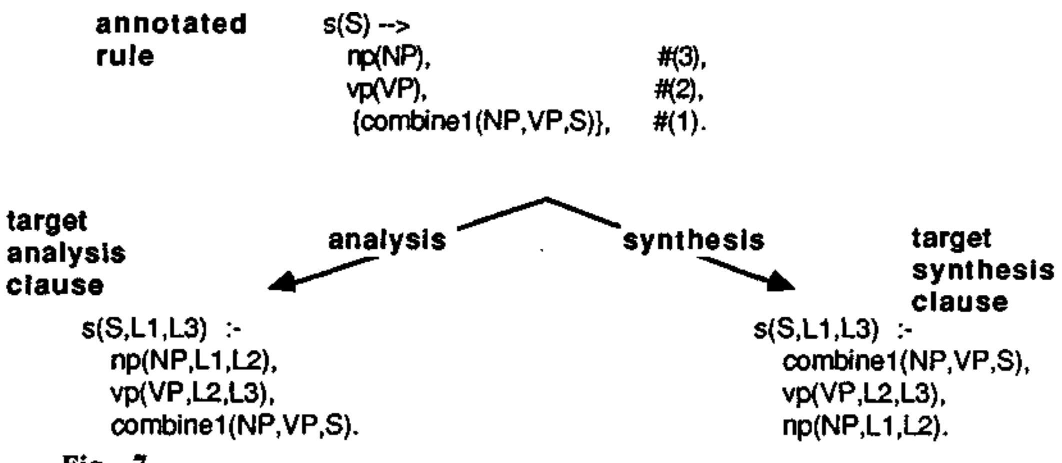
These annotations appear on the right hand-side of each goal, and are in the form *#(n),* where n is some number. Their intent is to specify the relative order in which goals are to be executed during *synthesis.* As for the order in which goals are to be executed during *analysis,* it is the same as the textual order of goals[1].

From these annotated rules, the grammar compiler will now perform two different tasks:

- translate the grammar into an analysis Prolog program.

- translate the grammar into a synthesis Prolog program.

An illustration of such a "double" compilation is given in Fig. 7:



```
s(S) -->
  np(NP),                 #(3),
  vp(VP),                 #(2),
  {combine1(NP,VP,S)},    #(1).
```

```
s(S,L1,L3) :-
  np(NP,L1,L2),
  vp(VP,L2,L3),
  combine1(NP,VP,S).
```

```
s(S,L1,L3) :-
  combine1(NP,VP,S),
  vp(VP,L2,L3),
  np(NP,L1,L2).
```

Fig. 7

The target analysis and synthesis clauses thus produced share the same declarative content, but will perform their calls in the appropriate order.

- **Handling left-recursion.**

There are further inadequacies of the DCG formalism from the point of view of reversibility that also need to be remedied.

One of these shortcomings has to do with rules such as the following:

```
np(NP) ->
  np(NP'),
  pp(PP),
  {combine2(PP,NP',NP)}.

np(NP) ->
  n1(N1).
```

Fig. 8

The intent of the *np* rule[2] given above is to incorporate within a *np* an unspecified number of prepositional

[1] Thus one has complete control over the order in which goals will be executed in analysis and synthesis, with the exception that *in analysis,* the order in which *nonterminal* goals will be executed has to be the same as the order of nonterminals in the string, which is what is normally needed.

[2] This example is given for illustrative purposes only and does not claim linguistic adequacy.

modifiers *(the hotel in Vancouver with a Chinese roof). n1* is the "bare" *np (the hotel),* and *combine2* builds the syntactic and semantic structures of NP from those of PP and NP'.

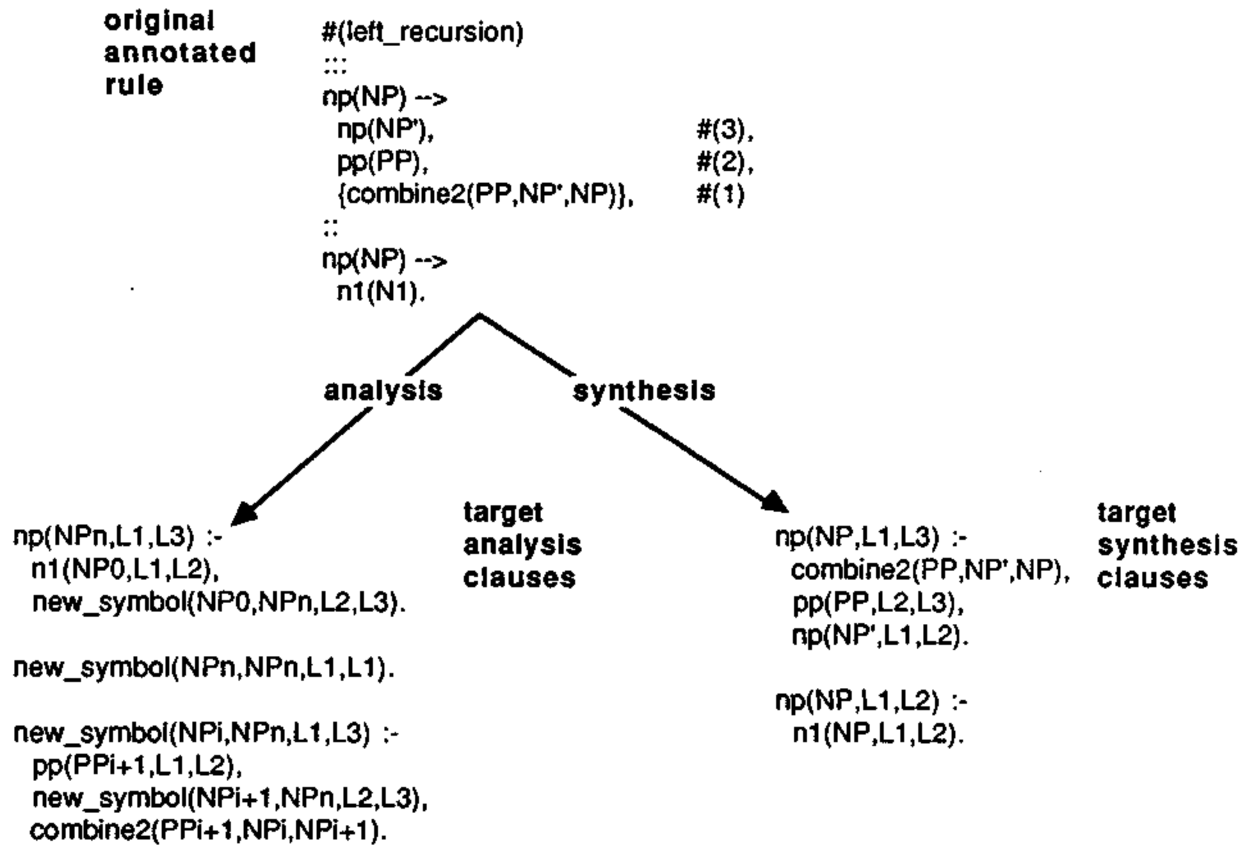
The obvious problem with this rule, however, is that it is "left-recursive": although its declarative reading is perfectly natural and directly represents the intended meaning of the rule, the standard DCG translation will result in a loop when used in analysis.

For this reason, no DCG programmer ever writes such rules in the way shown in Fig. 8, but rather she transforms them into some equivalent rules, which although more complex, will display the desired computational behavior (see infra Fig. 9).

On the other hand, it is interesting to observe that the rule of Fig. 8, when used in synthesis, will work perfectly well if only one takes care - as in the previous *vp* example - to order the *combine2* goal before the *pp* and *np* goals [1].

Fortunately, there is a way out of this difficulty: the grammarian should be allowed to write the *np* rule in the natural way, and the compiler should be made to perform the transformations needed for analysis and for synthesis.

This is precisely what is done in the CRITTER system, as illustrated in Fig. 9:



```
#(left_recursion)
:::
np(NP) -->
  np(NP'),                    #(3),
  pp(PP),                     #(2),
  {combine2(PP,NP',NP)},      #(1)
::
np(NP) -->
  n1(N1).
```

```
np(NPn,L1,L3) :-
  n1(NP0,L1,L2),
  new_symbol(NP0,NPn,L2,L3).

new_symbol(NPn,NPn,L1,L1).

new_symbol(NPi,NPn,L1,L3) :-
  pp(PPi+1,L1,L2),
  new_symbol(NPi+1,NPn,L2,L3),
  combine2(PPi+1,NPi,NPi+1).
```

```
np(NP,L1,L3) :-
  combine2(PP,NP',NP),
  pp(PP,L2,L3),
  np(NP',L1,L2).

np(NP,L1,L2) :-
  n1(NP,L1,L2).
```

Fig. 9

[1] It is quite ironic that - in our experience - if one writes the np rule in such a way as to eliminate left-recursion in analysis, then tries to reorder goals for use in synthesis, further difficulties reappear that make the whole process quite meaningless!

In the current implementation, it is necessary to indicate to the compiler where a left-recursive rule appears, and to flag the relevant block of rules. This explains the appearance of a *#(left_recursion)* directive, as well as that of the "separators"::: and:: . The analysis translator will then introduce a "dummy" nonterminal *new_symbol,* and perform a transformation leading to the target analysis clauses shown in the figure. Space is lacking here to explain in detail the transformation done, but it can be formally demonstrated that the resulting clauses keep the intended meaning of the rule unchanged [1].

As for the synthesis clauses, they are similar to the original rules, apart from the fact that the order annotations in these have been used to order subgoals, as in the case of Fig. 7.

• other notational devices.

Taking advantage of the flexibility gained by the introduction of a custom-made compiler, we add a notational device similar to the functional notation used in unification grammars <Sag et al., 1986>, namely the ability to refer to substructures of a linguistic object by way of a "dot" notation.

For instance, the rule of Fig. 6. is actually written in the following form:

```
s(S) -->
      np(NP),                        #(3),
      vp(VP),                        #(2),
      {S.head = VP.head,
       S.head.subcat.suj = NP,
       S.sem_form = VP.sem_form,
       NP.number = VP.number,
       …
      },                             #(1).
```

The compiler (in the case of analysis as well as synthesis) will then replace e.g. "S.head" by a variable Shead, add to the generated clause the constraint "head(S,Shead)" (and similarly for the other cases), and perform the unifications required by the equalities appearing in the rule.

3. AN EXAMPLE.

We now turn to an example which illustrates, albeit in a simplified manner, some essential features of our approach to reversibility. We concentrate on the problem of parsing and synthesizing sentences such as (1):

(1) Jack seems to like Jill.

The verb "seem" belongs to a class known as "raising verbs". According to a well-established analysis of those structures, the semantic representation associated with (1) would be something like (2):

(2) seem'(like'(jack,jill))

That is, the surface subject "Jack" is not interpreted as an argument of seem'. Rather seem' is viewed as a propositional operator, and "Jack" is an argument in the predication expressed by the infinitival complement.

In our grammatical model, such peculiarities of the syntax/semantics mapping are accounted for in the lexical component, in a way quite similar to that of <Pollard and Sag, 1988>. The lexicon assigns to "seem" a feature structure VB with the following properties:

[1] The process is akin to techniques for putting a context-free grammar into some normal form. The delicate part, however, is the way in which the constraints between variables should be transferred.

```
(3) VB.cat=vb                                        (a)
    VB.cit_form = seem                               (b)
    VB.head.subcat = [NP, VCOMP]                     (c)
    NP.cat = np                                      (d)
    VCOMP.cat = vp                                   (e)
    VCOMP.head.form = infinitive                     (f)
    VCOMP.head.sem_form = A                          (g)
    VCOMP.head.subcat = [VPSUBJ | X]                 (h)
    VPSUBJ.head.sem_form = NP.head.sem_form          (i)
    VB.head.sem_form = seem'(A)                      (j)
    inflect(seem, NP.agree, VB.inflected_form)       (k)
```

In the description of (3), equations (c) through (j) assign to "seem" a *head* structure which, under the action of our syntax rules, will percolate up the parse tree until the maximal constituent of which "seem" is the head is reached. This head structure contains a "subcat" substructure [NP, VP](further described in (d) through (i)) and a semantic form seem'(A). The *inflect* predicate establishes the inflected form of the verb on the basis of its citation form and agreement attributes (for simplicity, we ignore tense here).

The subcat substructure accounts for the syntactic dependents of "seem" both in terms of their syntactic properties and in terms of their contribution to the larger semantic structure. More specifically, "seem" requires a subject noun phrase and an infinitival complement The latter is associated with a semantic object A which happens to be unified with the unique argument of seem' -- like'(jack,jill) in the case of sentence (1). Equations (h) and (i) bind the semantic value of the subject of "seem" with the sem_form associated with the subject slot of the main verb of the infinitival complement. This accounts for the fact that this complement is "controlled" by the subject of "seem". During the synthesis of the complement, the semantic value attached to its subject slot -- 'jack' in the case of sentence 1- will become known. By the same token, the sem_form of the subject of "seem" will become instantiated.

The lexical specification of (3) provides an example of an important feature of "lexicalist" models: in a syntactic phrase, the mapping between syntactic configurations (or syntactic functions) and argument places in semantic structures is determined by lexical properties of the head.

We write grammar rules such as the following:

(4)

```
s(S) -->
   {S.cat=s},
   {S.daughters = [NP, VP]},
   {S.head = VP.head},
   {S.head.subcat = [NP | _]}.
   np(NP),
   vp(VP).

np(NP) -->
  det(DET),
    ... etc.

np(NP) -->
   {NP.cat=np},
   {NP.daughters = [NPR]},
   {NP.head = NPR.head},
   npr(NPR).
```

```
vp(VP)-->
   {VP.cat = vp)},
   {VP.daughters = [VB, COMP])},
        {VP.head = VB.head},
   {VP.head.subcat = [_, COMP]},
   vb(VB),
   complement(COMP).

complement(COMP) -->
   np(COMP).
complement(COMP) -->
   pp(COMP).
complement(COMP) -->
   vp(COMP).
```

The s rule, for example, will be compiled into the following clause:

```
(5) s(S,L1,,L2):-
        cat(S, s),
        daughters(S, [NP, VP]),
        head(S, H), head(VP, H),
        subcat(H, [NP | _]),
        np(NP, L1, L3),
        vp(VP, L3, L2).
```

The resulting program will be capable of parsing sentence (1) in an efficient manner. A call to the goal s(S, "Jack seems to like Jill","") will succeed, instantiating S to an object having the intended properties. In particular, it will be established that:

(6) S.head.sem_form = seem'(like'(jack, jill))

In order for the parser to get that result, equation (i) of lexical entry (3) plays a critical role. Binding the sem_form of the subject of "seem" with the sem_form of the subject of its complement produces the correct "lowering" effect.

Now suppose we want to use the same program for synthesis. In that case, we have to demonstrate s(S, L1,""), and all we know about S is that (6) holds. The problem is that the grammar starts by firing np(NP, L1, L2) while NP, L1, and L2 are all uninstantiated variables. The program may well keep enumerating well-formed np's forever since some of the rules expanding np's are likely to provide for recursive modifier structures. Generally speaking, it makes little computational sense to attempt the synthesis of a substructure whose semantic form is not yet known.

There is no way to determine the sem_form of that noun phrase by a mere examination of the sem_form of the sentence. What is required is to find in the dictionary a lexical item whose sem_form matches that of the sentence; the relevant dictionary entry will then determine the mapping of semantic arguments onto syntactic positions. The predicate seem' gives access to the entry of the verb "seem" where it is stipulated: 1) that the unique argument of the predicate seem' is to be associated with an infinitival complement on the verb, and 2) that the subject of the verb is bound to the subject slot of the infinitival complement.

With other verbs, the situation could be different. The active form of a simple transitive verb generally maps the first argument of the corresponding predicate onto the subject position. With the passive form of the same verb, it is rather the second argument that is associated with the subject position. With so-called "tough-movement" adjectives, the syntactic subject is associated with a non-subject syntactic slot in the infinitival complement:

(7) a) Mary is difficult to convince.
    b) difficult'(convince'(one, mary))

Given the requirement that the sem_form of a phrase should be known before attempting synthesis, there is no choice but to synthesize the verb phrase before the subject noun phrase. For that purpose, we annotate the s rule above as follows:

```
(8) s(S) -->
        (S.cat = s,
         S.daughters = [NP, VP]),
         S.head = VP.head,
         S.head.subcat = [NP | _] },          #(1),
        np(NP),                               #(3),
        vp(VP),                               #(2).
```

The compiler described in section 2.3.3 will compile rule (8) into the synthesis clause (9):

```
(9) s(S, L1. L2) :-
        cat(S, s),
        daughters(S, [NP, VP),
        head(S, H), head(VP, H),
        subcat(H, [NP |_]),
        vp(VP, L3, L2),
        np(NP, L1, L3).
```

The goal s(S, L1, L2) with only equation (6) known will trigger the following sequence of unifications:

The s rule is started:

```
(a)     S.head.sem_form = seem'(like'(jack,jill))
(b)     S.daughters = [NP, VP]
(c)     VP.head = S.head
```

The vp rule is then called:

```
(e)     VP.daughters = [VB, COMPL]
(f)     VB.head = VP.head
(g)     VP.head.subcat = [ NP, VCOMP]
```

Vp calls the vb rule, which performs a dictionary lookup:

```
(h)     VB.cat = vb
(i)     VB.cit_form = seem
(j)     VB.head.subcat = [NP, VCOMP]
```

"Seem" takes an infinitival complement whose sem_form is bound to the unique argument of seem':

```
(k)     VCOMP.head.sem_form = like'(jack,jill)
```

The subject slot of the infinitival complement of "seem" and the subject of "seem" share the same semantic content:

```
(1)     VCOMP.head.subcat = [NP2 | X]
(m)     NP2.sem_form = NP.sem_form
```

Since the syntactic form of the subject of "seem" is not yet known, the agreement values on the verb are not known. The process that inflects the verb has to be postponed[1]:

```
(n)     VB.inflected_form = ??
```

WAIT until NP.agree is instantiated:

inflect(seem,NP.agree,VB.inflected_form)

The vb rule returns, vp calls the complement rule which makes a new call to the vp rule, to synthesize the infinitival complement of "seem":

```
(o)     VCOMP.head = VB2.head
(p)     VB2.form_cit = like
(q)     VB2.head.subcat = [NP2, NP3]
(r)     NP2.sem_form = NP.sem_form = jack
(s)     NP3.sem_form = jill
```

[1] This is done through the mechanism of *goal-freezing* (see section 2.3.2).

The synthesis of the vp "to like Jill" is complete. Now the sem_form of the subject of "seem" is known: it has been equated with the sem_form of subject slot of "like", which has itself been bound to the first argument of the predicate like', namely 'jack'.

We now return to the larger vp. Its synthesis is also complete, except for the process dealing with the inflection of "seem" that remains on the waiting list of *frozen* goals.

We then return to the level of s, which fires the subject np goal:

(t) NP.cit_form = 'Jack'
(u) NP.agree.pers = p3
(v) NP.agree.numb = sing

The synthesis of the subject np is complete. Since NP.agree is now known, the delayed goal can now be "melted" (i.e. called):

```
(n')                    MELT:
                              inflect(seem,NP.agree,VB.inflected_form)

                        VB.inflected_form = seems
```

(o) the goal s(S, L1, L2) succeeds with:
L1 = "Jack seems to like Jill".

4. CONCLUSION

After experimenting with different approaches to the problem of reversibility in the framework of logic grammars, we have found that the solution of annotating rules with directives to perform compile-time goal ordering and global transformations - resulting in an analysis program and a synthesis program with the same grammatical content - was the most useful.

This approach gives the linguist full control over the processing behaviour of the grammar - both in analysis and synthesis - without compromising the declarative perspicuity of grammatical specifications.

Through the implementation of this idea, the CRITTER translation system is currently capable of translating moderately complex sentences from English to French and vice-versa, using only one grammar for each language (the reversibility of the whole CRITTER system depends also on the reversibility of components other than the grammars, like the morphological and the transfer component, which have not been described in this paper).

Using the system bidirectionnally - both in translational mode and especially in monolingual mode (analyzing and then "resynthesizing" a sentence to produce its paraphrases) - has had some unexpected but interesting consequences:

- It often happens that mistakes or omissions in grammatical description which can easily go unnoticed in analysis mode (because one doesn't naturally submit sentences exhibiting them) become immediately apparent when grammatically ill-formed paraphrases are synthesized.
A reversible grammar thus acts as a powerful debugger to itself!

- A grammar RG which is reversible has to be much closer to observational adequacy than would have to be two different grammars, one for analysis (AG), the other for synthesis (SG). This is because AG can be overly permissive on the strings it accepts, since it can depend to a large degree on the well-formedness of the input text[1]. On the other hand SG can be overly ungenerous on the strings it generates, for it has to guarantee only the well-formedness of its output, not that it will output all possible variations in the expression of the same content. On the other hand, working with a reversible grammar RG does put some pressure on the linguist, who is obliged to fine tune his descriptions to match linguistic reality. It is difficult at this point to assess whether this will finally be seen as a handicap or as a bonus.

As the previous remarks show, the problem of grammar reversibility really has two different facets:

The problem of *computational* reversibility : how to computationally perform analysis and synthesis tasks from the same declarative grammar. This problem has substantially been solved in the case of DCG grammars by the techniques presented here. However, some work remains to be done to extend this treatment to handle unbounded dependencies, as in extraposition grammars <Pereira 1981>.

The problem of *linguistic* reversibility : is it *theoretically* correct to use one and the same grammar to describe the linguistic aspects of analysis and of synthesis, and if so, is this *practically* feasible, or does it put unrealistic constraints on linguistic descriptions? On this question, which the present paper has hardly touched upon, much work remains to be done.

ACKNOWLEDGMENTS

We would like to thank Elliott Macklovitch, Jean-Luc Cochard, Ron Ferguson, François Perreault and Michel Simard for their comments and suggestions.

REFERENCES

Appelt, D. (1987) "Bidirectional Grammars and the Design of Natural Language Generation Systems", in Theoretical Issues in Natural Language Processing. New Mexico State University.

Colmerauer A. (1982) PROLOG II: Manuel de référence et modèle théorique. Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille.

Isabelle P., Dymetman M., Macklovitch E. (1988) "CRITTER: a Translation System for Agricultural Market Reports", to appear in Proceedings of COLING-88.

Landsbergen, J (1987) Montague Grammar and Machine Translation. Philips Research M.S. 14.026, Eindhoven.

Pereira, F. (1981) "Extraposition Grammars", in Computational Linguistics 7.4, 243-256.

Pereira F., Warren D.H.D. (1980) "Definite Clause Grammars for Natural Language Analysis", Artificial Intelligence. 13, 231-278.

Pollard, C., Sag, I. (1988) Information-Based Syntax and Semantics, vol. 1: Fundamentals. CSLI lecture Notes no 13, CSLI, Stanford University.

Sag I., Kaplan R., Karttunen L., Kay M., Pollard C., Shieber S., Zaenen A. (1986) "Unification and Grammatical Theory", in Proceedings of the West Coast Conference on Formal Linguistics. Stanford Linguistics Association, Stanford University.

Zaharin Y, (1987) "String-Tree Correspondence Grammar a declarative formalism for defining the correspondence between strings of terms and tree structures", in Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics. Copenhagen.

[1] Of course, this doesn't come free: underspecification of AG will eventually show up under the ugly face of false syntactic ambiguities, even when the input text can be relied on to be well-formed!