Some Open Problems in Head-driven Generation[1]

Dale Gerdemann and Erhard Hinrichs
Seminar für Sprachwissenschaft
Universität Tübingen

1 Introduction

In recent years, a great deal of effort has gone into developing natural language systems that are reversible between parsing and generation (Shieber 1988, Shieber et al. 1990, Gerdemann 1991). A number of advantages of having such a grammar are discussed in Appelt (1987). The primary advantage is that having a single knowledge source ensures consistency between the two modes of processing. The grammar writer can write a single grammar, which will account for both the sentences which can be parsed and the sentences which can be generated. This differs significantly from the older Augmented Transition Network (ATN) approach of Woods (1970) and Bates (1978), in which a completely different ATN would have to be written for each mode of processing.

The idea of having a declaratively represented grammar which is neutral between parsing and generation originated in the Functional Unification Grammar framework of Kay (1979). The fact that unification is both associative and commutative allows processing in any order with the same results. Thus, in principle, there is no problem with using such grammars bidirectionally. The major issue, however, is the question of control strategy. Simple parsers such as the Prolog-based interpreter for definite clause grammar may be run in reverse, but the result is hopelessly slow and in some cases may not terminate.

The most promising control strategy explored in previous studies has been the idea of *head-driven* generation. The basic idea of this approach is that heads should be generated first using top-down information and then the bottom-up information from the heads' subcategorization list should be used to generate its dependents. Although this is a natural idea, there are nevertheless problems with the approach when the head is an empty category or otherwise noncanonical (such as a type-raised NP).

[1]Research reported in this paper was supported by Sonderforschungsbereich SFB 340 sponsored by the Deutsche Forschungsgemeinschaft. The views and conclusions contained herein are those of the authors and should not be interpreted as representing official policies.

2 Head-Driven Algorithms

In principle, it is likely that any parsing algorithm could be adapted to run in reverse for generation. In practice however, two algorithms, Earley's algorithm (Earley 1970) and the left corner algorithm originally due to Rosenkrantz and Lewis (1970), have been most widely used. The reason for the preference of these algorithms is clearly that both allow a mixture of bottom-up and top-down information flow. This mixture is crucial in generation since the top-down semantic information from the goal category must be integrated with the bottom up subcategorization information from the lexicon. Top-down algorithms suffer additionally from what may be called *head recursion*, i.e., the generation analog of left recursion in parsing. Shieber et al. (1990) show that top-down algorithms will fail for rules such as VP $\rightarrow$ VP X, even when these algorithms are extended with goal freezing as proposed in Dymetman and Isabelle (1988).

Although both Earley's algorithm and the left-corner algorithm allow for a top-down/bottom-up mixture, they do so in rather different ways. Earley's algorithm is essentially a bottom-up algorithm in that the nodes of a parse tree are constructed in postorder (i.e., all daughters before the mother). Nevertheless, the top-down information from the prediction step is used to guide this bottom-up construction. The left-corner algorithm, on the other hand, is more truly a mixture of top down and bottom up in that the nodes of the parse tree are constructed in an inorder (i.e., left daughter, then the mother, then the remaining daughters[2]). Thus the left-corner parser can be described as bottom up for the left most daughter in each subtree and top down for the remaining daughters.

When these algorithms are adapted for head-driven generation, however, the tree traversals become rather different in that heads are constructed first, rather than left most daughters. Nevertheless, the basic top-down/bottom-up information flow remain the same. These two algorithms differ also, however, in other respects as we can see by exploring each of them in more detail.

2.1 Adaptations of Earley's Algorithm

Earley's algorithm provides a mixed regime, in which the bottom-up *completion* and *scanning* steps are guided by the top-down *prediction* step. The algorithm, then, is basically bottom up, since the parse tree is constructed by the completion and scanning steps. The easiest way to understand Ear-

[2] The *inorder* and *postorder* tree traversals are defined more precisely in Illingworth (1990).

ley's algorithm is in terms of chart parsing.[3] Suppose, for example, that we want to parse the sentence *The duck dreams.* with the following grammar:

S	$\rightarrow$ NP VP	V	$\rightarrow$ duck|dreams
NP	$\rightarrow$ Det N	N	$\rightarrow$ duck|dreams
VP	$\rightarrow$ V	Det	$\rightarrow$ the

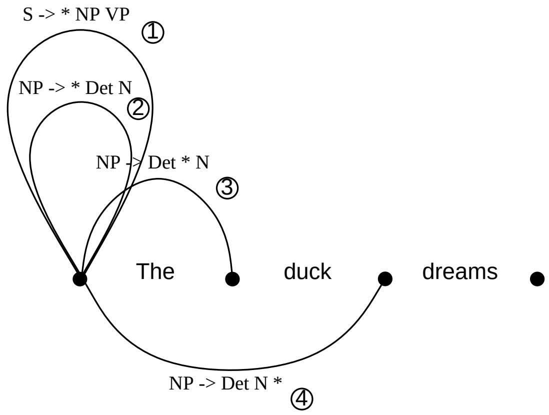


Figure 1: The Chart After Two Predictions and Two Scans.

The parser begins by predicting S $\rightarrow$ * NP VP since the left hand side of this rule matches the start symbol (step 1 in fig. 1). The "dot" in this rule indicates how much of the right hand side has already been recognized, i.e., none. Then, in step 2, the predictor applies to predict NP $\rightarrow$ Det N, since NP matches the category to the right of the dot in the previous prediction. At this stage, the scanner may apply twice (steps 3 & 4) first to scan *the* and then to scan *duck*.[4] Note here that although *duck* is ambiguous between N and V, the top-down prediction only allows it to be parsed as a V.

[3] Chart parsing was introduced by Kaplan (1973) and Kay (1973). Introductory treatments are provided in Pereira and Shieber (1987), Gazdar and Mellish (1989) and Ross (1989). The discussion of Earley's algorithm here will be necessarily brief. For details of the algorithm, the reader is referred to Earley (1970). For a logical reconstruction of this parsing algorithm, see Pereira and Warren (1983). For augmentation of Earley's algorithm with feature structures, see Shieber (1985).

[4] To follow Earley (1970), the predictor would have to first predict every rule of the form N $\rightarrow$ *terminal* and then scanning would be performed just for the prediction N $\rightarrow$ duck. The scanner assumed here avoids this inefficiency by scanning preterminals instead of terminals.

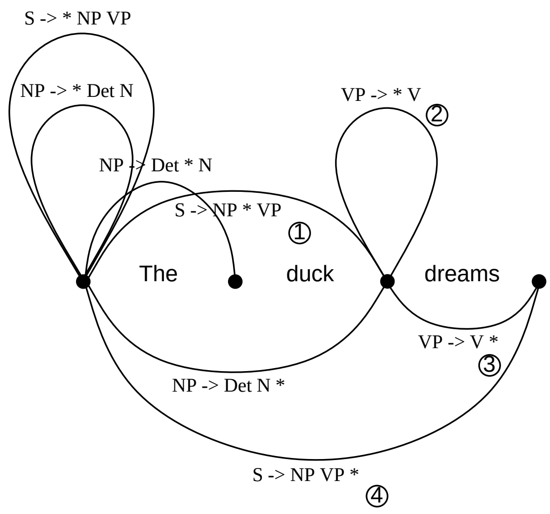


Figure 2: The Finished Chart.

The remaining steps of the parser are then shown in fig. 2. First, since the inactive edge,[5] NP $\rightarrow$ Det N * was added in the previous step,[6] the completer may apply to complete part of the S $\rightarrow$ NP VP rule. The dot is moved to the right of the NP to indicate that this category has been recognized. Then since VP is to the right of the dot, the VP $\rightarrow$ V rule is predicted. This results in the scan of *dreams* in the third step. Notice again that even though *dreams* is ambiguous, the top-down prediction allows it only to be recognized as a V. Finally, since both the NP and the VP have been recognized, the completer recognizes the full sentence.[7]

Now how can this parser be turned into a generator? First, the generator will, of course, need to start with an uninstantiated string and the scanner will need to work in reverse to instantiate this string. In parsing, it makes sense to go from left to right since then the input string can guide the parser. On the other hand, in generation, going from left to right provides

[5] *Inactive/active edges* are referred to as *final/nonfinal states* in Earley (1970) and as *nonconditional/conditional items* in Shieber (1988).

[6] We follow Ross (1989) here in putting active states on the top and inactive states on the bottom.

[7] A parser can be obtained from this recognizer either by building up a tree structure with each scan/completion step or by incorporating features and building up the tree structure as a complex valued feature as described in Pereira and Shieber (1987). The latter approach is more general since it allows a semantic analysis of the sentence to be built up in the same manner as the syntactic structure.

no guidance. It makes more sense to follow the strategy of generating heads first and then complements.[8] The other main difference is that for the generator the start category will need to be much more fully specified. It is pointless to generate just an S. One normally wants to generate a sentence with a particular semantic reading (represented as a feature structure) or perhaps a semantic reading with some additional syntactic properties. To be able to specify such a complex start category, we must assume that Earley's algorithm is augmented to allow feature structures as described in Shieber (1985).

2.1.1 Shieber's Approach

Earley's algorithm was first adapted for generation by Shieber (1988). Shieber's approach is based on Pereira and Warren's (1983) idea of *Earley deduction*. In this approach, each edge is represented as a state $[i, N \leftarrow V_m \ldots Vn, j]$. This state represents an edge from i to j with categories $V_m \ldots Vn$ yet to be found. The categories "to the left of the dot" are ignored. This representation for states differs from Earley (1970) primarily in the index j. In Earley's algorithm, this right index corresponds to the notion of a *state set*. A Shieber state with right index j thus corresponds to an Earley state in the jth state set. We will return to the notion of state sets in the next section, where these sets are manipulated to provide an alternative control structure for generation.

Given this representation of states, the rule of prediction is as shown in (1) and the rule of completion is as shown in (2).

$$(1) \quad \frac{[i, A \leftarrow BC_1 \ldots C_m, j] \quad B' \rightarrow D_1 \ldots Dn \quad \theta = mgu(B\restriction\Phi, B')}{[j, B' \rightarrow D_1\theta \ldots D_n\theta, j]}$$

$$(2) \quad \frac{[i, A \leftarrow BC_1 \ldots C_m, j] \quad [j, B', k] \quad \theta = mgu(B, B')}{[i, A\theta \rightarrow C_1\theta \ldots C_m\theta, k]}$$

The rule of prediction in (1) says that if we have an edge from i to j, with category B to the right of the dot, and if we have a rule whose left hand side unifies with B, then we can infer the instance of this rule with

[8]This head-driven approach has also been applied to parsing by Kay (1989) and van Noord (1991). For parsing, however, this approach is more controversial. In particular, given an input string, it is difficult to guess which element of the string might be the head. In Kay's approach, a chart parser is used in order to save partial results in case some structure is built up from the wrong element. A head-driven parser can, however, make errors which don't lead to useful partial results. For example, in parsing the sentence *The duck swims*, the parser might guess that *duck* is the head, since *duck* is also a verb.

variable substitution θ as an edge from j to j.[9] The variable substitution θ requires restriction as expressed by $B\lceil\Phi$ in order to ensure that prediction terminates (see Shieber (1985), Gerdemann (1989)). The basic problem is that without restriction, an infinite sequence of predictions might be made, where new predictions are not subsumed by previous predictions. Restriction solves this problem by specifying finite bounds on feature values. $B\lceil\Phi$ expresses the category B restricted to just the information specified by the paths in the restrictor Φ. Since restriction involves eliminating information which could be used for top-down prediction, its use in generation is problematic. We will return to this problem in Section 2.1.2.

The completion rule in (2) says that if there is an edge from i to j with B to the right of the dot and if there is an inactive edge B'[10] from j to k, then we may infer a new edge from i to k with the dot moved to the right of B and with the variable substitution θ. Since the lexical items in the input string are encoded as inactive edges, this rule of completion also covers Earley's scanning step.

With the parsing steps expressed as logical inferences, it then becomes possible to explore parsing and generation simply as different theorem proving strategies, which differ in the heuristics they use to infer new items. To parse then *Sonny left*, the parser would start with the items $[0, sonny, 1]$ and $[1, left, 2]$ for the string and the item $[0, S \leftarrow NP\ VP, 0]$ assuming this is the only rule, whose left hand side unifies with the start category. The goal criterion for this case, is to infer an item from 0 to 2 which is subsumed by the start category. This goal can be achieved, however, in different ways. Shieber shows both a breadth-first strategy, which he refers to as Earley style parsing, and a depth-first strategy, which he refers to as shift-reduce style parser. Both parsers are bottom up, however, since the bottom up nature of Earley's algorithm is inherent in the rules of prediction and completion.

Shieber then argues that a generation system could be developed using the same basic architecture but with a different set of heuristics. The initial prediction and the success criterion in such a system would remain the same as in a parser, but it is not clear what to do with the lexicon, since we don't know ahead of time which lexical items will be used or where they will occur. As a first approximation, Shieber claims that, in principle, one could insert each lexical item at each position. He rejects the idea, however, since it would obviously be inefficient.[11] Moreover, he argues that

---

[9] The rule is stated in terms of term unification and variable substitution. The same idea applies for graph unification, though it is somewhat more difficult to express.

[10] This is equivalent to $B' \rightarrow .$ I.e., since elements to the left of the dot are ignored, a final item has an empty right hand side.

[11] In fact, since we don't know in advance how long the generated sentence will be, this

string position indexing is problematic in general for generation, since it may cause a phrase to be redundantly generated at two or more different string positions. For example, a rule allowing adverbs to come optionally before or after a VP, would force the VP to be generated at two string positions depending on where the adverb is placed.

Instead of this approach, Shieber proposes that each lexical item should be inserted at just the single position where $i=j=0$. The same inference rules in (1) and (2) can still be used, but the effect of string position indexing is essentially eliminated. Shieber then erroneously suggests that removing string indexing will cause the algorithm to become non-goal directed in the sense that "many phrases might be built that could not contribute in any way to a sentence with the appropriate meaning." Shieber then goes on to propose a semantic filter to eliminate this supposed problem.[12] In fact however, even with string position indexing eliminated, no phrases can be completed which are not unifiable with some prediction. More precisely, the feature structure of the mother must be unified with the category to the right of the dot in some prediction. The algorithm, in fact, becomes more or less goal directed depending only upon how much information is eliminated by the restrictor in the prediction step.[13]

Thus, it appears that Shieber's algorithm will work, even without the extra semantic filtering that he proposes. The primary disadvantage is that eliminating the state sets may eliminate possible control strategies, such as the strategy proposed in the next section. In general, if multiple strings could be generated from a single goal semantics, it would be reasonable to generate shorter strings before longer strings. It is hard to see how a regime that completely eliminates string indexing could follow this strategy. A second disadvantage is that the generation algorithm no longer corresponds to the parsing as deduction model that Shieber argues for in the case of parsing. The problem is that the lexical items with indices $i=j=0$ corre-

approach involves inserting each lexical item into an infinite number of positions, so the problem is more than just inefficiency.

[12] The semantic filter suggested requires the principle of *semantic monotonicity*, which holds if for each grammatical phrase with semantics *Sem* the semantics of each subphrase is subsumed by some portion, i.e. sub-DAG, of *Sem*. When this condition holds, then any lexical items whose semantics is not subsumed by some portion of the goal semantics can be eliminated before the generation process begins. This of course places a rather strong condition on the grammar. But since Shieber has not convincingly shown that the condition is necessary, we will not consider it further here.

[13] The question of choosing an appropriate restrictor for generation is a complex issue, which Shieber does not discuss. In general, however, the choice of restrictor is even more important in generation than in parsing. In Gerdemann (1991), it is argued that no information corresponding to the semantic translation of a phrase should be eliminated by restriction. It is also argued that it may be advantageous for the restrictor to eliminate other semantic features, such as those corresponding to selectional restrictions, since such features may lead to local ambiguities.

spond to no reasonable premises, i.e., we don't want premises saying that all of the lexical items correspond to the empty string. Moreover, the lexical premises used will not indicate the word order of the string that is generated, so some other extra-logical mechanism is needed for this purpose. More reasonably, lexical items should have variable indices so that they may be inserted at any position. One could have an indexing scheme where $j = i + 1$ (assuming lexical items correspond to single words). Then the precise values for i and j would be determined only when the lexical item is used by the completion inference.

2.1.2 Our Approach

The approach presented in Gerdemann (1991) is more conservative than Shieber's approach in that the state sets as in Earley (1970) are preserved. However, when the predictor would make a prediction which is subsumed by a prediction in a previous state set, a readjustment is made to the chart so that this new prediction will not be necessary. In order to detect cases where duplicate predictions would be made, a record is kept of each restricted category RC that has been used to make predictions.[14] The list of these records is kept in the global restricted category list GRC. Each record in the GRC is of the form $[RC, F, C]$, where F is the number of the state set in which RC was used to make some predictions and C is a list of the completions that have been found so far. In addition to this GRC, states are also modified to include a back pointer BP, which indicates with which restricted category the state was originally predicted.[15] Thus, each state is of the form $[BP, F, A \rightarrow B_1 \ldots B_m]$. For the sake of comparison, the state is written in a similar form to Shieber's *items*. Shieber's left index corresponds to F, and the right index corresponds to the state set number in which this state occurs.

In addition to these chart readjustments, Earley's algorithm also needs to be modified to achieve head-driven generation. For the sake of simplicity, we assume that this is implemented by reordering the right hand sides of all rules so that the head comes first.[16] Thus the order of right hand side categories represents order of processing and not word order. The actual word order can be given by an HPSG-style **phon** attribute as in Gerdemann

[14] We use the term *restricted category* here in order to be neutral between restricted terms or restricted DAGS for term unification or DAG unification, respectively.

[15] In Gerdemann (1989, 1991), these back pointers were added originally to optimize the completion step. In completion the back pointer of the final state must match the forward pointer of the non-final state. The special back pointer r is also used to mark the initial prediction for the termination condition. In this paper, however, we will ignore this additional complication.

[16] The algorithm could, of course, be modified to do this reordering on line.

(1991).[17] Given these additions, the prediction and completion steps are then as follows:[18]

Prediction Given a state $S = [BP, i, A \leftarrow BC_1 \ldots C_m]$ in state set j, let $RC = B \upharpoonright \Phi$. Then:

1. If there exists a $[RC', F, C] \in GRC$ such that $RC' \sqsubseteq RC$, then:
   - Remove S from state set j;
   - Add S to state set F;
   - For each $B' \in C$ such that $\theta = mgu(B, RC)$, add $[BP, i, A \leftarrow C_1\theta \ldots C_m\theta]$ to state set j.
2. Otherwise:
   - Add $[RC, j, []]$ to GRC;
   - For each rule $B' \rightarrow D_1 \ldots D_n$ such that $\theta = mgu(B', B)$, add $[RC, j, B' \leftarrow D_1\theta \ldots D_n\theta]$ to state set j.

Scanning Given a state $S = [BP, i, A \leftarrow BC_1 \ldots C_m]$ in state set j and lexical item B' such that $\theta = mgu(B', B)$, add $[BP, i, A \leftarrow C_1\theta \ldots C_n\theta]$ to state set $j + 1$.[19]

Completion Given a final state $[BP, j, B']$ in state set k. Then:

- For $[BP, F, C] \in GRC$,[20] add B' to C;
- For each $[BP', i, A \leftarrow BC_1 \ldots C_m] \in$ *state set* j such that $\theta = mgu(B', B)$, add $[BP', i, A\theta \leftarrow C_1\theta \ldots C_m\theta]$ to state set k.

Compared to Shieber's Earley deduction approach, this algorithm is certainly more procedural. The readjustments to the chart are really non-monotonic inferences from the perspective of Earley deduction. A readjustment corresponds to retracting a premise with right index j and then reasserting this same premise with right index $j - N$ (for some number N). By including the state sets from the original Earley's algorithm, it becomes possible to do these readjustments in a fairly natural way. The disadvantage

[17] Following the practice of definite clause grammars, this attribute should have a difference list value. Thus, ignoring other features, the standard sentence rule could be written **s(phon(P0,P))** $\rightarrow$ **np(phon(P0,P1)) vp(phon(P1,P))**.

[18] Unlike Gerdemann (1991), the algorithm here uses term unification. No different claims are being made. The difference is only to simplify the presentation and, for the sake of comparison, to make it more similar to Shieber's presentation of Earley deduction.

[19] This rule of scanning assumes that all lexical items consist of single words. A generalization to allow for compounds and traces is given in Gerdemann (1991).

[20] There must be such a member of GRC since new back pointers are only created by the predictor, and whenever the predictor creates a new back pointer, it adds an entry to GRC.

of using a chart, however, is that it takes up considerable space and access times may be slow. The use of the chart is made somewhat more efficient by applying the subsumption test to restricted categories in the first case of the prediction rule (Gerdemann 1989, 1991). In the case of generation, however, these restricted categories will be rather large since they must at least contain the semantic information from the full (unrestricted) category. At least for some grammars, then, it may be advantageous to consider a back-tracking style generator as presented in the next section.

2.2 Adaptations of Left-corner Parsing

The basic idea of left-corner parsing is quite simple. One could think of the left-corner parser as having two basic operations: *matching* (as in a top-down parser) and *reduction* (similar to shift-reduce parsing). Reduction in left-corner parsing differs from reduction in shift-reduce parsing in that reduction in left-corner parsing is made before the full right hand side of a rule is found. In fact, reductions are made as soon as the first element on the right hand side of a rule (i.e. the *left corner* of the rule) is found. This overanxious reduction will only be justified if the list of categories in the rest of the right hand side of the rule can serve as top-down goals to parse some prefix of the remaining input. Once this "justification" has been performed, the left hand side of the rule becomes the new left corner of the input[21] and then further reductions and matches are tried in order to satisfy the remaining goals.

In order to make this idea more concrete, consider the Prolog program in fig. 3 for a rather simple left-corner parser.[22] The first clause is the termination condition: given no more goals, then none of the input can be parsed. The second clause is the matching procedure.[23] Notice here that rather than a single goal, there will always be a list of remaining goals. Each goal may be either terminal or nonterminal. The third clause is the reduction procedure. The first recursive call of **parse** justifies the "overanxious reduction" by parsing the rest of the right hand side for the reduction, and then **parse** is again called recursively to parse the goals with a new left corner.

[21]The term *left corner* is standardly used in this ambiguous way to mean either the left most element of the right hand side of a rule or the left most element in the input string. In reduction, the left most element in the first sense must match the left most element in the second sense.

[22]This parser is given for simple context free grammars. It is quite trivial, however, to add term unification to the rules.

[23]Shieber et al. (1990) argue that this matching step should really use correct unification with the occurs check. Without the occurs check, it is possible that difference lists used for gap-threading would become circular.

```
parse(Cat,String):-
   parse([Cat],String,[]).

parse([],String,String).                          % terminate
parse([Goal|Goals],[Goal|String0],String):-      % match
   parse(Goals,String0,String).
parse([Goal|Goals],[LC|String0],String):-        % reduce
   rule(LHS,[LC|RestRHS]),
   parse(RestRHS,String0,String1),
   parse([Goal|Goals],[LHS|String1],String).

rule(s,[np,vp]). rule(n,[boy]).  % format of rules
```

Figure 3: The Left-Corner Parser

This style of parsing provides a mix of top-down and bottom-up information flow, which makes it attractive to be adapted for generation. The reduction step is clearly bottom up (as in shift-reduce parsing) but each time a reduction is performed, a new set of top-down goals is obtained, which need to be satisfied before any higher reduction can be performed.

Shieber et al. (1990) in their head-corner adaptation of the left corner algorithm (henceforth referred to as the head-corner algorithm) rely on the fact that unification grammars frequently employ *chain rules*, i.e., rules in which the translation of the mother is reentrant with the translation of one of the daughters. For any constituent whose root node is licensed by a chain rule, the daughter whose translation is reentrant with that of the mother can be referred to as the *semantic head*. Assuming that *semantic head* is a transitive relation, the lowest semantic head is referred to as the *pivot*.[24] The pivot itself does not have a semantic head, since otherwise it would not be lowest. Typically, pivots will be lexical items. But it is also possible that the pivot will be an unheaded phrase such as an idiom or possibly a conjunction.[25] The pivot in the head-corner algorithm plays the same role as the left corner in the standard left-corner algorithm.

[24] This is essentially the definition of pivot given by Shieber et al. It may be useful, however, to have a slightly more general notion of pivot. In the left-corner parser, when a left corner is reduced it is replaced by a new left corner. Similarly, it makes sense to say that when a pivot is reduced, it is replaced by a new higher pivot, even though this new pivot is no longer the lowest semantic head.

[25] Conjunctions are not unheaded in all linguistic analyses, but they are unheaded in Gazdar et al. (1985) in the sense that there is not one unique head.

In fig. 4 we give a simplified version of the head-corner generator, which is intended to show the similarity to the left-corner parser in fig. 3. This generator differs from the parser in two main respects. First, phonology must be handled differently since the string is not simply built up from left to right. For simplicity, we assume that phonology is given in lexical entries and rules as the value of an attribute **phon** as in footnote 17. Thus, a result of generating Cat will be that the **phon** attribute of Cat will be instantiated.

```
generate(Cat):-
   generate([Cat],[]).

generate([],[]). % terminate
generate([Pivot|Goals],[Pivot]):- % match
   generate(Goals,[]).
generate([Goal|Goals],[]):- % reduce1
   applicable_non_chain_rule(Goal,Pivot,RHS),
   generate(RHS,[]),
   generate([Goal|Goals],[Pivot]).
generate([Goal|Goals],[Pivot]):-     % reduce2
   applicable_chain_rule(Pivot,LHS,Goal,RHS),
   generate(RHS,[]),
   generate([Goal|Goals],[LHS]).
```

Figure 4: The Head-Corner Generator

The second difference corresponds to the difference between choosing a left corner and choosing a pivot. Choosing a left corner is easy since the left corner is simply the left most element on the input string.[26] Choosing a pivot, however, requires the special procedure **applicable_non_chain_rule** to select a pivot and the procedure **applicable_chain_rule** to reduce a pivot to a higher pivot.[27] The first of these procedures selects a pivot, which will be a non-chain rule for which the semantic features are unified with the goal semantic features. The second procedure selects a chain rule and unifies the current pivot with the semantic head of the rule. The left hand side of this chain rule then becomes the new pivot. Both of the procedures incorporate linking (Matsumoto et al. 1983, Pereira and Shieber 1987) so

[26] The term *input string* is perhaps misleading since the algorithm in fig. 3 allows non-terminals to be pushed onto this list after a reduction is made.

[27] These procedures are defined in Shieber et al. (1990).

that a pivot will only be used if it can be connected to the current goal by a series of chain rules.

One potential problem for this generator is that for certain types of grammars it may not terminate. The problem arises in the recursive call **generate(RHS,[])** in the third and fourth clauses. If, for example, the top level goal is to generate an S and an embedded S occurs as a member of **RHS**, then, unless the features on the embedded S are significantly different from the features on the top level goal S, the generator will go into a loop. The left-corner parser has no trouble with such recursion since (assuming off line parsibility[28]) each loop will involve consuming some of the input. The Earley generator, as well, would have no problem with such cases, since the algorithm is primarily bottom up and the subsumption test in the predictor rules out making duplicate top-down predictions. It is not clear, however, whether or not this is a practical problem. It is likely that for reasonable grammars, the features on goal and embedded categories will be different enough so that it will not go into a loop.

A more serious problem for both the Earley generator and the head-corner generator is that they both make assumptions about headedness that may not be valid for all grammars. Both generators will be significantly slower when some of the rules are non-headed. Moreover, non-canonical heads, such as empty heads and type-raised heads, are problems for both algorithms as described in the next section.

3 Problematic Constructions for Head-Driven Generation

Ideally, a reversible approach to parsing and generation should allow precisely the same grammars for both directions of processing. In practice, however, generation systems have placed restrictions on the grammar that are not required for parsing (Russell et al. 1990). The primary restriction imposed by generation systems is the chain-rule restriction of Shieber et al. (1990), requiring that phrases should generally have semantic heads. Beyond just having heads, however, these algorithms impose restrictions on the types of heads involved. Empty heads (Section 3.1), for example, are problematic since typically they inherit their subcategorization lists from elsewhere. A similar problem is posed by type-raised functors in Categorial

[28]The idea of off line parsibility (Kaplan and Bresnan 1982) is that a grammar should not contain chains of rules allowing a non-terminal N to derive just N. Obviously, if such chains exist then (assuming that the features on the top N are the same as the features on the bottom N are the same) the parser could loop forever. This idea is extended to PATR syle grammars (without a context free backbone) in Shieber (1989).

Unification Grammar (Section 3.2).[29] If these functors count as heads, the generation process will not be sufficiently constrained, since these functors clearly do not contain the proper subcategorization information to guide the generation process. Other deviations from head-complement types of structure, such as specifier-head or head-adjunct, may cause problems for such generators as well, though the problems for these constructions have been studied less thoroughly.

3.1 Empty Heads

As pointed out previously (Shieber et al. 1990, Russell et al. 1990), the position of verbal elements in main clauses of modern German provides an interesting challenge to the family of head-driven generation algorithms that we presented in Section 2. We will briefly review some basic facts about German word order in different clause types, before we can comment on the issues that the syntactic distribution of verbal elements raises for head-driven generation.

Sentences in modern German exhibit systematic differences with respect to the placement of the finite verb. Subordinate clauses, as the *daß*-clauses in (3), obey SOV order, with the finite verb occurring in sentence final position, regardless of whether it is an auxiliary such as *wird* in (3a) or a main verb such as *kaufte* in (3b).

(3) a. *Fritz glaubt, daß Peter das Buch kaufen wird.*
Fritz believes that Peter the book buy will.
'Fritz believes that Peter will buy the book.'

b. *Fritz glaubt, daß Peter das Buch kaufte.*
Fritz believes that Peter the book bought
'Fritz believes that Peter bought the book.'

If the main clause is a yes/no question, as in (4), the finite verb occurs in sentence-initial position. In the case of constituent questions, as in (5), or assertion clauses, as in (6), the finite verb appears in second position, following a single topicalized constituent. Again, this generalization holds for main verbs and auxiliaries alike, as (4)–(6) show.

(4) a. *Wird Peter das Buch kaufen?*
Will Peter the book buy
'Will Peter buy the book?'

[29] By this term we intend to refer to various unification-based extensions to categorial grammar (Uszkoreit 1986, Bouma 1987) and including what has also been called Unification Categorial Grammar (Zeevat 1988).

b. *Kaufte Peter das Buch*?
Bought Peter the book
'Did Peter buy the book?'

(5) a. *Welches Buch wird Peter kaufen*?
Which book will Peter buy
'Which book will Peter buy?'

b. *Welches Buch kaufte Peter*?
Which book bought Peter
'Which book did Peter buy?'

(6) a. *Peter wird das Buch kaufen.*
Peter will the book buy
'Peter will buy the book.'

b. *Peter kaufte das Buch.*
Peter bought the book
'Peter bought the book.'

In assertion main clauses, the sentence-initial topicalized constituent is not limited to the subject of the sentence. Rather direct objects, as in (7a), or non-finite verbal constituents, as in (7b), can appear in sentence-initial position as well.

(7) a. *Das Buch wird Peter kaufen.*

b. *Kaufen wird Peter das Buch.*

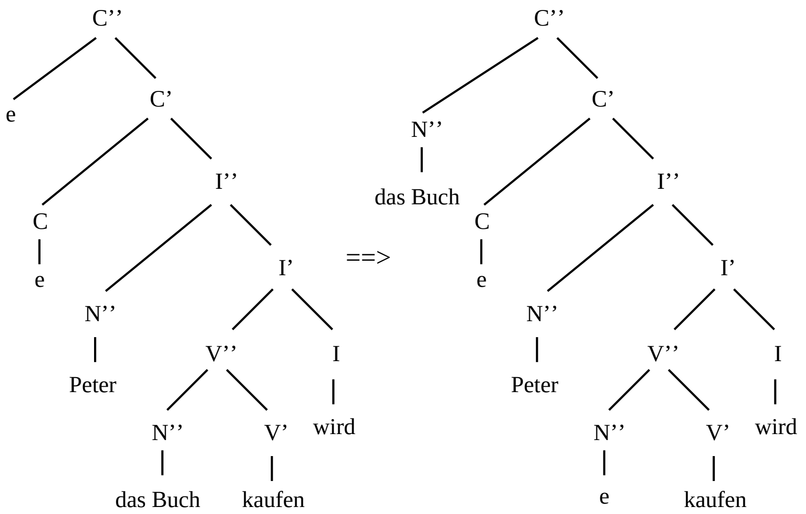
In fact, a descriptive generalization about assertion main clauses that virtually any syntactician will agree on, is that any major syntactic constituent can appear sentence-initially, as long as the topicalized material constitutes a **single** constituent.

It is generally agreed upon in the GB literature (cf. Haider and Prinzhorn 1986, von Stechow and Sternefeld 1988) and its precursors (cf. Thiersch 1978) that the word order in subordinate clauses is the basic word order, with the word order in main clauses being derived transformationally via move-alpha. There is no general agreement as to the structural position that the finite verb moves to at S-structure (INFL and COMP being the two chief options), nor as to the factors that in such analyses condition the obligatory fronting of finite verbs in main clauses (see Haider (1986) for an review of the issues involved).

One way to flesh out such a movement analysis of verb-second position is to posit a mapping between D-structure and S-structure as shown in (8).

The d-structure would map into the corresponding s-structure by two applications of move-alpha: movement of the N″ into the head position of C″ and of the finite verb into the head position I″.

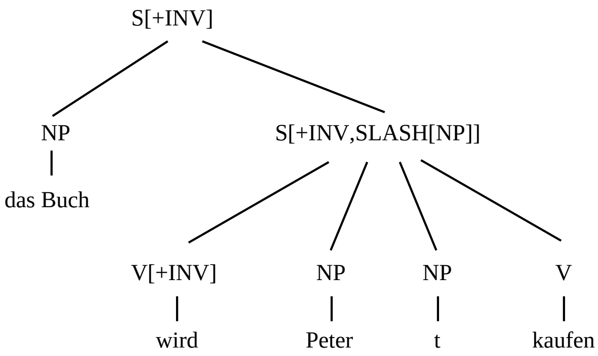
(8)



Monostratal analyses of German main clauses, on the other hand, have challenged the view that verb-initial or verb-second position of the finite verb in German is due to syntactic movement. Uszkoreit (1985) in the framework of GPSG and Pollard (1990) in the framework of Head-Driven Phrase Structure Grammar have proposed analyses in which the finite verb is generated as a sister of S, regardless of clause type. The position of the finite verb in a particular clause type is accounted for by linear precedence rules which impose constraints on the linear order of (sister) constituents. According to Pollard's analysis, an assertion main clause such as *Das Buch wird Peter kaufen* is assigned the structure in (9).

The placement of the finite verb in Pollard's analysis is conditioned by a linear precedence rule which requires that a constituent carrying the feature specification [+INV] precede its sister constituents. If finite verbs in subordinate clauses are marked by the feature specification [-INV] and a linear precedence rule is added which requires that a constituent carrying the feature specification [-INV] has to follow its sister constituents, then the correct word order for subordinate clauses can be effected as well.

(9)



While monostratal and GB-style analyses differ considerably in their treatment of the verb-second phenomenon, both types of syntactic approaches show remarkable similarity in the treatment of sentence-initial topicalized constituents in assertion main clauses and constituent questions. Regardless of syntactic framework, it is generally agreed upon that topicalization involves a syntactic dependency between the fronted constituent and a syntactic position marked by a trace. The main difference between monostratal analyses à la GPSG or HPSG and GB-style multistratal analyses involves the mechanism that accounts for this dependency: a slash mechanism or a transformational account via move-alpha, respectively. It goes beyond the scope of this paper to enter into a full discussion of the verb-second or the topicalization phenomenon, two of the central topics of German syntax. However, see Hinrichs and Nakazawa (1989a, 1989b) for additional discussion.

For a head-driven strategy, the displacement of verbal constituents via topicalization or via verb-fronting—if a movement analysis is assumed—can cause serious problems. Assuming that the verbal trace is the lexical head of the sentence, then the first lexical item that the generator will try to generate is this verb in final position. Since the verb turns out to be gapped, only a trace is generated at this point. However, it is imperative for efficient generation that the generator find the full lexical entry for the verb at this point in order to know what complements to generate.

Notice that this problem is going to arise for monostratal and multistratal analyses alike when the strategy of head-driven generation is applied to sentences in which a verbal constituent is topicalized. This is due to the fact that both types of analyses presuppose that there is a syntactic dependency between such a fronted constituent and a trace within the VP. In addition, the same type of problem arises in the case of sentences whose finite verb is in second position, if verb-second placement is treated via a trace-mechanism.

From the perspective of computational linguistics, it is highly desirable to develop truly general algorithms for natural language generation that can accommodate a variety of well-motivated analyses of the same syntactic or semantic phenomenon. Since the problems for head-driven generation are particularly pervasive for those analyses that treat topicalization and verb-second phenomenon via a trace mechanism, we will concentrate on such GB-style analyses in the remainder of this section. In order to clarify this problem further, it will be useful to consider a specific grammar. Since virtually all head-driven generation algorithms presuppose unification-based grammar formalisms for the encoding of grammatical information, we will attempt to recast a movement style analysis of topicalization and verb-second in German in such a unification-based framework. The resulting analysis is mono-stratal. In fact, the grammar we present here is encoded in a simplified version of HPSG, though the same problem can easily be illustrated within other linguistic frameworks.

We will begin considering the lexical entry for a verbal trace in this grammar. This entry is shown in (10), whereas the lexical entry for the non-null verb *gibt* is show in (11).

(10)
$$\begin{bmatrix} \text{head:} & [1]\begin{bmatrix} \text{maj:} & \text{vp} \end{bmatrix} \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & \begin{bmatrix} \text{head:} & [1] \\ \text{subcat:} & [2] \end{bmatrix} \\ \text{top-slash:} & \text{none} \end{bmatrix} \\ \text{subcat:} & [2] \end{bmatrix}$$

(11)
$$\begin{bmatrix} \text{head:} & \begin{bmatrix} \text{maj:} & \text{vp} \\ \text{sem:} & \text{give([1],[2],[3])} \end{bmatrix} \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & \text{none} \\ \text{top-slash:} & \text{none} \end{bmatrix} \\ \text{subcat:} & \left\langle \begin{bmatrix} \text{cat:} & \text{np} \\ \text{sem:} & [3] \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{np} \\ \text{sem:} & [2] \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{np} \\ \text{sem:} & [1] \end{bmatrix} \right\rangle \end{bmatrix}$$

In these lexical entries, the **v2-slash** attribute may take as its value the verbal category that appears in second position and the **top-slash** attribute may take as its value a category to be topicalized. The **none** value for these attributes is simply an arbitrary atomic value, which indicates that a complex value is not allowed. The primary difference between these two lexical entries is that whereas *gibt* has a fully instantiated subcategorization list, the trace simply indicates that its **subcat** value is the same as the

subcat value in the category value of the **v2-slash** attribute. The idea that this lexical entry for trace represents, is that traces do not have inherent properties such as subcategorization, but rather inherit these values from elsewhere. Although the lexical entry in (10) is cast in the framework of HPSG, any other linguistic theory that allows traces would also need a way of expressing the fact that traces have inherited properties.[30]

The grammar should allow categories to be cancelled off of the subcategorization list in either of two ways. The rule in (12) allows a VP to combine with a category that it is subcategorizing for and the rule in (13) allows a category to be moved from the subcategorization list to **top-slash**.[31]

(12)
$$\begin{bmatrix} \text{head:} & [1] \\ \text{bind:} & [2] \\ \text{subcat:} & [4] \\ \text{dtrs:} & \begin{bmatrix} \text{head-dtr:} & \begin{bmatrix} \text{head:} & [1] \\ \text{bind:} & [2] \\ \text{subcat:} & \begin{bmatrix} \text{first:} & [3] \\ \text{rest:} & [4] \end{bmatrix} \end{bmatrix} \\ \text{comp-dtrs:} & \langle\ [3]\ \rangle \end{bmatrix} \end{bmatrix}$$

(13)
$$\begin{bmatrix} \text{head:} & [1] \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & [2] \\ \text{top-slash:} & [3] \end{bmatrix} \\ \text{subcat:} & [4] \\ \text{dtrs:} & \begin{bmatrix} \text{head-dtr:} & \begin{bmatrix} \text{head:} & [1] \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & [2] \\ \text{top-slash:} & \text{none} \end{bmatrix} \\ \text{subcat:} & \begin{bmatrix} \text{first:} & [3] \\ \text{rest:} & [4] \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Finally, the sentence rule is shown in (14). For simplicity, we have assumed flat structure at the top even though most linguistic analyses assume a binary branching structure.

---

[30] One alternative approach which avoids the use traces altogether is suggested by current research in extended categorial grammar (Steedman 1985; Moortgat 1988, 1990). These researchers account for unbounded dependencies by including non-Lambek versions of functional composition and type raising (Steedman) or by adding discontinuous type constructors (Moortgat). While we believe that this line of research is a very promising one to pursue, very little work has been done so far on applying these formalisms to the problem of natural language generation (however, see van der Linden and Minnen 1990).

[31] The Head Feature and Binding Inheritance principles are, in effect, incorporated into these rules. Since there are so few rules in HPSG, it makes little computational sense to try to extract out general principles applying across all rules.

(14)
$$\begin{bmatrix} \text{head:} & [1] \\ \text{dtrs:} & \begin{bmatrix} \text{head-dtr:} & \begin{bmatrix} \text{head:} & [1] \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & [2]\ {}_{complex}[\,] \\ \text{top-slash:} & [3]\ {}_{complex}[\,] \end{bmatrix} \\ \text{subcat:} & \text{none} \end{bmatrix} \\ \text{comp-dtrs:} & \langle\ [2],\ [3]\ \rangle \end{bmatrix} \end{bmatrix}$$

The most important thing to notice about this rule is that the head daughter is designated to have complex values for both of its **slash** attributes. As a sentence is generated, these values will percolate downward. The rule in (13) will allow the value for **top-slash** to become **none**, though the complex value for **v2-slash** will percolate all the way down to the lexical head. Thus, given the start DAG [head|sem|give(hans,marie,book)], when the generator gets to the point of generating a lexical item, this start DAG will have become instantiated to the DAG in (15).

(15)
$$\begin{bmatrix} \text{head:} & \begin{bmatrix} \text{maj} & \text{vp} \\ \text{sem:} & \text{give(hans,marie,book)} \end{bmatrix} \\ \text{bind:} & \begin{bmatrix} \text{v2-slash:} & {}_{complex}[\,] \\ \text{top-slash:} & \text{none} \end{bmatrix} \end{bmatrix}$$

The goal DAG at this point, of course, contains all of the needed semantics but still it will not unify with the lexical entry for *gibt* since the complex value for **v2-slash** will not unify with the atomic value **none** in the lexical entry for *gibt*. So at this point it is still not possible to generate this lexical item with its subcategorization list.

This complex value for **v2-slash** will, however, unify with the complex value for **v2-slash** in the lexical entry for the trace in (10). Thus, only the trace can be generated as the lexical head. Once the lexical head is generated, the subcategorization list of this head needs to be used to build the rest of the phrase bottom-up. But the subcategorization list of the trace in (10) is completely uninstantiated, so the generator will, at this point, have no constraints at all that can be used to build the rest of the phrase. The semantics of the phrase is available, as shown in (15), but it is not known how this semantics corresponds to a subcategorization list.

One strategy to avoid this problem was proposed by Shieber et al. (1990). These authors propose that rules such as (14), which introduce gap filler constructions, should be given some kind of mark or flag by the grammar writer. Then when the generator encounters such a flagged rule, the head-driven strategy can be temporarily abandoned in order to generate the filler first. In the grammar given here, the head features of the trace (including the semantic value) are identical to the head features of the verb in second position. Thus, the goal semantics could be used to immediately generate the second position verb before generating the head daughter. If this is done, then when the trace is generated as the lexical head of the head daughter, it will have an instantiated subcategorization list.

A general problem with this idea is that it reduces the modularity between the grammar and the processor. In this case, the grammar writer needs to be aware of processing concerns in order to properly annotate the rules.

A more specific problem with this approach is that it will fail when the same rule sometimes results in the displacement of a head and sometimes results in the displacement of a non-head. Topicalization appears to be an example of such a rule. In (16), a lexical non-head *Das Buch* is topicalized, while in (17) the lexical head *lesen* is topicalized.[32]

(16) Das Buch lesen wird Peter.

(17) Lesen wird Peter das Buch.

Given the existence of rules of this sort, it will not be possible for the grammar writer to determine once and for all whether or not the topic constituent needs to be generated first. Whether or not the topic needs to be generated first depends on how the topic is instantiated in a particular application of the rule.

Another strategy for dealing with dependencies is suggested in Gerdemann (1991). The strategy suggested there was to treat all dependencies as being strictly phonological. Given the existence of a **phon** attribute as in HPSG, it would be possible for the phonology of a constituent to pass up a tree, say as the value of a **slash** type of attribute. Then it would be possible for a phrase to "actually" occur in a lower part of a tree, while its phonology occurs somewhere higher in the tree. In this analysis, a phonological dependency could be introduced by a rule such as that in (18).

(18)
$$\begin{bmatrix} \text{phon:} & \langle\,\rangle \\ \text{head:} & [1] \\ \text{bind:} & \begin{bmatrix}\text{phon-slash:} & [2]\end{bmatrix} \\ \text{dtrs:} & \begin{bmatrix} \text{head-dtr:} & \begin{bmatrix}\text{phon:} & [2] \\ \text{head:} & [1] \\ \text{bind:} & \text{none}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

This approach, then, would allow for certain mismatches between syntax and phonology. The idea is reminiscent of Hendrix's (1987) idea of having

[32] *Lesen* is the head here in the sense that it contains the relevant subcategorization list (Hinrichs & Nakazawa 1989b).

different algebras apply in the syntax and semantics to account for quantifier scopings that don't match surface word order. More generally, the idea is similar to the idea of having multiple levels of representation in Government Binding Theory or Lexical Functional Grammar. It would be interesting to compare these various approaches to see how they relate.

One problem with this approach, though, is that with such phonological dependencies, it may no longer be possible to parse in a strictly left-to-right fashion. Thus it may turn out that the optimal grammar for generation is no longer optimal for parsing. There appears to be two possible solutions to this problem. One idea would be to develop new parsing algorithms which could deal with such dependencies. Or a second idea would be to use a grammar compiler to produce separate grammars for parsing and generation.

At this point, we do not know which of the above solutions will ultimately be preferable. We conjecture that the problems involved in efficient generation can only be fully understood if the range of data is significantly expanded. Although we have concentrated here on the problem of verb second constructions in German, it may ultimately turn out that other phenomena are even more problematic. In this regard, cases of VP ellipsis (Sag 1976), empty head nouns (Nerbonne et al. 1989) or the possibility of empty prepositions in phrases such as *next Tuesday* (Larson 1985, McCawley 1988) constitute related phenomena that will require close attention.

3.2 Type Raising in Categorial Unification Grammar

The problem of empty heads can be thought of as a special case of the general problem of applying the head-driven strategy to lexical heads whose semantics is not sufficiently specified. One limiting case is, of course, a lexical entry whose semantics is specified only in terms of a reentrancy. It may often be the case that empty phonology correlates with empty semantics as in the case of a trace, but this need not be the case. In versions of categorial unification grammar, for example, the lexical entries for type-raised NPs may have unspecied semantics, though the phonology is certainly not null. In Gerdemann and Hinrichs (1990), we give an example of such a grammar. In this simple grammar, there is just the one function application rule in (19), which conforms to the chain rule condition, and yet, given type-raised lexical entries for proper nouns, the generator may go into an unconstrained search.

(19)
$$\begin{bmatrix} \text{x0:} & [1]\begin{bmatrix}\text{trans:} & [2]\end{bmatrix} \\ \text{x1:} & \begin{bmatrix} \text{cat:} & \begin{bmatrix}\text{result:} & [1] \\ \text{arg:} & [3]\end{bmatrix} \\ \text{trans:} & [2]\end{bmatrix} \\ \text{x2:} & [3]\end{bmatrix}$$

(i.e. X/Y Y $\rightarrow$ X)

The PATR-style rule in (19) clearly corresponds to the function application rule of categorial grammar, though there are two main differences. Most importantly, we satisfy the chain-rule condition by having the translation of the mother reentrant with the translation of the functor. The other difference is that forward and backward application are collapsed into one rule. This collapse is possible since phonology is handled by means of a **phon** attribute which takes difference list values. This allows functors to specify either left or right concatenation of phonologies.[33]

As is shown in Gerdemann and Hinrichs (1990) and Gerdemann (1991), the function application rule (19) may lead to unconstrained generation (i.e. generate and test) when proper nouns are given type-raised lexical entries as they would be in Montague (1973). The problem is, essentially, that when a proper noun is given a type-raised entry, its translation will have to be reentrant with the translation of the functor it combines with. This is the reentrancy marked with number 2 in the lexical entry for a proper name like *Tom* (20).

(20)
$$\begin{bmatrix} \text{cat:} & \begin{bmatrix} \text{result:} & [1] \\ \text{arg:} & \begin{bmatrix} \text{cat:} & \begin{bmatrix} \text{result:} & [1] \\ \text{arg:} & \begin{bmatrix} \text{cat:} & \text{np} \\ \text{trans:} & \text{tom} \end{bmatrix} \end{bmatrix} \\ \text{trans:} & [2] \end{bmatrix} \end{bmatrix} \\ \text{trans:} & [2] \end{bmatrix}$$

To understand this type-raised entry, suppose that this higher functor combines with the first order functor *smiles* whose lexical entry is given in (21).

(21)
$$\begin{bmatrix} \text{cat:} & \begin{bmatrix} \text{result:} & \begin{bmatrix}\text{cat:} & \text{s}\end{bmatrix} \\ \text{arg:} & \begin{bmatrix} \text{cat:} & \text{np} \\ \text{trans:} & [1] \end{bmatrix} \end{bmatrix} \\ \text{trans:} & \begin{bmatrix} \text{pred:} & \text{smiles} \\ \text{arg1:} & [1] \end{bmatrix} \end{bmatrix}$$

[33] For details, see Gerdemann and Hinrichs (1990) and Gerdemann (1991).

The entry for *smiles* specifies a function from **np** to **s**. The translation for this functor is (translated into a term) **smiles(X)**, where the value of **X** comes from the argument that *smiles* combines with. When *tom* combines with *smiles*, the resulting translation cannot be just the translation of *smiles*, but rather the translation that *smiles* would have if it were to combine with a non-type raised lexical entry for *tom*. The path ⟨cat arg cat arg trans⟩ in the lexical entry for *tom* "feeds" this non-type raised semantics for *tom* into ⟨cat arg trans⟩ path of *smiles*. Thus the translation of *tom* after it has combined with *smiles* is just the translation of *smiles* with this argument position instantiated.

To generate the sentence *tom smiles*, the generator would begin by unifying the goal semantics with the translation for **x0** in (19). The result of this unification is shown in (22).

(22)
$$\left[\begin{array}{ll}
\text{x0:} & [1]\left[\begin{array}{ll}\text{trans:} & [2]\left[\begin{array}{ll}\text{pred:} & \text{smile}\\ \text{arg1:} & \text{tom}\end{array}\right]\end{array}\right]\\
\text{x1:} & \left[\begin{array}{ll}\text{cat:} & \left[\begin{array}{ll}\text{result:} & [1]\\ \text{arg:} & [3]\end{array}\right]\\ \text{trans:} & [2]\end{array}\right]\\
\text{x2:} & [3]
\end{array}\right]$$

Now the generator must find the lexical entries for *tom* and *smiles*. Since the resulting parse tree will be only of depth 1, there is essentially no difference between top-down, bottom-up or head/left-corner generation. In any case, the first thing the generator must find is the head (i.e., the functor). Clearly the lexical entry for *tom* will unify with the **x1** path in (22), but so would the lexical entry for any other proper noun in the lexicon. Suppose, for example, that the **x1** path unified with the lexical entry for *harry*. Then the instantiated rule would appear as in (23).

(23)
$$\left[\begin{array}{ll}
\text{x0:} & [1]\left[\begin{array}{ll}\text{trans:} & [2]\left[\begin{array}{ll}\text{pred:} & \text{smile}\\ \text{arg1:} & \text{tom}\end{array}\right]\end{array}\right]\\
\text{x1:} & \left[\begin{array}{ll}\text{cat:} & \left[\begin{array}{ll}\text{result:} & [1]\\ \text{arg:} & [3]\end{array}\right]\\ \text{trans:} & [2]\end{array}\right]\\
\text{x2:} & [3]\left[\begin{array}{ll}\text{cat:} & \left[\begin{array}{ll}\text{result:} & [1]\\ \text{arg:} & \left[\begin{array}{ll}\text{cat:} & \text{np}\\ \text{trans:} & \text{harry}\end{array}\right]\end{array}\right]\\ \text{trans:} & [2]\end{array}\right]
\end{array}\right]$$

The unification with *harry* succeeds, but then the **x2** path will no longer be able to unify *smiles* (21). So what a head-driven generator will need to do is to try unifying every possible proper noun with the **x1** position and then weed out those cases where this causes failure to generate anything for the **x2** position. Clearly, this is a case of generate and test.

As seen then from this example, type-raised NPs provide another case where unspecified semantics may lead a head-driven generator into uncontrolled search. As noted in Gerdemann and Hinrichs (1990), the problem does not arise with higher type NPs in general, but rather only with proper NPs which are type raised following Montague's strategy of generalizing to the worst case. It seems then, that the generator imposes the constraint on the grammar writer that NPs should be assigned minimal types and then only raised when necessary.[34] Interestingly, the principles of *minimal type assignment* and *type raising only as needed* have been argued for on entirely independent semantic grounds by Partee and Rooth (1983). This seems to be one of the interesting cases where constraints imposed by generation systems correspond to the constraints which are linguistically motivated.

The question of how *type raising only as needed* can be incorporated into the grammar, however, is a tricky issue. In Gerdemann and Hinrichs (1990), we advocate an approach based on *supercombinators* as discussed in Wittenburg (1987) and Wall and Wittenburg (1989). The basic idea of this approach is to combine two (or possibly more) rules into one encompassing rule. Thus, for example, if type raising is only needed in the context of coordination, then type raising could be built into the coordination rule. This, then, forces type raising only where it is needed.

Calder et al. (1989) provide an alternative approach to generation of type-raised NPs in the context of Unification Categorial Grammar.[35] Calder et al. use essentially the same head-corner algorithm as in Shieber et al. (1990). However, since all proper names have type-raised lexical entries, a special case needs to be added to the algorithm to handle this case. In their approach, when a type-raised NP combines with a VP (i.e., s/np), the normal functor driven routine is reversed so that the VP is generated first. Since the VP requires a non-type raised NP argument, this argument is first

[34] Type raising would be necessary, e.g., if an untyped NP were to coordinate with a higher type quantifier NP.

[35] Unification Categorial Grammar (UCG) is a unification version of categorial grammar that encodes directionality in types by means of an **order** attribute taking values **pre** and **post**. Thus, simplifying somewhat, **A:pre** is a simple type which precedes its functor and **A/(B:post)** is a functor taking a following argument **B**. This encoding of types becomes highly problematic, however, in the case of type raising. This is so since type raising in their approach reverses direction of combination; e.g., an NP marked **pre** becomes a functor looking for an argument which is also marked **pre**. In effect, Calder et al. allow only harmonic type-raising to the exclusion of the disharmonic type raising standardly used in categorial grammar. Since Calder et al. incorporate type raising into their generation algorithm, this is a problem not only for their grammar, but also for their algorithm.

raised and then the generation procedure is recursively called to generate this type-raised argument.[36] There are two main problems with this approach. First, as we argue in Section 3.1, the general strategy of marking specific rules as exceptional for the generator does not always work, since at least some rules may require exceptional treatment only in special cases. Second, Calder et al.'s approach is fixed specifically for a grammar in which all proper NPs are given type-raised lexical entries. But, as shown by Partee and Rooth (1983), there are strong semantic arguments against such a treatment. Since Calder et al.'s approach lacks modularity between the processor and the grammar, it would be difficult to modify it for a different grammar which would conform to Partee and Rooth's principles of *minimal type assignment* and *type raising only as needed*.

The Gerdemann and Hinrichs (1990) approach and the Calder et al. (1989) approach provide examples of two general approaches to the problem of applying the head-driven strategy to heads whose semantics is not sufficiently specified. In the Gerdemann and Hinrichs approach, restrictions are placed on the form of the grammar, whereas in the Calder et al. approach, the algorithm is extended to handle special cases. It is still an open question which of these two approaches is better, since at present, neither is fully adequate. If restrictions on grammar are to be accepted, then one would like a general mechanism to enforce these mechanisms. For example, one might want a compiler to automatically create the supercombinators required in Gerdemann and Hinrichs (1990). On the other hand, if extensions are made to the generator, then one would like these extensions to generalize to a variety of cases rather than ad hoc extensions for each special case.

4 Completeness and Coherence

A further problem for unification-based generation systems is that they normally only generate phrases whose semantics is unifiable with the goal semantics. The semantics of the generated string may be either more or less specific than the goal semantics. Wedekind (1988) has dubbed this the *completeness* and *coherence* problem for generation. If the goal semantics always subsumes the semantics of the generated string then, the generator is complete. And if the semantics of the generated string always subsumes the goal semantics, then the generator is coherent.

A coherent generator may be thought of as one that does not overgenerate. For example, if the semantic representation for *John eats* is **eat(john,X)**, the incoherent generator may instantiate variable **X** and gen-

[36]In fact, for complex reasons, this recursive call to **generate** causes their algorithm, as presented in the appendix, to go into an infinite loop.

erate *John eats peanuts*. A complete generator, on the other hand, may be thought of as a generator that does not undergenerate. For example, if the goal semantics is **eat(john,peanuts)**, then the incomplete generator may simply generate the string *John eats*. As seen from these examples, completeness and coherence become practical problems most when the semantics representations follow the common practice of using the Prolog (metalanguage) variables for (object language) variables in the semantic representation. This use of Prolog variables causes a problem since the incoherent generator may simply instantiate the variable **X** in **eat(john,X)** to **peanuts** in order to generate the string *John eats peanuts*. Similarly, if the lexical entry for intransitive *eat* is of the form **lex(subcat([sem(X)]), eat(X,Y))**, then the incomplete generator may simply instantiate the variable **Y** in this lexical entry to **peanuts** in order to generate *John eats* from the goal semantics **eat(john,peanuts)**.

The naive way to solve the completeness/coherence problem is to follow the strategy of generate and test. This, however, will be very inefficient since many strings may be generated only to be weeded out by the completeness and coherence tests. Shieber et al. (1990) suggest a better way to handle the coherence problem. They suggest that this problem could be solved if variables in the goal logical form were not represented by Prolog variables but rather by constants.[37] Thus if the goal semantics for *John eats* is **eat(john,var23)**, there is no possibility that the constant **var23** could be instantiated to *peanuts* to generate *John eats peanuts*.

For completeness, however, Shieber et al. can only suggest a generate and test type of strategy. What they fail to notice is that the completeness problem could also be solved in an entirely analogous manner if Prolog variables in lexical entries were instantiated to constants. Thus, if the lexical entry for intransitive *eat* were **lex(subcat([sem(X)]),eat(X,var19))** then it would not be possible for the semantics of this entry to unify with the goal semantics **eat(john,peanuts)**. Unfortunately, it would also not be possible to generate an intransitive sentence with the goal semantics **eat(john,var23)** since the two variables will not unify. So it seems that the practice of numbering variables allows one to solve either completeness or coherence but not both at the same time.

In Gerdemann (1991) an alternative is proposed within the framework of graph unification. In the term unification approach, each different variable must be given a different unique constant name. In the graph unification approach, however, all variables could be represented by the same constant, say **var**. Then the distinction between same and different variables can be recaptured by using reentrancies. Instead of having two instances of the

[37] They suggest using Prolog procedure **numbervars** may be used for the purpose of turning Prolog variables into constants.

same variable type, it is possible in graph unification for the two variables to be represented by exactly the same variable token.[38]

This approach of using atomic values for variables in logical form goes in the direction of making completeness and coherence the grammar writer's responsibility. This is even more the case for the problem of adjuncts. One might imagine that the logical form for *John eats* subsumes the logical form for *John eats quickly*. This will be the case, for example, if the logical forms are [**pred: eat, arg1: john**] and [**pred: eat, arg1: john mod: quickly**] respectively. Again, it would have to be the grammars writer's responsibility to ensure that logical forms do not subsume each other in this manner. In this case, however, there is a good alternative. Instead of having the **mod** attribute take an atomic value, it would be more reasonable to say that **mod** takes as its value a list of modifiers.[39] Then the first logical form should really contain the **mod** attribute with an empty list as a value. Since the empty list will not unify with a non-empty list, the completeness/coherence problems are again solved. Thus the completeness/coherence problems can be avoided, but the burden is really placed on the grammar writer.

Another approach to this problem might be to consider why completeness and coherence do not create problems for the parser. In principle, of course, these problems could arise for a parser. A parser that further instantiated the input string would be "incoherent". Whereas a parser that returned a parse after only consuming part of the input string would be "incomplete". It's clear that these problems do not arise because parsers are designed to go from left to right through the input string and terminate precisely at the end of the input list. It would be interesting to consider whether a generator could traverse an input logical form in a similar deterministic manner. If the logical form is represented as a graph, it would not be hard to fix some traversal order. It would be more problematic, however, to develop a generation control routine which would generate in the order of this fixed traversal. Moreover, given that reentrancies may enter into the logical form, it would be difficult to stop the logical form from being further instantiated during the traversal. In short, it is much easier to traverse through a simple input list than it is to traverse through a complex logical form graph.

[38]For this type/token distinction in feature structures, see Shieber (1986). Although the feature structure logic of Kaspar and Rounds (1986) makes no type/token distinction for atomic values, this is by no means a necessary assumption. In HPSG (Pollard and Sag 1987), for example, the type/token distinction is made for atomic types, which correspond to atomic values in untyped approaches.

[39]Another alternative would be for *quickly* to be interpreted as a higher predicate.

5 Conclusion

Unification based grammars are inherently neutral with respect to processing direction. So, at least from a logical perspective, there is no problem with using these grammars for generation. The practical problem arises, however, when it is required that the generator function with reasonable efficiency. As we have argued in this paper, this requires that the generator be able to use both top-down and bottom-up information. This desideratum can be met either by using a left/head-corner type of routine or by using a bottom-up approach such as Earley's algorithm, which includes top-down filtering. Since subcategorization information in lexically-driven linguistic theories is encoded in the lexical head of a phrase, algorithms for generation need to adopt a head-driven strategy. Both left-corner and Earley-style generation algorithms can be modified in this manner. However, as we have shown in Section 3.1, the efficiency of head-driven strategies is jeopardized when they encounter heads whose semantics is not sufficiently specified, as is the case with traces, certain type-raised NPs and semantically vacuous head constituents such as case-marking prepositions. At this point, it remains unclear whether such cases are best handled by suspending the head-driven strategy under certain circumstances or by imposing constraints on the grammar. Furthermore, as discussed in Section 4, generation algorithms for unification grammars, regardless of whether they adopt a head-driven strategy or not, are subject to the problems completeness and coherence, in the sense of Wedekind (1988). That is, these algorithms may generate strings which encode somewhat more or somewhat less information than is contained in the goal semantics. We have shown that both of these problems can be solved by eliminating the confusion between metalanguage and object language variables in logical forms.

References

Appelt, D. E. 1987. Bidirectional grammars and the design of natural language generation systems. In *Theoretical Issues in Natural Language Processing-3*, 185–191.

Bates, M. 1978. The Theory and Practice of Augmented Transition Network. In *Natural Language Communication with Computers* 151–259. Berlin: Springer.

Bouma, G. 1987. A Unification-based analysis of unbounded dependencies in categorial grammar. In *Proceedings of the Sixth Amsterdam Colloquium*, ed. J. Groenendijk, M. Stokhof, and F. Veltman.

Calder, J., M. Reape and H. Zeevat. 1989. An Algorithm for Generation in Unification Categorial Grammar. In *Proceedings of the Fourth Conference of the European ACL*, 233–240.

Dymetman, M. and P. 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, PA, Carnegie-Mellon University.

Earley, J. 1970. An Efficient Context-free Parsing Algorithm. *Communications of the ACM* 13(2):94–102. Also in Grosz et al. 1986.

Gazdar, G., E. Klein, G. Pullum and I. Sag. 1985. *Generalized Phrase Structure Grammar.* Cambridge, MA: Harvard University Press.

Gazdar, G. and C. Mellish. 1989. *Natural Language Processing in Prolog/Lisp/Pop 11.* Addison-Wesley. These are different books, one for each language.

Gerdemann, D. 1989. Using Restriction to Optimize Parsing. In *Proceedings of International Workshop on Parsing Technologies*, 8–17.

Gerdemann, D. 1991. *Parsing and Generation of Unification Grammars.* PhD thesis, University of Illinois. Published as Beckman Institute Cognitive Science Technical Report CS-91-06.

Gerdemann, D. and E. W. Hinrichs. 1990. Functor-Driven Natural Language Generation with Categorial-Unification Grammar. In *COLING-90, vol 2,* ed. H. Karlgren, 145–150.

Grosz, B., K. S. Jones and B. L. Webber (ed.). 1986. *Readings in Natural Language Processing.* Los Altos, CA: Morgan Kaufmann.

Haider, H. 1986. V-Second in German. In *Verb Second Phenomena in Germanic Languages*, Vol. 21, 49–86. Dordrecht: Foris.

Haider, H. and M. Prinzhorn (ed.). 1986. *Verb Second Phenomena in Germanic Languages*, Vol. 21 of Publications in Languages Sciences. Dordrecht: Foris.

Hendrix, H. 1987. Type change in Semantics: The scope of quantification and coordination. In Ewan Klein and Johan van benthem (eds.), *Categories, Polymorphism and Unification*, 95–120. Edinburgh/Amsterdam: CCS/ILLI.

Hinrichs, E. and T. Nakazawa. 1989a. Flipped Out in AUX in German. In *Papers from the 25th Regional Meeting of the Chicago Linguistic Society*, 193–202.

Hinrichs, E. and T. Nakazawa. 1989b. Subcategorization and VP Structure. Talk presented at the *Third Symposium on Germanic Linguistics*, (published as "Aspects of German VP Structure." SFS-Report-01, Seminar für sprachwissenschaft, Eberhard-Karls-Universität, Tübingen, Germany, 1993).

Illingworth, V. (ed.). 1990. *Dictionary of Computing.* Oxford University Press. Third Edition.

Kaplan, R. 1973. A General Syntactic Processor. In *Natural Language Processing*, ed. R. Rustin, 193–241. New York: Algorithmics Press.

Kaplan, R. and J. Bresnan. 1982. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, ed. J. Bresnan, 171–281. Cambridge, MA: MIT Press.

Kaspar, R. and W. Rounds. 1986. A Logical Semantics for Feature Structures. In *ACL Proceedings, 24th Annual Meeting*, 257–271.

Kay, M. 1973. The MIND System. In *Natural Language Processing*, ed. R. Rustin, 155–188. New York: Algorithmics Press.

Kay, M. 1979. Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, 17–19.

Kay, M. 1989. Head-Driven Parsing. In *Proceedings of International Workshop on Parsing Technologies*, 52–62.

Larson, R. 1985. Bare-NP Adverbs. *Linguistic Inquiry* 16:595–621.

Matsumoto, Y., H. Hirakawa, H. Miyoshi and H. Yasukawa. 1983. BUP: A Bottom-Up Parser Embedded in Prolog. *New Generation Computing* 1(2):145–158.

McCawley, J. 1988. Adverbial NPs: Bare or clad in see-through garb? *Language* 64:583–590.

Montague, R. 1973. The Proper Treatment of Quantification in Ordinary English. In *Approaches to Natural Language.* Dordrecht: Reidel.

Moortgat, M. 1988. *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus.* Dordrecht: Foris Publications.

Moortgat, M. 1990. Discontinuous type constructors. Workshop Categorial Grammar and Linear Logic, Leuven.

Nerbonne, J., M. Iida and W. Ladusaw. 1989. Running on Empty. In *Proceedings of the Eighth West Coast Conference on Formal Linguistics*, ed. E. J. Fee and K. Hunt, 276–288.

Partee, B. and M. Rooth. 1983. Generalized Conjunction and Type Ambiguity. In *Meaning, Use and Interpretation of Language*, ed. R. Bauerle, C. Schwarze and A. von Stechow, 361–383. Walter de Gruyter.

Pereira, F. and D. Warren. 1983. Parsing as Deduction. In *ACL Proceedings, 21st Annual Meeting*, 137–144.

Pereira, F. and S. Shieber. 1987. *Prolog and Natural Language Analysis.* CSLI Lecture Notes No. 10. Chicago: Chicago University Press.

Pollard, C. 1990. On Head Non-Movement. In *Proceedings of the Tilburg Conference on Discontinuous Constituents.*

Pollard, C. and I. Sag. 1987. *An Information-Based Approach to Syntax and Semantics: Volume 1 Fundamentals.* CSLI Lecture Notes No. 13. Chicago: Chicago University Press.

Rosenkrantz, D. and P. Lewis. 1970. Deterministic left corner parser. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, 139–152.

Ross, P. 1989. *Advanced Prolog.* Reading MA: Addison-Wesley.

Russell, G., S. Warwick and J. Carroll. 1990. Asymmetry in Parsing and Generation with Unification Grammars: Case Studies from ELU. In *ACL Proceedings, 28th Annual Meeting*, 205–211.

Sag, I. 1976. *Deletion and Logical Form.* PhD thesis, MIT.

Shieber, S. 1985. Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. In *ACL Proceedings, 23rd Annual Meeting*, 145–152.

Shieber, S. 1986. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes No. 4. Chicago: Chicago University Press.

Shieber, S. 1988. A Uniform Architecture for Parsing and Generation. In *COLING-88*, 614–619.

Shieber, S. 1989. *Parsing and Type Inference for Natural and Computer Languages*. PhD thesis, Stanford University. Published as SRI International Technical Note 460.

Shieber, S., G. van Noord, R. C. Moore and F. Pereira. 1990. Semantic-Head-Driven-Generation. *Computational Linguistics* 16:30–42.

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

Thiersch, C. 1978. *Topics in German Syntax*. PhD thesis, MIT.

Uszkoreit, H. 1985. *Word Order and Constituent Structure*. CSLI Lecture Notes No. 8. Chicago: Chicago University Press.

Uszkoreit, H. 1986. Categorial Unification Grammar. In *COLING-86*, 187-194.

van der Linden, E.-J. and G. Minnen. 1990. Algorithms for Generation in Lambek Theorem Proving. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, 220–226. Pittsburgh.

van Noord, G. 1991. Head Corner Parsing for Discontinuous Constituency. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, 114–121.

von Stechow, A. and W. Sternefeld. 1988. *Bausteine Syntaktischen Wissens*. Opladen: Westdeutscher Verlag.

Wall, R. and K. Wittenburg. 1989. Predictive Normal Forms for Composition in Categorial Grammars. In *Proceedings of International Workshop on Parsing Technologies*, 152–161.

Wedekind, J. 1988. Generation as Structure Driven Derivations. In *COLING-88*, 732–737.

Wittenburg, K. 1987. Predictive Combinators: A Method for Efficient Parsing of Combinatory Categorial Grammars. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 73–80.

Woods, W. A. 1970. Transition network grammars for natural language analysis. *Communications of the ACM* 13:591–596.

Zeevat, H. 1988. Combining categorial grammar and unification. In *Natural language parsing and linguistic theories*, ed. U. Reyle and C. Rohrer. Dordrecht: Reidel.