AN ALGORITHM FOR GENERATION IN UNIFICATION CATEGORIAL GRAMMAR

Jonathan Calder, Mike Reape and Henk Zeevat
University of Edinburgh
Centre for Cognitive Science
2 Buccleuch Place
Edinburgh
EH8 9LW

Abstract

We present an algorithm for the generation of sentences from the semantic representations of Unification Categorial Grammar. We discuss a variant of Shieber's *semantic monotonicity* requirement and its utility in our algorithm. We indicate how the algorithm may be extended to other grammars obeying the same requirement. Appendices contain a full listing of the program and a trace of execution of the algorithm.

1. Introduction

In this paper we present an algorithm for generating sentences using unification categorial grammars (UCGs, Zeevat et al. 1987) but which extends to any categorial grammar with unification (e.g., categorial unification grammars, Uszkoreit 1986, Karttunen 1987). We relate the algorithm to proposals by Shieber (1988). Following Shieber, we address the basic generation problem; that is, given a syntactic category K and a semantic representation Φ, generate every possible string defined by the grammar of category K with a semantic representation that is logically equivalent to Φ. In more concrete terms, this means that we dispense with any planning component and directly address the intrinsic complexity of the basic generation problem. The development of such algorithms is as fundamentally important as the corresponding work on parsing algorithms.

We also discuss the properties of a semantic representation language (SRL) and the manner of its construction which makes our algorithm effective. The crucial property is a stricter form of Shieber's (1988) property of *semantic monotonicity*. We not only require that the semantics introduced by all subconstituents of an expression appear within the semantics of the expression as a whole; we also require that the semantics of any containing expression be a further instantiation of one of its subexpressions.

We introduce the algorithm on a case-by-case basis, at each stage extending its coverage and include a listing of the program implementing this algorithm, as appendix A.

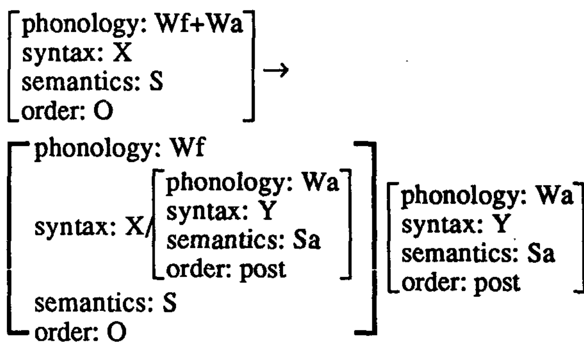
2. Basis of the algorithm

The most important feature of categorial grammars is the close correspondence of syntactic and semantic functors. In generation, if the semantic functor of an expression can be identified, possible values of the syntactic functor can also be determined. Under these circumstances, a simple recursive procedure can be stated which implements a mixed top-down and bottom-up strategy, alternately determining the functor of an expression and generating the arguments to that functor. In the presentation of the basic algorithm below we will make the simplifying assumption that for any formula of the semantic representation language, the syntactic and semantic functors are immediately identifiable. We will have to relax this restriction in order to deal with phenomena such as type raising and identity semantics.

UCG employs only two types of phrase structure rules. First, there are two *binary* rules of forward and backward application. Schematically, these can be represented as follows.

Result → Functor/Active   Active
Result → Active   Functor\Active

The first of the actual rules is stated below. The second is just like the first except that **pre** is substituted for **post** and the order of the daughters is reversed. Notice the use of the *order* feature. If a sign is an argument, then its order value is **pre** (**post**) if it precedes (follows) its functor.



Second, UCG employs a small set of unary rules of the form $\alpha \rightarrow \beta$ where α and β are UCG signs. Unary rules have several uses. These include the treatment of unbounded dependencies, syntactic forms of type-raising (e.g., generic noun to np rules) and subcategorization for optional modifiers. In general, unary rules relate one category to another. In particular, unary rules can change the category of a functor. This will require a modification to the basic strategy we present below.

The language *InL* (Indexed Language) is a variant of Kamp's (1981) Discourse Representation Theory. Its most important properties are i) every expression has a privileged variable (its *index*) and ii) every variable is sorted, so indicating the ontological category of the object denoted by the variable. The only logical connectives are conjunction and implication. The semantics of an expression is constructed compositionally via unification. As discussed further below, the semantic representation of any sentence in UCG is simply a further instantiation of the semantics associated lexically with one element of the sentence.

3. A sketch of the algorithm

Below we present the basic algorithm which implements the informal description given above. We give the algorithm in Prolog for convenience because various refinements to the algorithm to be discussed below (e.g., the use of a chart) depend directly on the procedural aspects of Prolog's control strategy. This basic version of the algorithm requires that UCG signs be encoded as first order terms and that term unification is used. This includes both InL formulas and sorted indices. A graph encoding of signs and graph unification could be used but this would make the presentation of the basic ideas more complicated. Unary rules are not covered in this first approximation.

```
generate(Sign) :-
  path_value(semantics,Sign,InL),
  path_value(semantics,Sign0,InL),
  lexical(Sign0),
  reduce(Sign0,Sign).

reduce(Sign,Sign).
reduce(Sign0,Sign) :-
  path_value(syntax:active,Sign0,
                   Active),
  apply(Sign0,Active,Result),
  generate(Active),
  reduce(Result,Sign).
```

The predicate **path_value**(*Path,Sign,Value*) succeeds if the value of the path *Path* through the sign *Sign* is *Value*. **lexical**(*Sign*) succeeds if the sign *Sign* can be unified with a lexical entry. **apply**(*Functor,Active,Result*) implements the rules of forward and backward functional application as discussed above.

generate(*Sign*) generates a sign *Sign* with phonology Φ, syntactic category K and semantics Σ by creating a new sign *Sign0* with phonology Φ', syntactic category K' and semantics Σ, unifying the sign with a lexical entry and then *reducing Sign0* to *Sign* in a bottom-up fashion. Thus **generate** implements the top-down half of the control strategy by "predicting" the syntactic category of *Sign* on the basis of which lexical entries unify with it. The bottom-up reduction is necessary as it is not necessarily the case that $\Phi = \Phi'$ or that K = K'. In particular, unless Σ corresponds to a nonfunctor lexical entry, *Sign0* will be of the schematic form **X/Y** (i.e., a lexical functor).

reduce has two clauses. The first reduces a sign *Sign* to itself. The second reduces a sign *Sign0* to a sign *Sign* if *Sign0* is a functor **Functor/Active** which when applied to **Active** by one of the rules of functional application gives the result sign **Result**, the sign **Active** can be generated and **Result** can be reduced to *Sign*. A sample execution of the algorithm, using only the above clauses for the two predicates, is given in Appendix B.

There is a major deficiency in this algorithm. Unification is the only method used to test the logical equivalence of two semantic representations. This means that not even the axioms of commutativity or associativity are available for testing logical equivalence[1]. One of the

[1]Strictly speaking, we test for a very strict form of consistency. Two LFs are considered logically

consequences of this is that, given a semantic representation ϕ, it may not be possible to generate a sentence with semantic representation ϕ', where ϕ and ϕ' are logically equivalent. In fact, it may not be possible to generate any sentence even though there are sentences defined by the grammar which have semantic representations which are equivalent to ϕ. So, for example, an semantic representation which is produced by parsing a nontopicalised sentence cannot be used to generate a topicalised sentence.

Shieber (1988) claims that the problem of logical equivalence reduces to the *knowledge representation problem*. The claim is that there will be no full solution to this problem in the near future. To satisfy our definition of generation however, we must generate all sentences whose semantic representations are logically equivalent to the semantic representation being generated under the rules of inference and axioms of the semantic representation language. In the case of InL, the primary axioms are simply associativity and commutativity. However, these two axioms alone give the equivalence problem factorial complexity. We will discuss these issues below after we have introduced some refinements to the algorithm.

4. Refinements to the basic algorithm

The algorithm presented above is deficient in other respects. There are three other aspects of UCG analyses that are not covered. First, all NPs are *type-raised*. The standard UCG analysis of non-lexical NPs is adequately handled using the above definitions, as the resulting semantic structure contains information introduced by the determiner. On the other hand, a lexical NP such as *Harry* will bind a variable in the semantics of an expression indicating that the translation of *Harry* is a constant. However, no other semantic material will be introduced from which the need to generate a lexical NP could be inferred. This is remedied quite easily by adding the condition, to the second clause of **reduce** above, that the category of *Sign0* is not *np*, and by adding the following clause to **reduce**:

```
reduce(Sign0, Sign) :-
  path_value(category:active,
                  Sign0, Active),
  path_value(category, Active, np),
  path_value(semantics, Active,
                  Index),
  proper_name(Index),
  typeraise_np(Active,
                  TypeRaisedNP),
  apply(TypeRaisedNP, Sign0,
                  Mother),
  generate(TypeRaisedNP),
  reduce(Mother, Sign).
```

The most important part of the above definition is the restriction of the clause to the generation of elements which satisfy the predicate **proper_name**; we assume this test to be appropriately defined according to the semantic representation language used. In our case, it is a simple test for instantiation. The predicate **typeraise_np**(*Active, TypeRaisedNP*) relates a non-type-raised to a type-raised NP. Note that in the call to **generate**, we attempt to generate from the constructed type-raised NP. The reasons for this are that lexical NPs have to be type-raised prior to the lexical lookup in **generate** and that the argument to the type-raised NP is generated in exactly the same manner as other arguments.

Two further problems are the treatment of unary rules and functors with what Shieber (1988) calls *vestigial semantics*, which we prefer to call *identity semantics*. The latter identify the semantics of their argument with their own semantics. That is, they are semantically vacuous. Examples from English are complementisers and case-marking prepositions. Again, we add an additional clause to **reduce** which enumerates the set of relations that may hold between signs under unary rules and under functors with identity semantics, using the auxiliary predicate **transform**. The clause recursively invokes **reduce** as it may be the case that a unary rule or functor with identity semantics introduces further syntactic arguments.

```
reduce(Sign0, Sign) :-
  transform(Sign0, Sign1),
  reduce(Sign1, Sign).

transform(Daughter, Mother) :-
  unary_rule(Mother, Daughter).
transform(Sign0, Sign) :-
  path_value(category:active,
           Sign1, Sign0),
  identity(Sign1),
  apply(Sign1, Sign0, Sign).
```

identity enumerates those lexical entries whose semantics is the same as that of one of its arguments. Note that both of these clauses continue the basic bottom-up reduction strategy. Essentially, we must freely apply both identity semantics functors and unary rules to guarantee completeness of the algorithm. Given that we apply unary rules and identity functors freely, our algorithm will only terminate if the bottom-up closure of such elements

---

equivalent if their sorted indices are consistent but the rest of the formula is logically equivalent. We return to this point briefly below.

with respect to elements of the lexicon is finite. In other words, we require that the grammar adhere to the offline parsability constraint (Kapland and Bresnan 1982). If this condition does not hold, the algorithm will not terminate.

5. Optimizations of the algorithm

Given the fairly high degree of top-down control, it should be obvious that the generator will generate some subformulas of its input repeatedly as it explores the search space. The obvious solution is to use a *lemma table* or *chart* (as discussed by Pereira and Warren 1984 and Shieber 1988). Shieber (1988) states that to guarantee completeness in using a precomputed entry in the chart, the entry must subsume the formula being generated top-down. However, empirical tests have shown that a naive chart strategy results in the chart *never* being used at all. This is to be expected given the nature of generation; since most of the signs being generated top-down are very partial (often they will have only the semantics instantiated) and chart entries will be very complete (since most information is projected from the lexicon) it will almost never be the case that a top-down sign is subsumed by the chart.

The result is that we must either abandon the idea of using a chart[1] or else devise a strategy for its use which is complete, does not rely on the subsumption test and does not put too many entries in the chart We have followed the latter strategy. This technique depends crucially on avoiding any top-down instantiation of candidate chart entries and by guaranteeing bottom-up completeness of chart entries consistent with a restriction of the top-down sign. The nature of the restriction that we use depends on properties of the semantic representation language itself. In particular, the only use of variables in the language is in representing existentially quantified variables over individuals. Thus every appearance of a variable can only be further instantiated by a semantic individual constant and so the semantic representation after generation cannot be further instantiated in such a way that the denotation of that expression differs from that of the input semantic representation.

The program presented in Appendix A illustrates the use of the chart. The reader will notice that the instruction to add information to the chart follows calls to **generate** but precedes calls to **reduce**. This strategy means that we keep the chart free of the top-down instantiations caused by equating a bottom-up solution (the first argument of **reduce**) to a top-down goal (the second argument of **reduce**).

Another method for reducing the search space is to use the technique of *freezing* in cases where the premature instantiation of variables will lead to avoidable backtracking. In the case of our current UCG grammar, it is often the case that the *order* feature is not instantiated when **apply** is called. If the argument is generated before the phonology is instantiated, then unnecessary generations with the wrong word order can be prevented. Therefore, we freeze the value of the *phonology* and *order* attributes until after an argument is generated. This requires some care to ensure that the freezing interacts with the chart strategy correctly. This is illustrated in the full program listing below. It is to be expected that more complex grammars would benefit from an extension of this technique to other attributes with mutually dependent values.

6. Extension to other grammatical formalisms

We alluded above to our assumption about the relationship between the semantics of lexical and non-lexical expressions. To recap, any semantic representation is a further instantiation of the semantic representation of some lexical item. This assumption will not hold for any grammar in which semantic material is introduced by rule (i.e. syncategorematically). The reason for this should be obvious given the definition of **generate** above. If a particular semantic representation possibly contains semantic structure not present in the lexicon, then any attempt to find an appropriate lexical functor on the basis of the semantics of an expression is not guaranteed to succeed. Relaxing this assumption would effectively remove all top-down predictive capacity for generation. The only solution in the context of this algorithm would then be to allow top-down application of all rules and to delay calls to lexical lookup until after rule application. This generate and test strategy is not only likely to be inefficient, it will also result in non-termination for many grammars.

In contrast, for grammars which do adhere to our assumption, our algorithm is effective, even if rules other than simple binary and unary rules are used. To see this, consider the following extension to **reduce**:

---

[1] A recent implementation of a similar algorithm by Thierry Guillotin and Agnes Plainfossé (Personal communication) suggests that the top-down application of unary rules, while making it impossible to guarantee completeness if making use of a chart, nevertheless leads to an overall improvement in efficiency by limiting the search space engendered by unary rules. This supports the contention that unary rule application is the dominant cost in generation with UCG.

```
reduce(Sign0, Sign) :-
  rule(Mom, Sign0, Kids),
  generate_sisters(Kids),
  reduce(Mom, Sign).
```

Note that this clause is very similar in structure to the second clause for **reduce**, the main difference being that the new clause makes fewer assumptions about the feature structures being manipulated. **rule** enumerates rules of the grammar, its first argument representing the mother constituent, its second the head daughter and its third a list of non-head daughters which are to be recursively generated by the predicate **generate_sisters**. (We assume, as with UCG, that information indicating the resulting phonology and order of constituents is contained within the feature structures of the rule). The behaviour of this clause is just like that of the clause for **reduce** which implements the UCG rules of function application. On the basis of the generated lexical sign *Sign0* an application of the rule is hypothesised and we then attempt to prove that that rule application will lead to a new sign *Mom* which reduces to the original goal *Sign*.

The same conditions apply to the generalized form of the predicate as to the clause for unary rules, namely the algorithm will terminate if the bottom-up closure of the rules of the grammar is finite. We conjecture that this algorithm extends naturally to the rules of composition, division and permutation of Combinatory Categorial Grammar (Steedman 1987) and the Lambek calculus (1958).

7. Implementation

The algorithm discussed in this paper has been implemented in C-Prolog. Recent work has looked at generation from semantic representations which are not in canonical format but which are equivalent, under the axioms of associativity and commutativity to the canonical semantics of sentences recognised by the grammar. Our effort is directed at formulating appropriate notions of "semicanonicality", which lessen the strict (and in many cases unobtainable) requirement that the representation to be generated from is identical to that obtained as the result of parsing. Such notions would increase the utility of generators such as we have presented while avoiding the dangers of factorial complexity.

A further source of inefficiency is the naive lexical indexing strategy used by the predicate **lexical**. We have presented the algorithm as if **lexical** simply enumerates the lexicon. This is clearly inefficient and some form of indexing strategy seems essential. The simplest is to choose the principal functor of the semantic representation to use as the index for lexical entries which have the same principal functor in their semantics. Much of the time however, the principal functor is simply the conjunction operator. A more sophisticated indexing strategy involves calculating the best (set of) key(s) to identify candidate lexical entries. This necessarily involves considerable complexity itself. Furthermore, if such indexing is to be automatic, very sophisticated compilation techniques and metaknowledge about the possible structure of semantic representations are required. We are also investigating these possibilities.

Acknowledgements

The work reported here is supported by ESPRIT project P393 ACORD: The Construction and Interroagtion of Knowledge Bases using Natural Language text and Graphics. Thanks are due to Philippe Alcouffe, Lee Fedder, Thierry Guillotin, Dieter Kohl and Agnes Plainfossé for discussions of problems in generation with UCG. All errors are of course our own.

References

Kaplan R.. M. and Bresnan J. (1982) Lexical-Functional Grammar: a formal system for grammatical representation, Chapter 4 in J. Bresnan (ed.) *The Mental Representation of Grammatical Relations*, 173-281, MIT Press, Cambridge Mass.

Karttunen, L. (1986) Radical Lexicalism. Report No. CSLI-86-68, Center for the Study of Language and Information, December, 1986. Paper presented at the Conference on Alternative Conceptions of Phrase Structure, July 1986, New York.

Lambek, J. (1958) The mathematics of sentence structure. *American Mathematical Monthly*, **65**, 154-170.

Pereira, F. C. and Warren, D. H. (1983) Parsing as Deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Massachusetts Institute of Technology, Cambridge, Mass., June, 1983, 137-144.

Shieber, S. (1988) A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, Budapest, 22-27 August, 1988, 614-619.

Steedman, M. J. (1987) Combinatory Grammars and Parasitic Gaps. *Natural Language and Linguistic Theory*, **5**, 403-439.

Uszkoreit, H. (1986) Categorial Unification Grammars. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computatinoal Linguistics*, Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, 25-29 August, 1986, 187-194.

Zeevat H., Klein, E. and Calder, J. (1987) An Introduction to Unification Categorial Grammar. In Haddock, N.J., Klein, E. and Morril, G. (eds.) *Edinburgh Working Papers in Cognitive Science*, Volume 1: *Categorial Grammar, Unification Grammar and Parsing.*

Appendix A: program listing

This listing contains all code discussed in the text for generation with UCG and includes a correct treatment of the chart. The second argument to generate is not discussed above: its function is simply to disable the check that determines whether to add information to the chart when that information has just been retrieved from the chart.

```
/* generate/2 */
generate(Sign, chart) :-
   verify(unifies_with_chart(Sign)), !,
   unifies_with_chart(Sign).
generate(Sign, nonchart) :-
   path_value(semantics, Sign, Sem),
   path_value(semantics, Sign0, Sem),
   lexicon(Sign0),
   reduce(Sign0, Sign).

/* reduce/2 */
reduce(Sign, Sign).
reduce(Sign0, Sign) :-
   path_value(category:active, Sign0,
                        Active),
   path_value(category, Active, np),
   path_value(semantics, Active, Index),
   proper_name(Index),
   typeraise_np(Active, TypeRaisedNP),
   apply(TypeRaisedNP, Sign0, Mother, [],
                        Freezer),
   generate(TypeRaisedNP, Chart),
   unfreeze(Freezer,[]),
   add_to_chart(TypeRaisedNP, Chart),
   reduce(Mother, Sign).
reduce(Sign0, Sign) :-
   path_value(category:active, Sign0,
                        Active),
   path_value(category, Active, Category),
   not Category = np, not Category = pp,
   apply(Sign0, Active, Mother, [],
                        Freezer),
   generate(Active, Chart),
   unfreeze(Freezer,[]),
   add_to_chart(Active, Chart),
   reduce(Mother, Sign).
reduce(Sign0, Sign) :-
   transform(Sign0, Sign1, [], Freezer),
   unfreeze(Freezer,[]),
   add_to_chart(Sign1, nonchart),
   reduce(Sign1, Sign).

/*transform/4 */
transform(Daughter, Mother, Freezer,
                        Freezer) :-
   unary_rule(Mother,[Daughter]).
transform(Sign0, Sign, Freezer0, Freezer)
                        :-
   path_value(category:active, Sign1,
                        Sign0),
   identity(Sign1),
   apply(Sign1, Sign0, Sign, Freezer0,
                        Freezer).

/* apply/5 */
apply(S1, S2, S3, F0,
   [freeze(Order2,Phonology,W1,W2)|F0]):-
   S1 = sign(W1,Cat1/S2,Sem1,Order),
   S2 = sign(W2,Cat2,Sem2,Order2),
   S3 = sign(Phonology,Cat1,Sem1,Order).

/* typeraise_np/2 */
typeraise_np(Sign0,Sign) :-
   Sign0 = sign(_,np,_,_),
   Sign = sign(_,Cat/
               sign(_,Cat/
                      sign(_,_,_,Order),
               Sem0,Order),
         Sem,_),
   Sign = sign(_,
               _/sign(_,_/Sign0,_,_),
               _,_).

/* proper_name/1 */
proper_name(N) :- nonvar(N).


/* unifies_with_chart/1 */
unifies_with_chart(S) :-
   chart(S).

/* add_to_chart/2 */
add_to_chart(S, nonchart) :-
   verify(unifies_with_chart(S)), !.
add_to_chart(S, nonchart) :-
   assertz(chart(S)).
add_to_chart(_, chart).
```

```
/* unfreeze/2 */
unfreeze([], []).
unfreeze([freeze(pre,W1+W2,W1,W2)|R],F):-
      unfreeze(R,F).
unfreezer([freeze(post,W2+W1,W1,W2)|R],F):-
      unfreeze(R,F).

/* verify/1 */
verify(Goal) :- \+ \+(Goal).
```

Appendix B: A trace of program execution

In this example, we use only the first two clauses of **reduce/2** above. Figure 1 gives a graphical representation of the information flow during generation. **reduce(1)** indicates a use of the first, base clause, and **reduce(2)** a use of the second. Circled numbers in the figure refer to the subsequent attribute value structures. We omit (8) and (13) as the corresponding feature structures are easily determined by inspection, corresponding to the base clause of **reduce/2**.

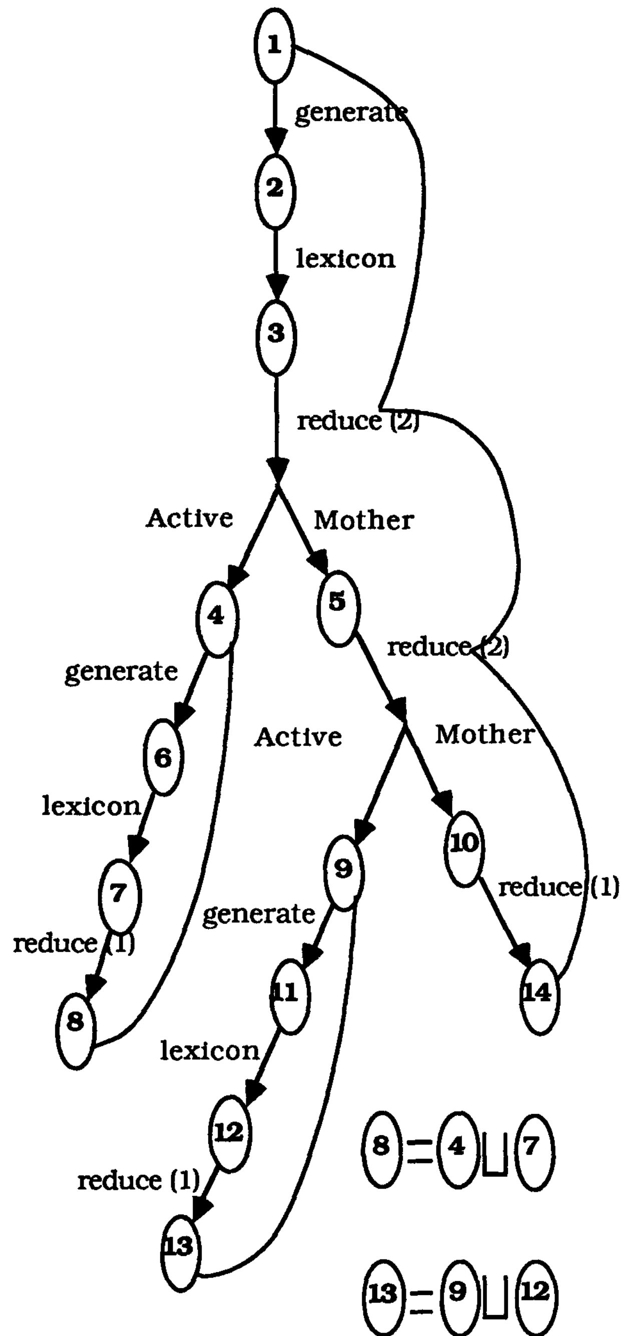


Figure 1: A trace of execution for the sentence *Every boy dreams*

(1)
$$\begin{bmatrix} \text{phon: W} \\ \text{cat: sent} \\ \text{sem: s:imp:[m:boy:[],e:dream:[m]]} \\ \text{order: Order} \end{bmatrix}$$

```
(2)  [ phon: W
       cat: Cat
       sem: s:imp:[m:boy:[],e:dream:[m]]
       order: Order ]

(3)  [ phon: every
       cat: Cat/[ phon: Wf
                  cat: Cat/[ phon: Wa
                             cat: np
                             sem: m
                             order: Oa ]
                  sem: e:dream:[m]
                  order: Oa ] /X
       sem: s:imp:[m:boy:[],e:dream:[m]]
       order: Of ]

where X = [ phon: Wn
            cat: noun
            sem: m:boy:[]
            order: pre ]

(4)  [ phon: Wn
       cat: noun
       sem: m:boy:[]
       order: pre ]

(5)  [ phon: every+Wn
       cat: Cat/[ phon: Wf
                  cat: Cat/[ phon: Wa
                             cat: np
                             sem: m
                             order: Oa ]
                  sem: e:dream:[m]
                  order: Oa ]
       sem: s:imp:[m:boy:[],e:dream:[m]]
       order: Of ]

(6)  [ phon: Wn
       cat: Cat
       sem: m:boy:[]
       order: Order ]

(7)  [ phon: boy
       cat: noun
       sem: m:boy:[]
       order: Order ]

(9)  [ phon: Wf
       cat: Cat/[ phon: Wa
                  cat: np
                  sem: m
                  order: Oa ]
       sem: e:dream:[m]
       order: Oa ]

(10) [ phon: every+boy+Wf
       cat: Cat
       sem: s:imp:[m:boy:[],e:dream:[m]]
       order: Of ]

(11) [ phon: Wf
       cat: Cat
       sem: e:dream:[m]
       order: Oa ]

(12) [ phon: dreams
       cat: sent/[ phon: Wa
                   cat: np
                   sem: m
                   order: pre ]
       sem: e:dream:[m]
       order: Oa ]

(14) [ phon: every+boy+dreams
       cat: sent
       sem: s:imp:[m:boy:[],e:dream:[m]]
       order: Of ]
```