A Symmetrical Approach to Parsing and Generation

Marc Dymetman, Pierre Isabelle and François Perrault

CCRIT, Communications Canada. 1575 Bld Chomedey. Laval (Québec) H7V 2X2 CANADA

Abstract. *Lexical Grammars* are a class of unification grammars which share a fixed rule component, for which there exists a simple left-recursion elimination transformation. The parsing and generation programs are seen as two dual non-left-recursive versions of the original grammar, and are implemented through a standard top-down Prolog interpreter. Formal criteria for termination are given as conditions on lexical entries: during parsing as well as during generation the processing of a lexical entry consumes some amount of a *guide*; the guide used for parsing is a list of words remaining to be analyzed, while the guide for generation is a list of the semantics of constituents waiting to be generated.

1. Introduction

Symmetry between parsing and generation. There is a natural appeal to the attempt to characterize parsing and generation in a symmetrical way. This is because the statement of the problem of reversibility is naturally symmetrical: parsing is concerned with recovering semantic content from phonological content, generation phonological content from semantic content. It has been noted by several researchers ([S88], [N89], [SNMP89]) that certain problems (left-recursion) and techniques (left-corner processing, linking, Earley deduction) encountered in the parsing domain have correlates in the generation domain. It is then natural to try and see parsing and generation as instances of a single paradigm; [S88] and [DI88, DI90] are attempts in this direction, but are hindered by the fact that there is no obvious correlate in generation of the string indexing techniques so prominent in parsing (string indices in chart parsing, differential lists in DCG parsing).

Guides. What we propose here is to take a step back and abstract the notion of string index to that of a *guide*. This general notion will apply to both parsing and generation, but it will be instantiated differently in the two modes. The purpose of a guide is to orient the proof procedure, specific to either parsing or generation, in such a way that: (i) the guide is initialized as a direct function of the input (the string in parsing, the semantics in generation), (ii) the current state of the guide strongly constrains the next access to the lexicon, (iii) after lexical access, the size of the guide strictly decreases (*guide-consumption condition*, see section 3). Once a guide is specified, the generation problem (respectively the parsing problem[1]) then reduces to a problem formally similar to the problem of parsing with a DCG [PW80] containing no empty productions[2] (ie rules whose right-hand side is the empty string []).

Several parsing techniques can be applied to this problem; we will be concerned here with a top-down parsing approach directly implementable through a standard Prolog interpreter. This approach relies on a *left-recursion-elimination transformation* for a certain class of definite clause programs (see section 3).

The ability to specify guides, for parsing or for generation, depends on certain compositionality hypotheses which the underlying grammar has to satisfy.

Hypotheses on compositionality. The parsing and generation problems can be rendered tractable only if certain hypotheses are made concerning the composition of linguistic structures. Thus generation can be arduous if the semantics associated with the composition of two structures is the unrestricted lambda-application[3] of the first structure's semantics on the second structure's semantics; this is because knowledge of the mother's semantics does not constrain in a usable way the semantics of the daughters.[4] On the contrary, parsing is greatly simplified if the string associated with the composition of two structures is the concatenation of the strings associated with each structure; one can then use string indexing to orient and control the progression of the parsing process, as is done in DCG under the guise of "differential lists".

Lexical Grammar. The formalism of *Lexical Grammar* (*LG*) makes explicit certain compositionality hypotheses which ensure the existence of guides for parsing as well as for generation.

A Lexical Grammar has two parts: a (variable) lexicon and a (fixed) rule component. The rule component, a definite clause specification, spells out basic linguistic compositionality rules: (i) how a well-formed linguistic structure A is composed from well-formed structures B and C; (ii) what are the respective statuses of B and C (*left* constituent vs *right* constituent, syntactic *head* vs syntactic *dependent*, semantic *head* vs semantic *dependent*); and (iii) how the string (resp. semantics, subcategorization list, ...) associated with A is related to the strings (resp. semantics, subcategorization lists, ...) associated with B and C (see section 2).

The ability to define a guide for parsing is a (simple) consequence of the fact that the string associated with A is the concatenation of the strings associated with B and C[5]. The ability to define a guide for generation is a (less simple) consequence of LG's hypotheses on subcategorization (see sections 2 and 4).

[1] This half of the statement may seem tautological, but it is not: see the attempt at a reinterpretation of left extraposition in terms of guides in section 5.

[2] Also called *null rules* [H78].

[3] By unrestricted lambda-application, we mean functional application followed by rewriting to a normal form.

[4] In theories favoring such an approach (such as GPSG [GKPS87]), parsing may be computationally tractable, but generation does not seem to be. These theories can be questioned as plausible computational models, for they should be judged on their ability to account for production behavior (generation) as well as for understanding behavior (parsing).

[5] A fairly standard assumption. If empty string realizations are allowed, then extraposition can still be handled, as sketched in section 5.

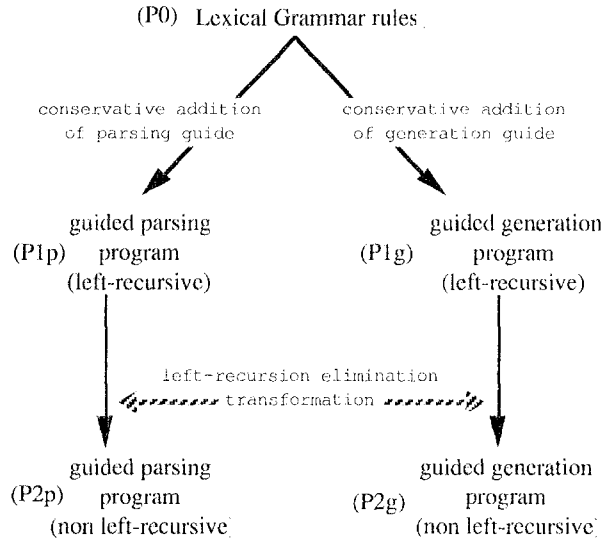


Fig. 1. A symmetrical approach to parsing and generation: paper overview

Parsing and Generation with Lexical Grammar. Fig. 1 gives an overview of our approach to parsing and generation. Let us briefly review the main points:

- (P0) is a definite clause specification of the original LG rules. It contains a purely declarative definition of linguistic compositionality, but is unsuitable for direct implementation (see section 2).
- (P1p) (resp (P1g)) is a *guided conservative extension* of (P0) for parsing (resp. for generation); that is, (P1p) (resp (P1g)) is a specification which describes the same linguistic structures as (P0), but adds a certain redundancy (*guiding*) to help constrain the parsing (resp. generation) process. However, these definite clause programs are not yet adequate for direct top-down implementation, since they are left-recursive (see section 3).
- (P1p) and (P1g) can be seen as symmetrical instantiations of a common program schema (P1); (P1) can be transformed into (P2), an equivalent non-left-recursive program schema (see section 3).
- (P2p) (resp (P2g)) is the non-left-recursive version of (P1p) (resp. (P1g)). Under the guide-consumption condition, it is guaranteed to terminate in top-down interpretation, and to enumerate all solutions to the parsing (resp. generation) problem (see section 4).

For lack of space, theorems are stated here without proofs; these, and more details, can be found in [D90b].

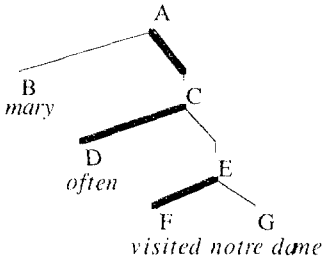
2. Lexical Grammar

Rule component The fixed rule component of LG (see Fig. 3) describes in a generic way the combination of constituents. A constituent A is either lexically specified (second clause in the *phrase* definition), or is a combination of two constituents B and C (first clause in the *phrase* definition). B and C play complementary roles along the following three dimensions:

- *combine_strings* : B is to the *left* of C in the surface order, or conversely to the *right* of C. This information is attached to each constituent through the *string_order* feature.
- *combine_syns* : B is the *syntactic-head* and C the *syntactic-dependent*, or conversely (*syn_order* feature).
- *combine_sems* : B is the *semantic-head* and C the *semantic-dependent*, or conversely (*sem_order* feature).

Because B and C play symmetrical roles[6], these seemingly eight combinations actually reduce to four different cases. To avoid duplicating cases, in the definition of the *phrase* predicate, the symmetry has been "broken" by arbitrarily imposing that B be the left constituent.[7]

Fig. 2 gives an example of a derivation tree in LG, using the lexicon of Fig. 4.



A.subcat	= []	A.sem	= C.sem
B.subcat	= []		= D.sem
C.subcat	= [B]		= often(visit(mary,nd))
D.subcat	= [E]	E.sem	= F.sem
E.subcat	= [B]		= visit(mary,nd)
F.subcat	= [G,B]	B.sem	= mary
G.subcat	= []	G.sem	= nd

Fig. 2. A derivation in LG
(heavy lines correspond to semantic-heads)

Our notion of *semantic-head* is a variant of that given in [SNMP89], where a daughter is said to be a semantic-head if it shares the semantics of its mother. The *combine_sems* predicate is responsible for assigning *sem_head* status (versus *sem_dep* status) to a phrase, and for imposing the following constraints:

i. the semantic-head shares its semantics with its mother,
ii. the semantic-head always *subcategorizes* its sister ((b) in Fig. 3),
iii. the mother's *subcategorization list* is the concatenation of the semantic-dependent list and of the semantic-head list minus the element just incorporated ((c) in Fig. 3).[8]

The subcategorization list attached to a constituent X corresponds to constituents higher in the derivation tree which are expected to fill *semantic* roles inside X. Subcategorization lists are percolated from the lexical entries up the derivation tree according to iii.

[6] Remark: the rules are *not* DCG rules, but simply *definite* (or *Horn*) *clauses*.

[7] If line (a) in the definition of *phrase* were omitted, the same *set* of linguistic structures would result, but some structures would be described twice. Line (a) is simply one means of eliminating these spurious ambiguities. The same effect would be produced by replacing (a) by *B.sem_order = sem_head* or by *B.syn_order = syn_head*.

[8] In fact, because of the constraints imposed by *combine_syns* (see discussion below) one of these two lists has to be empty.

```
phrase(A) :- phrase(B), phrase(C),                    (a)
    B.string_order = left,
    combine(B,C,A).
phrase(A) :- term(A).

combine(B,C,A) :-
    (combine_strings(B,C,A);combine_strings(C,B,A)),
    (combine_syns(B,C,A);combine_syns(C,B,A)),
    (combine_sems(B,C,A);combine_sems(C,B,A)).
combine_strings(B,C,A) :-
    B.string_order = left, C.string_order = right,
    append(B.string,C.string,A.string).
combine_sems(B,C,A) :-
    B.sem_order = sem_head, C.sem_order = sem_dep,
    A.sem = B.sem,
    B.subcat = [C|Rest],                              (b)
    append(C.subcat,Rest,A.subcat).                   (c)
combine_syns(B,C,A) :-
    B.syn_order = syn_head, C.syn_order = syn_dep,
    A.cat = B.cat,
    ( B.sem_order = sem_head, C.subcat = []
                    % complement
    ; C.sem_order = sem_head, C.subcat = [ _ ]).
                    % modifier
```

Fig. 3. The rules of Lexical Grammar[9]

Semantic-heads need not correspond to *syntactic-heads*. In the case of a *modifier* like *often*, *in paris*, or *hidden by john*, the modifier phrase, which is the syntactic-*dependent*, is the semantic-*head* and semantically subcategorizes its sister: thus, in the example of Fig. 2, the modifier phrase *D* semantically subcategorizes its sister *E*; *combine_sems* has then the effect of unifying the semantics of *E* (*visit(mary,nd)*) to the substructure *X* in the semantics (*often(X)*) attached to *D* (see the lexical entry for *often* in Fig. 4). This is reminiscent of work done in categorial grammar (see for instance [ZKC87]), where a modifier is seen as having a category of the form *A/A*, and acts as a functor on the group it modifies.

The *combine_syns* predicate is responsible for assigning *syn_head* status (versus *syn_dep* status) to a phrase, and for ensuring the following constraints:

i. The category *cat* of the syntactic-head is transmitted to the mother. The category of a phrase is therefore always a projection of the category (*n*,*v*,*p*,*a*,...) of some lexical item.
ii. When the syntactic-dependent is the same as the semantic-dependent, then the syntactic-dependent is semantically saturated (its *subcat* is empty). This is the case when the syntactic-dependent plays the syntactic role of a *complement* to its syntactic-head.
iii. When the syntactic-dependent is the same as the semantic-head, then the syntactic-dependent's subcat contains only one element[10]. This is the case when the syntactic-dependent plays the syntactic role of a *modifier* to its syntactic-head.

The lexicon in LG Because LGs have a fixed rule component, all specific linguistic knowledge

```
term(T) :- T.sem = mary,
    T.string = [mary],
    T.cat = n, T.subcat = [].
term(T) :- T.sem = notre_dame,
    T.string = [notre,dame],
    T.cat = n, T.subcat = [].
term(T) :- T.sem = paris,
    T.string = [paris],
    T.cat = n, T.subcat = [].
term(T) :- T.sem = die(S.sem),
    T.string = [died],
    T.cat = v, T.subcat = [S],
    S.string_order = left,
    S.cat = n, S.syn_order = syn_dep.
term(T) :- T.sem = visit(S.sem,O.sem),
    T.string = [visited],
    T.cat = v, T.subcat = [O,S],
    S.string_order = left, S.cat = n,
    S.syn_order = syn_dep,
    O.string_order = right, O.cat = n,
    O.syn_order = syn_dep.
term(T) :- T.sem = in(S.sem,O.sem),
    T.string = [in],
    T.cat = p, T.subcat = [O,S],
    S.string_order = left, S.cat = v,
    S.syn_order = syn_head,
    O.string_order = right, O.cat = n,
    O.syn_order = syn_dep.
term(T) :- T.sem = often(S.sem),
    T.string = [often],
    T.cat = adv, T.subcat = [S],
    S.string_order = _,     % may be left or right
    S.cat = v, S.syn_order = syn_head.
```

Fig. 4. Lexical entries in LG[11]

is contained in the lexicon. Fig. 4 lists a few possible lexical entries.

Consider a typical entry, for instance the entry for *in*. This entry specifies a possible leaf *T* of a derivation tree. *T* has the following properties:

i. *T* has string [in], and is of category *p* (preposition).
ii. *T* semantically subcategorizes two phrases: *O* (the object of the preposition), of category *n*, and *S* (the "implicit subject" of the preposition), of category *v*. By the general constraints associated with *combine_sems*, this means that *S* and *O* will both have semantic-dependent status.
iii. In the surface order, *S* is to the left of its semantic-head, while *O* is to the right of its semantic-head.
iv. The semantics *in(S.sem,O.sem)* of *T* is obtained by unification from the semantics of its subcategorized constituents *S* and *O*.
v. *S* is constrained to having syntactic-head status, and *O* to having syntactic-dependent status. Because of the constraints imposed by *combine_syns*, this means that *O* will be a syntactic complement of the preposition, and that the prepositional phrase will be a modifier of its "subject" *S*.

Idioms. The lexical apparatus allows for a direct account of certain types of idiomatic constructions. For instance, if the lexical entries of Fig. 5 are added to the

[9] Here, as in the sequel, we have made use of a "dot notation" for functional access to the different features of a linguistic structure *A*; for instance, *A.cat* represents the content of the *cat* feature in *A*.

[10] The "external argument" of the modifier, identified with the semantic-dependent by the semantic combination rule.

[11] For reasons of exposition, the contribution of the tense to the semantics of verbs is ignored here.

lexicon, then the expression "X kicked the bucket" will be assigned the semantics *die(X)*. Entry (a) expresses the fact that (in its idiomatic use), the verb form *kicked* subcategorizes for a subject *S* and an object *O* whose semantics is *the_bucket*, and is itself assigned the semantics *die(S.sem)*.

```
term(T) :- T.sem = die(S.sem),                    (a)
      T.string = [kicked],
      T.cat = v, T.subcat = [O,S],
      S.string_order = left, S.cat = n,
      S.syn_order = syn_dep,
      O.string_order = right, O.cat = n,
      O.syn_order = syn_dep,
      O.sem = the_bucket.
term(T) :- T.sem = the_bucket,                    (b)
      T.string = [the,bucket],
      T.cat = n, T.subcat = [].
```

Fig. 5. Idioms in LG

3. Guides and left-recursion elimination

Guides. Consider a finite string l_1, and let l_2 be a proper suffix of l_1, l_3 be a proper suffix of l_2, and so on. This operation can only be iterated a finite number of times. The notion of *guide-structure* generalizes this situation.

DEFINITION 3.1. *A guide-structure is a partially ordered set* **G** *which respects the* descending chain condition, *i.e the condition that in* **G** *all strictly decreasing ordered chains* $l_1 > l_2 > \ldots > l_i > \ldots$ *are finite.*

Consider now the following elementary definite clause program (P0)[12]:

```
a(A) :- a(B), D(B,A).                             (P0)
a(A) :- t(A).
```

We assume here that $\mathcal{D}$ is an abbreviation which stands for a disjunction $(C_1;\ldots;C_k)$ of conjunctions C_i of goals of the form $a(A)$, $t(A)$, or $\{T=S\}$ (*unification goals*) where the T, S are variables or partially instantiated terms. Among the variables appearing inside $\mathcal{D}$, only the "interface" variables A, B are explicitly mentioned. We further assume that the defining clauses (not shown) for the t predicate have right-hand sides which are conjunctions of term unification goals $\{T=S\}$. We call t the *lexicon predicate*, and a the *generic nonterminal predicate*.

Consider now the following program (P1), called a *guided extension* of (P0):

```
a'(A,L_in,L_out) :- a'(B,L_in,L_inter),           (P1)
      D'(B,A,L_inter,L_out).
a'(A,L_in,L_out) :- t'(A,L_in,L_out).
```

(P1) is obtained from (P0) in the following way: (i) *guide variables* (L_{in}, L_{inter}, L_{out}) have been *threaded* throughout (P0), and (ii) the 1-predicate t has been replaced by a 3-predicate t' which is assumed to be a *refinement* of t, ie, *for all* A, L_{in}, L_{out}, $t'(A,L_{in},L_{out})$ *implies* $t(A)$.

Program (P1) is a more constrained version of program (P0): t' can be seen as a version of t which is able to "consult" L_{in}, thus constraining lexical access at each step. We will be interested in programs (P1) which respect two conditions: (i) the *guide-consumption condition*, and (ii) the *conservative extension condition*.

DEFINITION 3.2. *Program (P1) is said to satisfy the guide-consumption condition iff: (i) the guide variables take their values in some guide-structure* **G**, *and (ii) any call to* $t'(A,L_{in},L_{out})$ *with* L_{in} *fully instantiated returns with* L_{out} *fully instantiated and strictly smaller in* **G**.

DEFINITION 3.3. *Program (P1) is said to be a conservative extension of (P0) iff:* $a(A)$ *is provable in (P0)* $\Leftrightarrow$ *there exist* L_{in},L_{out} *such that* $a'(A,L_{in},L_{out})$ *is provable in (P1).*

The $\Leftarrow$ part of the previous definition is automatically satisfied by any program (P1) defined as above. The $\Rightarrow$ part, on the other hand, is not, but depends on further conditions on the refinement t' of t. Saying that (P1) is a conservative extension of (P0) is tantamount to saying that (P1) adds some *redundancy* to (P0), which can be computationally exploited to constrain processing.

Left-recursion elimination[13]**.** Program (P1) is left-recursive: in a top-down interpretation, a call to a' will result in another immediate call to a', and therefore will loop. On the other hand the following program (P2) is not left-recursive, and Theorem 3.4 shows that it is equivalent to (P1):

```
a'(A_n,L_in,L_n) :- t'(A_0,L_in,L_0), aux(A_0,A_n,L_0,L_n).   (P2)
aux(A_n,A_n,L_n,L_n).
aux(A_i,A_n,L_i,L_n) :- D'(A_i,A_i+1,L_i,L_i+1),
      aux(A_i+1,A_n,L_i+1,L_n).
```

Here, $\mathcal{D}'$ and t' are the same as in (P1), and a new predicate *aux*, called the *auxiliary nonterminal predicate* has been introduced.[14]

THEOREM 3.4. *Programs (P1) and (P2) are equivalent in predicate* a',[15]

The fact that (P2) is not left-recursive does not alone guarantee termination of top-down interpretation. However, if (P1) respects the guide-consumption condition and a further condition, the *no-chain condition*, then (P2) does indeed terminate.[16]

DEFINITION 3.5. *Program (P1) is said to respect the no-chain condition if each goal conjunction* C_i' *appearing in* $\mathcal{D}'$ *contains at least one call to* a' *or to* t'.

THEOREM 3.6. *Suppose (P1) satisfies both the guide-consumption condition and the no-chain condition. Then relative to top-down, depth-first, interpretation of (P2), the query* $a(A,L_0,L_n)$, with L_0 *completely instantiated, has a finite SLD search tree*[17] *associated with it* (in other words, all its solutions will be enumerated through backtracking, and the program will terminate).

4. Parsing and generation in Lexical Grammar

The rules of Fig. 3 are completely symmetrical in their specification of syntactic compositionality,

[12] Only programs of the (P0) form are discussed here, but the subsequent discussion of guides generalizes easily to arbitrary definite clause programs.

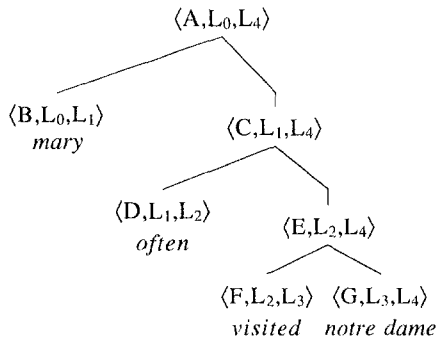
[13] The general problem of left-recursion elimination in DCGs (including *chain rules* and *null rules* [H78]) is studied in [D90a]; the existence of a *Generalized Greibach Normal Form* is proven, and certain decidability results are given.

[14] The (P1) $\leftrightarrow$ (P2) transformation is closely related to *left-corner parsing* [MTIIMY83], which can in fact be recovered from this transformation through a certain encoding procedure (see [D90b]).

[15] That is: $a'(A,L_{in},L_{out})$ is a consequence of (P1) iff $a'(A,L_{in},L_{out})$ is a consequence of (P2).

[16] In the context of CFGs, the no chain condition would correspond to a grammar without *chain rules*, and the guide consumption condition to a grammar without *null rules*.

[17] See [L87] for a definition of SLD search tree.



L_0 = [mary,often,visited,notre,dame]
L_1 = [often,visited,notre,dame]
L_2 = [visited,notre,dame]
L_3 = [notre,dame]
L_4 = []

Fig. 6. A guide for parsing

"string" compositionality and semantic compositionality[18]. The symmetry between string compositionality and semantic compositionality will allow us to treat parsing and generation as dual aspects of the same algorithm.

Orienting the rules. The *phrase* predicate can be rewritten in either one of the two forms: *phrase_p*, where emphasis is put on the relative linear order of constituents (*left* vs. *right*), and *phrase_g*, where emphasis is put on the relative semantic status (semantic *head* vs. semantic *dependent*) of constituents.

```
phrase_p(A) :- phrase_p(B), 𝒫(B,A).                    (P0p)
phrase_p(A) :- term(A)
```

where $\mathcal{P}(B,A)$ stands for:

```
𝒫(B,A) ≡ phrase_p(C),
    B.string_order = left,
    combine(B,C,A).
```

and

```
phrase_g(A) :- phrase_g(B), 𝒢(B,A).                    (P0g)
phrase_g(A) :- term(A)
```

where $\mathcal{G}(B,A)$ stands for:

```
𝒢(B,A) ≡ phrase_g(C),
    B.sem_order = head,
    combine(B,C,A).
```

LEMMA 4.1. phrase_p *and* phrase_g *are both equivalent to* phrase.

The *phrase_p* (resp. *phrase_g*) programs are now each in the format of the (P0) program of section 3, where *a* has been renamed: *phrase_p* (resp. *phrase_g*), and $\mathcal{D}$: $\mathcal{P}$ (resp. $\mathcal{G}$).

These programs can be extended into guided programs (P1p) and (P1g), as was done in section 3:

```
phrase_p'(A,Lin,Lout) :-                               (P1p)
    phrase_p'(B,Lin,Linter), 𝒫'(B,A,Linter,Lout).
phrase_p'(A,Lin,Lout) :- term_p'(A,Lin,Lout).
```

where:

```
𝒫'(B,A,Linter,Lout) ≡ phrase_p'(C,Linter,Lout),        (Dp)
    B.string_order = left,
    combine(B,C,A).
```

and

```
phrase_g'(A,Lin,Lout) :-                               (P1g)
    phrase_g'(B,Lin,Linter), 𝒢'(B,A,Linter,Lout).
phrase_g'(A,Lin,Lout) :- term_g'(A,Lin,Lout).
```

where:

```
𝒢'(B,A,Linter,Lout) ≡ phrase_g'(C,Linter,Lout),        (Dg)
    B.sem_order = head,
    combine(B,C,A).
```

In these programs, *term_p'* and *term_g'* are the refinements of *term* (corresponding to *t'* in program (P1) of section 3) used for parsing and generation respectively. Their definitions, which contain the substance of the guiding technique, are given below.

N.B. Programs (P1p) and (P1g) respect the *no-chain condition*: *phrase_p'* is called inside $\mathcal{P}$, and *phrase_g'* is called inside $\mathcal{G}$.

A conservative guide for parsing. Let us define *term_p'* in the following way:

```
term_p'(A,Lin,Lout) :- term(A),                        (Gp)
    append(A.string,Lout,Lin).
```

It is obvious that *term_p'* is a refinement of *term*. Using the definition of *combine_strings* in section 2, one can easily show that *program (P1p) is a conservative extension of program (P0p).*

The guide-structure **Gp** is the set of character strings, ordered in the following way: *st1* ≤ *st2* iff *st1* is a suffix of *st2*. If the lexicon is such that for any entry *term(A)*, *A.string* is instantiated and is different from the empty list, then it can easily be shown that *(P1p) respects the guide-consumption condition.*

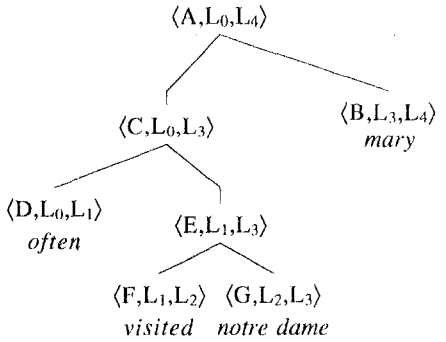
The guide just introduced for parsing is simply a restatement in terms of guides of the usual differential lists used in the Prolog translation of DCG rules.

A conservative guide for generation. Let us define *term_g'* in the following way (using the auxiliary predicate *extract_sems*):

```
term_g'(A,Lin,Lout) :- term(A),                        (Gg)
    Lin=[A.sem|Linter],
    extract_sems(A.subcat,SubcatSems),
    append(SubcatSems,Linter,Lout).
extract_sems([],[]).
extract_sems([X|Rest],[X.sem|RestSems]):-
    extract_sems(Rest,RestSems).
```

The guide structure *L* used for generation is a list of semantic structures, initially instantiated to *[S.sem]*, where *S* is the linguistic structure to be generated, of which the semantics *S.sem* is known. When a call *term_g'(A,Lin,Lout)* to the lexicon is made, with *Lin* instantiated to a list of semantic structures, the lexical structure *A* selected is constrained to be such that its semantics *A.sem* is the first item on the *Lin* list. The *A.sem* element is "popped" from the guide, and is replaced by the *list of the semantics of the phrases subcategorized by A*. (Fig. 7 illustrates the evolution of the guide in generation.)

[18] This symmetry should not be obscured by the fact that, in order to avoid duplicating clauses with the same logical content, the *presentation* of the rules appears otherwise (see above the discussion of "broken symmetry").



L_0 = [often(visit(mary,nd))]
L_1 = [visit(mary,nd)]
L_2 = [nd,mary]
L_3 = [mary]
L_4 = []

Fig. 7. A guide for generation

It is obvious that *term_g'* is then a refinement of *term*, and furthermore, using the definition of *combine_sems* in section 2, one can prove:

LEMMA 4.2. *Program (P1g) is a conservative extension of program (P0g).*

The guide-consumption condition in generation. Let us define recursively the *size* of an LG semantic representation as the function from terms to natural numbers such that:

$size[atom] = 1$
$size[atom(T_1,...,T_n)] = 1 + size[T_1] + ... + size[T_n]$

Assume now that, for any entry *term(A)*, the lexicon respects the following condition:

> *If* A.sem *is fully instantiated, then the* A.subcat *list is instantiated sufficiently so that, for any element* X *of this list, (i)* X.sem *is fully instantiated, and (ii)* X.sem *has a strictly smaller size than* A.sem.

Under these conditions, one can define a guide-structure **Gg** (see [D90b]), and one can prove:

LEMMA 4.3. *Program (P1g) satisfies the guide-consumption condition.*

The resulting programs for parsing and generation. After the left-recursion elimination transformation of section 3 is performed, the parsing and generation programs take the following forms:

```
phrase_p'(A_n,L_in,L_n) :- term_p'(A_0,L_in,L_0),
      aux_p(A_0,A_n,L_0,L_n).
aux_p(A_n,A_n,L_n,L_n).
aux_p(A_i,A_n,L_i,L_n) :- P'(A_i,A_i+1,L_i,L_i+1),
      aux_p(A_i+1,A_n,L_i+1,L_n).

phrase_g'(A_n,L_in,L_n) :- term_g'(A_0,L_in,L_0),
      aux_g(A_0,A_n,L_0,L_n).
aux_g(A_n,A_n,L_n,L_n).
aux_g(A_i,A_n,L_i,L_n) :- G'(A_i,A_i+1,L_i,L_i+1),
      aux_g(A_i+1,A_n,L_i+1,L_n).
```

That is, after expliciting *term_p'*, *term_g'*, $\mathcal{P}'$ and $\mathcal{G}'$ (see (Gp), (Gg), (Dp), (Dg), above), these programs take the forms (P2p) and (P2g) in Fig. 8; for

```
parse(S.string,S.sem) :-
      S.cat =v, S.subcat=[],        % S is a sentence
      phrase_p'(S,S.string,[]).

phrase_p'(A_n,L_in,L_n) :- term(A),                    (P2p)
      append(A.string,L_0,L_in),
      aux_p(A_0,A_n,L_0,L_n).
aux_p(A_n,A_n,L_n,L_n).
aux_p(A_i,A_n,L_i,L_n) :- phrase_p'(C,L_i,L_i+1),
      A_i.string_order = left,
      combine(A_i,C,A_i+1),
      aux_p(A_i+1,A_n,L_i+1,L_n).

generate(S.string,S.sem) :-
      S.cat =v, S.subcat=[],        % S is a sentence
      phrase_g'(S,[S.sem],[]).

phrase_g'(A_n,L_in,L_n) :- term(A),                    (P2g)
      L_in=[A.sem/L_inter],
      extract_sems(A.subcat,SubcatSems),
      append(SubcatSems,L_inter,L_0),
      aux_g(A_0,A_n,L_0,L_n).
aux_g(A_n,A_n,L_n,L_n).
aux_g(A_i,A_n,L_i,L_n) :- phrase_g'(C,L_i,L_i+1),
      A_i.sem_order = head,
      combine(A_i,C,A_i+1),
      aux_g(A_i+1,A_n,L_i+1,L_n).
extract_sems([],[]).
extract_sems([X/Rest],[X.sem/RestSems]):-
      extract_sems(Rest,RestSems).
```

Fig. 8. The final parsing and generation programs *parse* and *generate*

convenience interface predicates *parse* and *generate* are provided.

Under the conditions on the lexicon given above — which are satisfied by the lexicon of Fig. 4 —, programs (P1p) and (P1g) both respect the guide-consumption condition; they also respect the no-chain condition (see remark following the description of (P1p) and (P1g)); Theorem 3.6 applies, and we have the following result:

> *If* parse(A.string,A.sem) *(resp.* generate(A.string,A.sem)*) is called with* A.string *instantiated (resp.* A.sem *instantiated), then all solutions will be enumerated on backtracking, and the query will terminate.*

5. Further research

Handling extraposition with guides. The specific guides defined above for parsing and generation are not the only possible ones. If for some reason certain conditions on the lexicon are to be relaxed, then more sophisticated guides must and can be defined.

Thus, the guide introduced above for parsing essentially assumes that no lexical entry has an *empty string* realization. This condition may be too strict for certain purposes, such as handling *traces*. Interestingly, however, the guide consumption condition can still be imposed in these cases, if one takes care to suitably enrich the notion of guide.

Let us assume, for instance, that there be a general syntactic constraint to the effect that two empty lexical

items cannot immediately follow each other[19]. Let us then posit as a guide structure, instead of a list L of words, a *couple* $<L,B>$, where B is a variable restricted to taking values 0 or 1. Suppose further that these couples are ordered "lexicographically", ie that:

$$\forall\ L, L', B, B'$$
$$L < L' \Rightarrow <L,B> < <L',B'>$$
$$L = L' \wedge B < B' \Rightarrow <L,B> < <L,B'>\ .$$

It is easy to see that the set of guides is then a partially ordered set which respects the descending chain condition.

Let us finally assume that *term_p'* is redefined in the following manner:

```
term_p'(A,<Lin,Bin>,<Lout,Bout>) :-
    term(A),
    append(A.string,Lout,Lin),
    ( A.string = [], Bin =1, Bout = 0
    ; A.string ≠ [], Bin =_, Bout = 1 ).
```

It can be shown that this definition of *guide_parse* is sufficient to ensure the guide-consumption condition, and therefore guarantees the termination of the parsing process.

Variations on this idea are possible: for instance, one could define the guide as a couple $<L,X>$ where X is a list of left-extraposed constituents (see [P81]). Any time a constituent is added to the extraposition list X, this operation is required to consume some words from L, and any time a trace is encountered, it is required to "cancel" an element of X. Because the lexicographical order defined on such guides in the following way:

$$\forall\ L, L', X, X'$$
$$L < L' \Rightarrow <L,X> < <L',X'>$$
$$L = L' \wedge X < X' \Rightarrow <L,X> < <L,X'>\ .$$

respects the descending chain condition, the parsing process will be guaranteed to terminate.

6. Conclusion

This paper shows that parsing and generation can be seen as *symmetrical*, or dual, processes exploiting one and the same grammar and lexicon, and using a basic *left-recursion elimination* transformation. Emphasis is on the simplicity and symmetry of linguistic description, which is mostly contained in the lexicon; compositionality appears under three aspects: string compositionality, semantic compositionality, and syntactic compositionality. The analysis and generation processes each favor one aspect: string compositionality in analysis, semantic compositionality in generation. These give rise to two *guides* (analysis guide and generation guide), which are generalizations of string indexes. The left-recursion elimination transformation described in the paper is stated using the general notion of guide, and is provably guaranteed, under certain explicit conditions, to lead to termination of the parsing and generation processes. We claim that the approach provides a simple, yet powerful solution to the problem of grammatical bidirectionality, and are currently testing it as a possible replacement for a more rule-oriented grammatical component in the context of the CRITTER translation system [IDM88].

[19] A counter-example to this simplistic assumption is not hard to come by: *the person who$_1$ john persuaded e$_1$ PRO to drink*. However, the assumption gives the flavor of a possible set of strategies for handling empty categories.

Acknowledgments

Thanks to Michel Boyer, Jean-Luc Cochard and Elliott Macklovitch for discussion and comments.

References

[D90a] Dymetman, Marc. *A Generalized Greibach Normal Form for Definite Clause Grammars*. Laval, Québec: Ministère des Communications Canada, Centre Canadien de Recherche sur l'Informatisation du Travail.

[D90b] Dymetman, Marc. *Left-Recursion Elimination, Guiding, and Bidirectionality in Lexical Grammars* (to appear).

[DI88] Dymetman, Marc and Pierre Isabelle. 1988. Reversible Logic Grammars for Machine Translation. In *Proceedings of the Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*. Pittsburgh: Carnegie Mellon University, June.

[DI90] Dymetman, Marc and Pierre Isabelle. 1990. Grammar Bidirectionality through Controlled Backward Deduction. In *Logic and Logic Grammars for Language Processing*, eds. Saint Dizier, P. and S. Szpakowicz. Chichester, England: Ellis Horwood.

[GKPS87] Gazdar, Gerald, Ewan Klein, Geoffrey Pullum and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

[H78] Harrison, Michael A. 1978. *Introduction to Formal Language Theory*. Reading, MA: Addison-Wesley.

[IDM88] Isabelle, Pierre, Marc Dymetman and Elliott Macklovitch. 1988. CRITTER: a Translation System for Agricultural Market Reports. In *Proceedings of the 12th International Conference on Computational Linguistics*, 261-266. Budapest, August.

[L87] Lloyd, John Wylie. 1987. *Foundations of Logic Programming*, 2nd ed. Berlin: Springer-Verlag.

[MTHMY83] Matsumoto Y., H. Tanaka, H. Hirikawa, H. Miyoshi, H. Yasukawa, 1983. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing* 1:2, 145-158.

[PW80] Pereira, Fernando C. N. and David H. D. Warren. 1980. Definite Clause Grammars for Language Analysis. *Artificial Intelligence*:13, 231-78.

[P81] Pereira, Fernando C. N. 1981. Extraposition Grammars. *Computational Linguistics* 7:4, 243-56.

[S88] Shieber, Stuart M. 1988. A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, 614-19. Budapest, August.

[SNMP89] Shieber, Stuart, M., Gertjan van Noord, Robert Moore and Fernando Pereira. 1989. A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, 7-17. Vancouver, BC, Canada, June.

[N89] Van Noord, Jan. 1989. *BUG: A Directed Bottom-up Generator for Unification Based Formalisms*. Working Papers in Natural Language Processing No. 4. Utrecht, Holland: RUU, Department of Linguistics.

[ZKC87] Zeevat, H., E. Klein, and J. Calder. 1987. *Unification Categorial grammar*. Edinburgh: University of Edinburgh, Centre for Cognitive Science, Research Paper EUCCS/RP-21.