A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms

Stuart M. Shieber,* Gertjan van Noord,† Robert C. Moore,* and Fernando C. N. Pereira*

*Artificial Intelligence Center
SRI International
Menlo Park, CA 94025, USA

†Department of Linguistics
Rijksuniversiteit Utrecht
Utrecht, Netherlands

Abstract

We present an algorithm for generating strings from logical form encodings that improves upon previous algorithms in that it places fewer restrictions on the class of grammars to which it is applicable. In particular, unlike an Earley-deduction generator (Shieber, 1988), semantically nonmonotonic grammars can be used, yet unlike top-down methods, left-recursion is also permissible. The enabling design feature of the algorithm is its implicit traversal of the analysis tree for the string being generated in a semantic-head-driven fashion.

1 Introduction

The problem of generating a well-formed natural-language expression from an encoding of its meaning possesses certain properties which distinguish it from the converse problem of recovering a meaning encoding from a given natural-language expression. In previous work (Shieber, 1988), however, one of us attempted to characterize these differing properties in such a way that a single uniform architecture, appropriately parameterized, might be used for both natural-language processes. In particular, we developed an architecture inspired by the Earley deduction work of Pereira and Warren (1983), but which generalized that work allowing for its use in both a parsing and generation mode, merely by setting the values of a small number of parameters.

As a method for generating natural-language expressions, the Earley deduction method is reasonably successful along certain dimensions. It is quite simple, general in its applicability to a range of unification-based and logic grammar formalisms, and uniform, in that it places only one restriction (discussed below) on the form of the linguistic analyses allowed by the grammars used in generation. In particular, generation from grammars with recursions whose well-foundedness relies on lexical information will terminate; top-down generation regimes such as those of Wedekind (1988) or Dymetman and Isabelle (1988) lack this property, discussed further in Section 3.1.

Unfortunately, the bottom-up, left-to-right processing regime of Earley generation—as it might be called—has its own inherent frailties. Efficiency considerations require that only grammars possessing a property of semantic monotonicity can be effectively used, and even for those grammars, processing can become overly nondeterministic.

The algorithm described in this paper is an attempt to resolve these problems in a satisfactory manner. Although we believe that this algorithm could be seen as an instance of a uniform architecture for parsing and generation—just as the extended Earley parser (Shieber, 1985b) and the bottom-up generator were instances of the generalized Earley deduction architecture—our efforts to date have been aimed foremost towards the development of the algorithm for generation alone, so that we will have little to say about its relation to parsing, leaving such questions for later research.[1]

2 Applicability of the Algorithm

As was the case for the Earley-based generator, the new algorithm assumes that the grammar is a unification-based or logic grammar with a phrase-structure backbone and complex nonterminals. Furthermore, and again consistent with previous work, we assume that the nonterminals associate to the phrases they describe logical expressions encoding their possible meanings. We will describe the algorithm in terms of an implemen-

[1] Martin Kay (personal communication) has developed a parsing algorithm that seems to be the parsing correlate to the generation algorithm presented here. Its existence might point the way towards a uniform architecture.

tation of it for definite-clause grammars (DCG) although we believe the underlying method to be more broadly applicable.

A variant of our method is used in Van Noord's BUG (Bottom-Up Generator) system, part of MiMo2, an experimental machine translation system for translating international news items of Teletext, which uses a Prolog version of PATR-II similar to that of Hirsh (1987). According to Martin Kay (personal communication) the STREP machine translation project at the Center for the Study of Language and Information uses a version of our algorithm algorithm to generate with respect to grammars based on head-driven phrase-structure grammar (HPSG). Finally, Calder, Reape and Zeevat (1989) report on a generation algorithm for unification categorial grammar that appears to be a special case of ours.

3 Problems with Existing Generators

Existing generation algorithms have efficiency or termination problems in their use for generating with respect to certain classes of grammars. We review the problems of both top-down and bottom-up regimes in this section.

3.1 Problems with Top-Down Generators

Consider a naïve top-down generation mechanism that takes as input the semantics to generate from and a corresponding syntactic category and builds a complete tree top-down, left-to-right by applying rules of the grammar nondeterministically to the fringe of the expanding tree. This control regime is realized, for instance, when running a DCG "backwards" as a generator.

Clearly, such a generator may not terminate. For example, consider a grammar that includes the rule

```
s/S --> np/NP, vp(NP)/S.
```

(The intention is that VPs like, say, "loves Mary" be associated with a nonterminal `vp(X)/love(X, mary)`.) Once this rule is applied to the goal `s/love(john, mary)`, the subgoal `np/NP` will be considered. But the generation search space for that goal is infinite and so has infinite branches, because all noun phrases, and thus arbitrarily large ones, match the goal.This is an instance of the general problem known from logic programming that a logic program may not terminate when called with a goal less instantiated than what was intended by the program's designer. Dymetman and Isabelle (1988), noting this problem, propose allowing the grammar-writer to specify a separate goal ordering for parsing and for generation. For the case at hand, the solution is to generate the VP first—from the goal `vp(NP)/loves(john, mary)`—in the course of which the variable `NP` will become bound so that the generation from `np/NP` will terminate. Wedekind (1988) achieves this goal by expanding first nodes that are *connected*, that is, whose semantics is instantiated. Since the NP is not connected in this sense, but the VP is, the latter will be expanded first. In essence, the technique is a kind of goal freezing (Colmerauer, 1982) or implicit *wait* declaration (Naish, 1986). For cases in which the a priori ordering of goals is insufficient, Dymetman and Isabelle also introduce goal freezing to control expansion.

Although vastly superior to the naïve top-down algorithm, even this sort of amended top-down approach to generation based on goal freezing under one guise or another fails to terminate with certain linguistically plausible analyses. For example, the "complements" rule given by Shieber (1985a, pages 77-78) in the PATR-II formalism

$$
\begin{array}{l}
\mathrm{VP}_1 \rightarrow \mathrm{VP}_2\ \mathrm{X} \\
\qquad \langle \mathrm{VP}_1 \text{ head} \rangle = \langle \mathrm{VP}_2 \text{ head} \rangle \\
\qquad \langle \mathrm{VP}_2 \text{ syncat first} \rangle = \langle \mathrm{X} \rangle \\
\qquad \langle \mathrm{VP}_2 \text{ syncat rest} \rangle = \langle \mathrm{VP}_1 \text{ syncat} \rangle
\end{array}
$$

can be encoded as the DCG-style rule:

```
vp(Head, Syncat) -->
   vp(Head, [Compl|Syncat]), Compl.
```

Top-down generation using this rule will be forced to expand the lower VP before its complement, since `Compl` is uninstantiated initially. But application of the rule can recur indefinitely leading to nontermination.

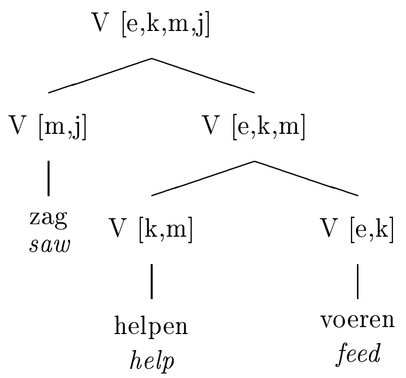
The problem arises because there is no limit to the size of the subcategorization list. Although one might propose an ad hoc upper bound for lexical entries, even this expedient may be insufficient. In analyses of Dutch cross-serial verb constructions (Evers, 1975; Huybrechts, 1984) subcategorization lists such as these may be appended by syntactic rules (Moortgat, 1984; Steedman, 1985; Pollard, 1988) resulting in indefinitely long lists. Consider the Dutch sentence

dat [Jan [Marie [de oppasser [de olifanten
that John Mary the keeper the elephants

[zag helpen voeren]]]]
saw help feed

that John saw Mary help the keeper feed the elephants

The string of verbs is analysed by appending their subcategorization lists as follows:



Subcategorization lists under this analysis can have any length, and it is impossible to predict from a semantic structure the size of its corresponding subcategorization list merely by examining the lexicon.

In summary, top-down generation algorithms, even if controlled by the instantiation status of goals, can fail to terminate on certain grammars. The case given above, reminiscent of analyses from HPSG, as well as analyses from categorial unification grammar are examples in which the well-foundedness of the generation process resides in lexical information unavailable to top-down regimes.

3.2 Problems with Bottom-up Generators

The bottom-up Earley-deduction generator does not fall prey to these problems of nontermination in the face of recursion because lexical information is available immediately. However, several important frailties of the Earley generation method were noted, even in the earlier work.

For efficiency, generation using this Earley deduction method requires an incomplete search strategy, filtering the search space using semantic information. The semantic filter makes generation from a logical form computationally feasible, but preserves completeness of the generation process only in the case of semantically monotonic grammars, those grammars in which the semantic component of each right-hand-side nonterminal subsumes some portion of the semantic component of the left-hand-side. The semantic monotonicity constraint itself is quite restrictive. Although it is intuitively plausible that the semantic content of subconstituents ought to play a role in the semantics of their combination—this is just a kind of compositionality claim—there are certain cases in which reasonable linguistic analyses might violate this intuition. In general, these cases arise when a particular lexical item is stipulated to occur, the stipulation being either lexical (as in the case of particles or idioms) or grammatical (as in the case of expletive expressions).

Second, the left-to-right scheduling of Earley parsing, geared as it is toward the structure of the string rather than that of its meaning, is inherently more appropriate for parsing than generation.[2] This manifests itself in an overly high degree of nondeterminism in the generation process. For instance, various nondeterministic possibilities for generating a noun phrase (using different cases, say) might be entertained merely because the NP occurs before the verb which would more fully specify, and therefore limit, the options. This nondeterminism has been observed in practice.

3.3 Source of the Problems

We can think of a parsing or generation process as *discovering* an analysis tree,[3] one admitted by the grammar and satisfying certain syntactic or semantic conditions, by traversing a virtual tree and constructing the actual tree during the traversal. The conditions to be satisfied—possessing a given yield in the parsing case, or having a root node labeled with given semantic information in the case of generation—reflect the different premises of the two types of problem.

¿From this point of view, a naïve top-down parser or generator performs a depth-first, left-to-right traversal of the tree. Completion steps in Earley's algorithm, whether used for parsing

[2] Pereira and Warren (1983) point out that Earley deduction is not restricted to a left-to-right expansion of goals, but this suggestion was not followed up with a specific algorithm addressing the problems discussed here.

[3] We use the term "analysis tree" rather than the more familiar "parse tree" to make clear that the source of the tree is not necessarily a parsing process; rather the tree serves only to codify a particular analysis of the structure of the string.

or generation, correspond to a post-order traversal (with prediction acting as a pre-order filter). The left-to-right traversal order of both of these methods is geared towards the given information in a parsing problem, the string, rather than that of a generation problem, the goal logical form. It is exactly this mismatch between structure of the traversal and structure of the problem premise that accounts for the profligacy of these approaches when used for generation.

Thus for generation, we want a traversal order geared to the premise of the generation problem, that is, to the semantic structure of the sentence. The new algorithm is designed to reflect such a traversal strategy respecting the semantic structure of the string being generated, rather than the string itself.

4 The New Algorithm

Given an analysis tree for a sentence, we define the pivot node as the lowest node in the tree such that it and all higher nodes up to the root have the same semantics. Intuitively speaking, the pivot serves as the *semantic head* of the root node. Our traversal will proceed both top-down and bottom-up from the pivot, a sort of semantic-head-driven traversal of the tree. The choice of this traversal allows a great reduction in the search for rules used to build the analysis tree.

To be able to identify possible pivots, we distinguish a subset of the rules of the grammar, the *chain rules*, in which the semantics of some right-hand-side element is identical to the semantics of the left-hand side. The right-hand-side element will be called the rule's semantic head.[4] The traversal, then, will work top-down from the pivot using a non-chain rule, for if a chain rule were used, the pivot would not be the lowest node sharing semantics with the root. Instead, the pivot's semantic head would be. After the non-chain rule is chosen, each of its children must be generated recursively.

The bottom-up steps to connect the pivot to the root of the analysis tree can be restricted to chain rules only, as the pivot (along with all intermediate nodes) has the same semantics as the root and must therefore be the semantic head. Again, after a chain rule is chosen to move up one node in the tree being constructed, the remaining (non-semantic-head) children must be generated recursively.

The top-down base case occurs when the non-chain-rule has no nonterminal children, i.e., it introduces lexical material only. The bottom-up base case occurs when the pivot and root are trivially connected because they are one and the same node.

4.1 A DCG Implementation

To make the description more explicit, we will develop a Prolog implementation of the algorithm for DCGs, along the way introducing some niceties of the algorithm previously glossed over.

In the implementation, a term of the form `node(Cat, P0, P)` represents a phrase with the syntactic and semantic information given by `Cat` starting at position `P0` and ending at position `P` in

[4] In case there are two right-hand-side elements that are semantically identical to the left-hand side, there is some freedom in choosing the semantic head, although the choice is not without ramifications. For instance, in some analyses of NP structure, a rule such as

```
np/NP --> det/NP, nbar/NP.
```

is postulated. In general, a chain rule is used bottom-up from its semantic head and top-down on the non-semantic-head siblings. Thus, if a non-semantic-head subconstituent has the same semantics as the left-hand-side, a recursive top-down generation with the same semantics will be invoked. In theory, this can lead to nontermination, unless syntactic factors eliminate the recursion, as they would in the rule above regardless of which element is chosen as semantic head. In a rule for relative clause introduction such as the following (in highly abbreviated form)

```
nbar/N --> nbar/N, sbar/N.
```

we can (and must) choose the nominal as semantic head to effect termination. However, there are other problematic cases, such as verb-movement analyses of verb-second languages, whose detailed discussion is beyond the scope of this paper.

the string being generated. As usual for DCGs, a string position is represented by the list of string elements after the position. The generation process starts with a goal category and attempts to generate an appropriate node, in the process instantiating the generated string.

```
gen(Cat, String) :-
   generate(node(Cat,String,[])).
```

To generate from a node, we nondeterministically choose a non-chain rule whose left-hand side will serve as the pivot. For each right-hand-side element, we recursively generate, and then connect the pivot to the root.

```
generate(Root) :-
    % choose non-chain rule
    applicable_non_chain_rule(Root,
       Pivot, RHS),
    % generate all subconstituents
    generate_rhs(RHS),
    % generate material on path to root
    connect(Pivot, Root).
```

The processing within generate_rhs is a simple iteration.

```
generate_rhs([]).

generate_rhs([First | Rest]) :-
   generate(First),
   generate_rhs(Rest).
```

The connection of a pivot to the root, as noted before, requires choice of a chain rule whose semantic head matches the pivot, and the recursive generation of the remaining right-hand-side. We assume a predicate `applicable_chain_rule(SemHead, LHS, Root, RHS)` that holds if there is a chain rule admitting a node `LHS` as the left-hand-side, `SemHead` as its semantic head, and `RHS` as the remaining right-hand-side nodes, such that the left-hand-side node and the root node `Root` can be themselves connected.

```
connect(Pivot, Root) :-
    % choose chain rule
    applicable_chain_rule(Pivot, LHS,
       Root, RHS),
    % generate remaining siblings
    generate_rhs(RHS),
    % connect the new parent to the root
    connect(LHS, Root).
```

The base case occurs when the root and the pivot are the same. One must be careful that all identity checks like this one be implemented correctly in the generator by using a sound unification algorithm with the occurs check. (The default unification in most Prolog systems is unsound in this respect.) For example, a grammar with a gap-threading treatment of wh-movement (Pereira, 1981; Pereira and Shieber, 1985) might include the rule

```
np(Agr, [np(Agr)/Sem|X]-X)/Sem ---> [].
```

stating that an NP with agreement `Agr` and semantics `Sem` can be empty provided that the list of gaps in the NP can be represented as the difference list `[np(Agr)/Sem|X]-X`, that is the list containing an NP gap with the same agreement features `Agr` (Pereira and Shieber, 1985, page 128). The above rule is a non-chain rule and so it will be considered when trying to generate any nongap NP, such as the proper noun `np(3-sing,G-G)/john`. The base case of `connect` will try to unify that term with the head of the rule above, leading to the attempted unification of `X` with `[np(Agr)/Sem|X]`, an occurs-check failure. The base case, incorporating the explicit call to a sound unification algorithm is thus as follows:

```
connect(Pivot, Root) :-
    % trivially connect pivot to root
    unify(Pivot, Root).
```

Now, we need only define the notion of an applicable chain or non-chain rule. A non-chain rule is applicable if the semantics of the left-hand-side of the rule (which is to become the pivot) matches that of the root. Further, we require a top-down check that syntactically the pivot can serve as the semantic head of the root. For this latter purpose, we assume a predicate `chained_nodes` that codifies the transitive closure of the semantic head relation over categories. This is the correlate of the link relation used in left-corner parsers with top-down filtering; we direct the reader to the discussion by Matsumoto et al. (1983) or Pereira and Shieber (1985, page 182) for further information.

```
applicable_non_chain_rule(Root, Pivot,
      RHS) :-
    % semantics of root and pivot are same
    node_semantics(Root, Sem),
    node_semantics(Pivot, Sem),
    % choose a non-chain rule
    non_chain_rule(LHS, RHS),
    % ...whose lhs matches the pivot
    unify(Pivot, LHS),
    % make sure the categories can connect
    chained_nodes(Pivot, Root).
```

A chain rule is applicable to connect a pivot to a root if the pivot can serve as the semantic head of the rule and the left-hand-side of the rule is appropriate for linking to the root.

```
applicable_chain_rule(Pivot, Parent,
      Root, RHS) :-
   % choose a chain rule
   chain_rule(Parent, RHS, SemHead),
   % ... whose sem. head matches pivot
   unify(Pivot, SemHead),
   % make sure the categories can connect
   chained_nodes(Parent, Root).
```

The information needed to guide the generation (given as the predicates `chain_rule`, `non_chain_rule`, and `chained_nodes`) can be computed automatically from the grammar; a program to compile a DCG into these tables has in fact been implemented. The details of the process will not be discussed further. The careful reader will have noticed, however, that no attention has been given to the issue of terminal symbols on the right-hand sides of rules. This is because during the compilation process, the right-hand side of a rule is converted from a list of categories and terminal strings to a list of nodes connected together by the difference-list threading technique used for standard DCG compilation. At that point, terminal strings can be introduced into the string threading and need never be considered further.

4.2 An Example

We turn now to a simple example to provide a flavor for the order of processing pursued by this generation algorithm. The grammar fragment in Figure 1 uses an infix operator `/` to separate syntactic and semantic category information. Subcategorization for complements is performed lexically.

Consider the generation from the category `sentence/decl(call_up(john,friends))`. The analysis tree that we will be implicitly traversing in the course of generation is given in Figure 2. The rule numbers are keyed to the grammar. The pivots chosen during generation and the branches corresponding to the semantic head relation are shown in bold face.

We begin by attempting to find a non-chain rule that will define the pivot. This is a rule whose left-hand-side semantics matches the root semantics `decl(call_up(john,friends))` (although its syntax may differ). In fact, the only such non-chain rule is

```
sentence/decl(S) ---> s(finite)/S.        (1)
```

We conjecture that the pivot is labeled `sentence/decl(call_up(john,friends))`. In terms of the tree traversal, we are implicitly choosing the root node [a] as the pivot. We recursively generate from the child's node [b], whose category is `s(finite)/call_up(john,friends)`. For this category, the pivot (which will turn out to be node [f]) will be defined by the non-chain rule

```
vp(finite,[np(_)/O,                       (4)
           p/up,
           np(3-sing)/S])
   /call_up(S,O) ---> [calls].
```

(If there were other forms of the verb, these would be potential candidates, but would be eliminated by the `chained_nodes` check, as the semantic head relation requires identity of the verb form of a sentence and its VP head.) Again, we recursively generate for all the nonterminal elements of the right-hand side of this rule, of which there are none.

We must therefore connect the pivot [f] to the root [b]. A chain rule whose semantic head matches the pivot must be chosen. The only choice is the rule

```
vp(Form,Subcat)/S --->                    (3)
   vp(Form,[Compl|Subcat])/S, Compl.
```

Unifying in the pivot, we find that we must recursively generate the remaining RHS element `np(_)/friends`, and then connect the left-hand side node [e] with category

```
vp(finite,[lex/up,
           np(3-sing)/john])
   /call_up(john,friends)
```

to the same root [b]. The recursive generation yields a node covering the string "friends" following the previously generated string "calls". The recursive connection will use the same chain rule, generating the particle "up", and the new node to be connected [d]. This node requires the chain rule

```
s(Form)/S --->                            (2)
   Subj, vp(Form,[Subj])/S.
```

for connection. Again, the recursive generation for the subject yields the string "John", and the new node to be connected `s(finite)/call_up(john, friends)`. This last node connects to the root [b] by virtue of identity.

This completes the process of generating top-down from the original pivot `sentence/`

```
sentence/decl(S) ---> s(finite)/S.                                    (1)
sentence/imp(S) ---> vp(nonfinite,[np(_)/you])/S.

s(Form)/S ---> Subj, vp(Form,[Subj])/S.                               (2)
vp(Form,Subcat)/S ---> vp(Form,[Compl|Subcat])/S, Compl.              (3)
vp(Form,[Subj])/S ---> vp(Form,[Subj])/VP, adv(VP)/S.
...
vp(finite,[np(_)/O,np(3-sing)/S])/love(S,O) ---> [loves].
...
vp(finite,[np(_)/O,p/up,np(3-sing)/S])/call_up(S,O) ---> [calls].     (4)
...
vp(finite,[np(3-sing)/S])/leave(S) ---> [leaves].
...
np(3-sing)/john ---> [john].                                          (5)
np(3-pl)/friends ---> [friends].                                      (6)
...
adv(VP)/often(VP) ---> [often].
...
det(3-sing,X,P)/qterm(every,X,P) ---> [every].
...
n(3-sing,X)/friend(X) ---> [friend].
n(3-pl,X)/friend(X) ---> [friends].
...
p/up ---> [up].                                                       (7)
p/on ---> [on].
...
```

Figure 1: Grammar Fragment

`decl(call_up(john,friends))`. All that remains is to connect this pivot to the original root. Again, the process is trivial, by virtue of the base case for connection. The generation process is thus completed, yielding the string "John calls friends up". The drawing summarizes the generation process by showing which steps were performed top-down or bottom-up by arrows on the analysis tree branches.

The grammar presented here was perforce trivial, for expository reasons. We have developed more extensive experimental grammars that can generate relative clauses with gaps and sentences with quantified NPs from quantified logical forms by using a version of Cooper storage (Cooper, 1983). We give an outline of our treatment of quantification in Section 6.2.

5 Important Properties of the Algorithm

Several properties of the algorithm are exhibited by the preceding example example.

First, the order of processing is not left-to-right. The verb was generated before any of its complements. Because of this, the semantic information about the particle "up" was available, even though this information appears nowhere in the goal semantics. That is, the generator operated appropriately despite a semantically nonmonotonic grammar.

In addition, full information about the subject, including agreement information was available before it was generated. Thus the nondeterminism that is an artifact of left-to-right processing, and a source of inefficiency in the Earley generator, is eliminated. Indeed, the example here was completely deterministic; all rule choices were forced.

Finally, even though much of the processing is top-down, left-recursive rules (e.g., rule (3)) are still handled in a constrained manner by the algorithm.

For these reasons, we feel that the semantic-head-driven algorithm is a significant improvement over top-down methods and the previous bottom-up method based on Earley deduction.

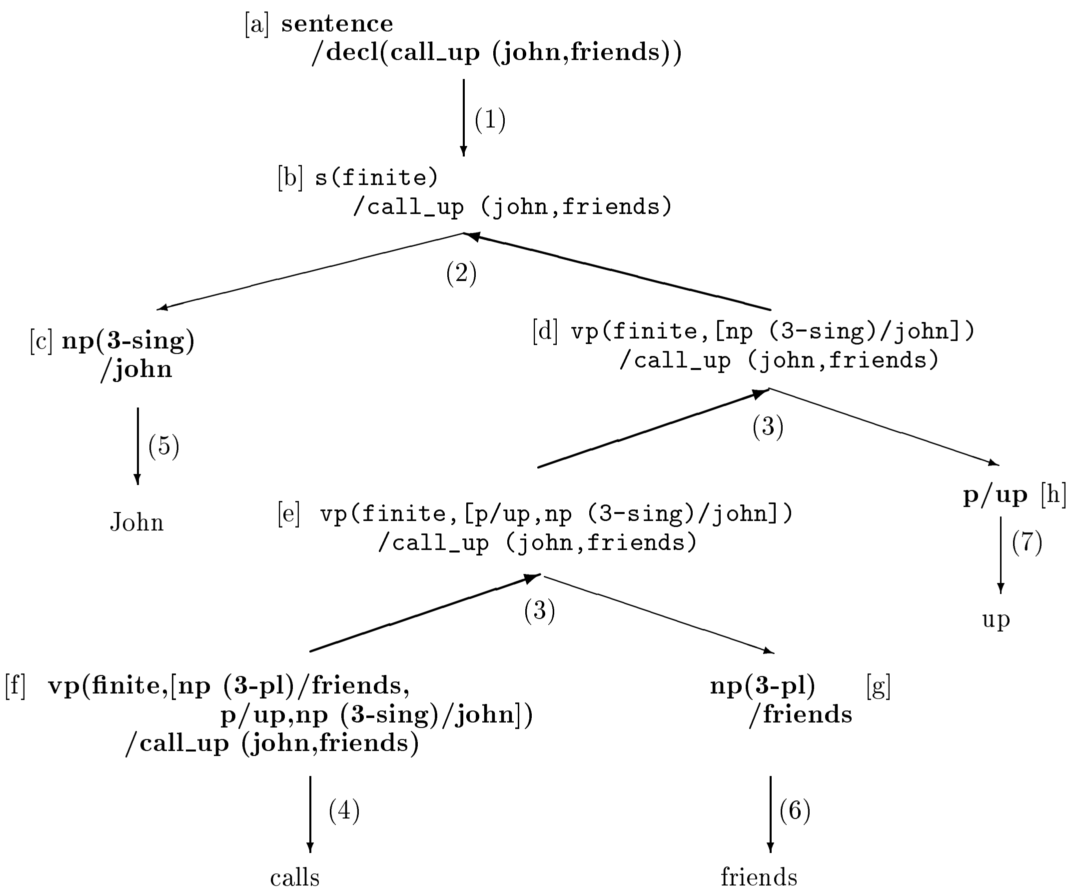


Figure 2: Analysis Tree Traversal Example

6 Extensions

We will now outline how the algorithm and the grammar it uses can be extended to encompass some important analyses and constraints.

6.1 Completeness and Coherence

Wedekind (1988) defines completeness and coherence of a generation algorithm as follows. Suppose a generator derives a string w from a logical form s, and the grammar assigns to w the logical form a. The generator is complete if s always subsumes a and coherent if a always subsumes s. The generator defined in Section 4.1 is not coherent or complete in this sense; it requires only that a and s be compatible, that is, unifiable.

If the logical-form language and semantic interpretation system provide a sound treatment of variable binding and scope, abstraction and application, completeness and coherence will be irrelevant because the logical form of any phrase will not contain free variables. However, neither semantic projections in LFG (Halvorsen and Kaplan, 1988) nor definite-clause grammars provide the means for such a sound treatment: logical-form variables or missing arguments of predicates are both encoded as unbound variables (attributes with unspecified values in the LFG semantic projection) at the description level. Then completeness and coherence become important. For example, suppose a grammar associated the following strings and logical forms.

`eat(john, X)`
'John ate'

`eat(john, banana)`
'John ate a banana'

`eat(john, nice(yellow(banana)))`
'John ate a nice yellow banana'

The generator of Section 4.1 would generate any of these sentences for the logical form `eat(john, X)` (because of its incoherence) and would generate 'John ate' for the logical form `eat(john, banana)` (because of its incompleteness).

Coherence can be achieved by removing the confusion between object-level and metalevel variables mentioned above, that is, by treating logical-form variables as constants at the description level. In practice, this can be achieved by replacing each variable in the semantics from which we are generating by a new distinct constant (for instance with the `numbervars` predicate built into some implementations of Prolog). These new constants will not unify with any augmentations to the semantics. A suitable modification of our generator would be

```
gen(Cat, String) :-
   cat_semantics(Cat,Sem),
   numbervars(Sem,0,_),
   generate(node(Cat,String,[])).
```

This leaves us with the completeness problem. This problem arises when there are phrases whose semantics are not ground at the description level, but instead subsume the goal logical form or generation. For instance, in our hypothetical example, the string 'John eats' will be generated for semantics `eat(john, banana)`. The solution is to test at the end of the generation procedure whether the feature structure that is found is complete with respect to the original feature structure. However, because of the way in which top-down information is used, it is unclear what semantic information is derived by the rules themselves, and what semantic information is available because of unifications with the original semantics. For this reason so called "shadow" variables are added to the generator that represent the feature structure derived by the grammar itself. Furthermore a copy of the semantics of the original feature structure is made at the start of the generation process. Completeness is achieved by testing whether the semantics of the shadow is subsumed by the copy.

6.2 Quantifier Storage

We will outline here how to generate from a quantified logical form sentences with quantified NPs one of whose readings is the original logical form, thus performing quantifier lowering automatically. For this, we will associate a quantifier store to certain categories and add to the grammar suitable store manipulation rules.

Each category whose constituents may create store elements will have a store feature. Furthermore, for each such category whose semantics can be the scope of a quantifier, there will be an optional non-chain rule to take the top element of an ordered store and apply it to the semantics of the category. For example, here is the rule for sentences:

```
s(Form, GO-G, Store)/quant(Q,X,R,S) --->
   s(Form, GO-G, [qterm(Q,X,R)|Store])/S.
```

The term `quant(Q,X,R,S)` represents a quantified formula with quantifier `Q`, bound variable `X`, restriction `R` and scope `S`, and `qterm(Q,X,R)` is the corresponding store element.

In addition, some mechanism is needed to combine the stores of the immediate constituents of a phrase into a store for the phrase. For example, the combination of subject and complement stores for a verb into a clause store is done in one of our test grammars by lexical rules such as

```
vp(finite, [np(_, SO)/O,
            np(3-sing, SS)/S], SC)
  /love(S,O) --->
     [loves], {shuffle(SS, SO, SC)}.
```

which states that the store `SC` of a clause with main verb 'love' and the stores `SS` and `SO` of the subject and object the verb subcategorizes for satisfy the constraint `shuffle(SS, SO, SC)`, meaning that `SC` is an interleaving of elements of `SS` and `SO` in their original order. [5]

Finally, it is necessary to deal with the noun phrases that create store elements. Ignoring the issue of how to treat quantifiers from within complex noun phrases, we just need lexical rules for determiners, of the form

```
det(3-sing,X,P,[qterm(every,X,P)])/X --->
   [every].
```

stating that the semantics of a quantified NP is simply the variable bound by the store element arising from the NP. For rules of this form to work properly, it is essential that distinct bound logical-form variables be represented as distinct constants in the terms encoding the logical forms. This is an instance of the problem of coherence discussed in the previous section.

The rules outlined here are less efficient than necessary because the distribution of store elements among the subject and complements of a

[5] Further details of the use of `shuffle` in scoping are given by Pereira and Shieber (1985).

verb does not check whether the variable bound by a store element actually appears in the semantics of the phrase to which it is being assigned, leading to many dead ends in the generation process. Also, the rules are sound for generation but not for analysis, because they do not enforce the constraint that every occurrence of a variable in logical form be outscoped by the variable's binder. Adding appropriate side conditions to the rules, following the constraints discussed by Hobbs and Shieber (Hobbs and Shieber, 1987) would not be difficult.

6.3 Postponing Lexical Choice

As it stands, the generation algorithm chooses particular lexical forms on-line. This approach can lead to a certain amount of unnecessary nondeterminism. For instance, the choice of verb form might depend on syntactic features of the verb's subject available only after the subject has been generated. This nondeterminism can be eliminated by deferring lexical choice to a postprocess. The generator will yield a list of lexical items instead of a list of words. To this list a small phonological frontend is applied. BUG uses such a mechanism to eliminate much of the uninteresting nondeterminism in choice of word forms. Of course, the same mechanism could be added to any of the other generation techniques discussed to in this paper.

7 Further Research

Further enhancements to the algorithm are envisioned. First, any system making use of a tabular link predicate over complex nonterminals (like the `chained_nodes` predicate used by the generation algorithm, and including the link predicate used in the BUP parser (Matsumoto *et al.*, 1983)) is subject to a problem of spurious redundancy in processing if the elements in the link table are not mutually exclusive. For instance, a single chain rule might be considered applicable twice because of the nondeterminism of the call to `chained_nodes`. This general problem has to date received little attention and no satisfactory solution in the logic grammar literature.

More generally, the backtracking regimen of our implementation of the algorithm may lead to recomputation of results. Again, this is a general property of backtrack methods, and is not particular to our application. The use of dynamic programming techniques, a la chart parsing, would be an appropriate augmentation to the implementation of the algorithm. Happily, such an augmentation would serve to eliminate the redundancy caused by the linking relation as well.

Finally, in order to incorporate a general facility for auxiliary conditions in rules, some sort of delayed evaluation triggered by appropriate instantiation (e.g., wait declarations (Naish, 1986)) would be desirable. None of these changes, however, constitutes restructuring of the algorithm; rather they modify its realization in significant and important ways.

Acknowledgements

Shieber, Moore and Pereira were supported in this work by a contract with the Nippon Telephone Telegraph Corp. and by a gift from the Systems Development Foundation as part of a coordinated research effort with the Center for the Study of Language and Information, Stanford University; van Noord was supported by the European Community and the Nederlands Bureau voor Bibliotheekwezen en Informatieverzorgin through the Eurotra project. We would like to thank Mary Dalrymple and Louis des Tombe for their helpful discussions regarding this work.

References

Jonathan Calder, Mike Reape, and Henk Zeevat. 1989. An algorithm for generation in unification categorial grammar. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240, Manchester, England, 10-12 April. University of Manchester Institute of Science and Technology.

Alain Colmerauer. 1982. PROLOG II: Manuel de réference et modèle théorique. Technical report, Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, France.

Robin Cooper. 1983. *Quantification and Syntactic Theory*, volume 21 of *Synthese Language Library.* D. Reidel, Dordrecht, Netherlands.

Marc Dymetman 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, Pennsylvania. Carnegie-Mellon University.

Arnold Evers. 1975. *The transformational cycle in German and Dutch.* Ph.D. thesis, University of Utrecht, Utrecht, Netherlands.

Per-Kristian Halvorsen and Ronald M. Kaplan. 1988. Projections and semantic description in lexical-functional grammar. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1116–1122, Tokyo, Japan. Institute for New Generation Computer Technology.

Susan Hirsh. 1987. P-PATR, a compiler for unification based grammars. In Veronica Dahl and Patrick Saint-Dizier, editors, *Natural Language Understanding and Logic Programming, II.* Elseviers Science Publishers.

Jerry R. Hobbs and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13:47–63.

Riny A. C. Huybrechts. 1984. The weak inadequacy of context-free phrase structure grammars. In G. de Haan, M. Trommelen, and W. Zonneveld, editors, *Van Periferie naar Kern.* Foris, Dordrecht, Netherlands.

Yuji Matsumoto, Hozumi Tanaka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. 1983. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing*, 1(2):145–158.

Michael Moortgat. 1984. A fregean restriction on meta-rules. In *Proceedings of NELS 14*, pages 306–325, Amherst, Massachusetts. University of Massachusetts.

Lee Naish. 1986. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Germany.

Fernando C. N. Pereira and Stuart M. Shieber. 1985. *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, California. Distributed by Chicago University Press.

Fernando C. N. Pereira and David H. D. Warren. 1983. Parsing as deduction. In *Proceedings of the 21st Annual Meeting*, Cambridge, Massachusetts, June 15-17. Association for Computational Linguistics.

Fernando C. N. Pereira. 1981. Extraposition grammars. *Computational Linguistics*, 7(4):243–256, October-December.

Carl Pollard. 1988. Categorial grammar and phrase structure grammar: an excursion on the syntax-semantics frontier. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures.* D. Reidel, Dordrecht, Netherlands.

Stuart M. Shieber. 1985a. *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, California. Distributed by Chicago University Press.

Stuart M. Shieber. 1985b. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Morristown, New Jersey. Association for Computational Linguistics.

Stuart M. Shieber. 1988. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619, Budapest, Hungary.

Mark Steedman. 1985. Dependency and coordination in the grammar of Dutch and English. *Language*, 61(3):523–568.

Jürgen Wedekind. 1988. Generation as structure driven derivation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 732–737, Budapest, Hungary.