BUG: A Directed Bottom Up Generator for Unification Based Formalisms.

Gertjan van Noord
Department of Linguistics RUU
Trans 10
3512 JK UTRECHT

March 17, 1989

Introduction

Lately there has been some interest in generators that derive strings from feature structures within unification based formalisms (e.g. Shieber 1988 for PATR II, and also references cited there, Wedekind 1988 for an LFG generator, and Dymetman & Isabelle 1988 for generation with DCG's). In the first section I will show that the generators that have been proposed all face some problems. Some analyses in the spirit of Unification Categorial Grammar (Zeevat et al. 1987) and HPSG (Pollard & Sag 1987) that account for Dutch crossing dependencies cannot be handled by top-down generators. On the other hand, Shieber's bottom up generator for PATR II requires grammars to be *semantically monotonic.* I will assume that this requirement is too strong for general usage. As an alternative I will define BUG, a bottom up generator that uses a top-down *oracle* , as is common in directed bottom up parse strategies (for an overview cf. Kay 1980, see also the BUP parser in Matsumoto et al. 1983, and a similar parser in chapter 6.3 of Pereira & Shieber 1987). I require that all grammar rules have exactly one semantic head. Furthermore a restriction on semantic heads will be defined. In the third section I will compare this restriction with Shieber's requirement on semantic monotonicity. It will be shown that the requirement on semantic heads allows some useful analyses not available in Shieber's approach.

BUG is part of an experimental machine translation system for translating international news items of Teletext. The analysis/generation module of the system uses a Prolog version of PATR II (similar to Hirsch 1987). For this reason I will define BUG within this Prolog based PATR II environment, but I assume that the approach is valid for unification based formalisms in general.

1 Problems with existing generators

Currently available generation algorithms face a number of problems. First I will describe the principal problem for top-down generators. Then I will discuss Shieber's bottom up generator.

1.1 A major problem for top-down generators: left recursion

In this section I will assume a general generation strategy, where rules are applied in a top-down fashion. The order in which nonterminals are expanded can be important. I will assume that this ordering is defined in a satisfactory way. Dymetman & Isabelle 1988 propose a version of DCG where the order of the goals of the body of a DCG clause is defined in two ways: one for parsing and one for generation. Furthermore some version of goal freezing (Colmerauer 1982) is used in cases where this scheme is not powerful enough. The LFG generator of Wedekind 1988 checks whether the resulting daughters of a rule are *connected*, before this rule can be applied. A node is connected iff its semantics is instantiated. Connectedness not only influences the order of expansion of nonterminals, but also influences the order in which rules are selected. This constraint therefore excludes some useful analyses, as will be shown in the following. Top-down strategies where rules are applied nondeterministically will not terminate for these analyses.

To introduce the problem for top-down generators grammar rule <1> will be considered first. This rule shows that generation is undecidable in general [1]. The rule states that a VP can be combined with the lexical entry 'to'. The result is a VP that has the same feature structure. This example (albeit unrealistic) shows that for some type of grammars the generation process is undecidable because for every VP we can add the word 'to' as often as we want (e.g. generating strings like "john likes to to to to kiss mary"). In most natural language grammars syntactic information will be available to stop this source of infinite ambiguity. Often however this information is only available after lexical lookup. Therefore a top-down generator may fail to terminate for some grammars, although syntactic information is available that should have blocked this type of undecidability. I will discuss an example within PATR II, in which an analysis of subcategorization in HPSG style is used. The crucial rule is given in <2>, a complete grammar can be found in appendix A (sample grammar 3) in Shieber 1986. In this analysis subcategorization is expressed by associating each word in the lexicon with a list of arguments (the 'subcat' feature in the example). These arguments are selected one at a time with rules like <2>. Notice that there is no upper limit to the length of this list. Syntactic information blocks infinite ambiguities, because the subcat list will have some fixed length. This length is specified for each word in the lexicon. Top-down generators cannot use this syntactic information because this information is rooted in the lexicon. The lexicon for example specifies that the subcat list of the verb 'kiss' has two members. However, this information is not available until the lexical entry 'kiss' is reached. The generation process will also predict longer subcat lists, without the possibility to know that these lists will never be realized in a lexical entry. At a certain moment in the generaton process rule <2> can be

```
<1> VP_1 -> to VP_2
    <VP_1> = <VP_2>
```

```
<2> VP_1 -> VP_2 X :
    <VP_1 head> = <VP_2 head>
    <VP_2 subcat first> = <X>
    <VP_2 subcat rest> = <VP_1 subcat>
```

[1] Note that I use the well known PATR II notation for grammar rules throughout this paper

applied to a feature structure that can be unified with feature structure VP_1. As a result, rule <2> can also be applied to VP_2, and so on. For the LFG-generator of Wedekind this analysis is impossible; in his approach rule <2> can never be applied because node X will be unconnected. For the DCG-generator of Dymetman & Isabelle the generation process will indeed not terminate. The problem displays some similarities with the problem of left recursion for top-down parsing strategies; therefore I will call this problem the left recursion problem for top-down generation [2] .
Note that the problem arises because there is no limit to the size of the subcategorization list. Although one might propose an upper limit for lexical entries, it is linguistically wrong to pose an upper limit in general because subcat lists may be appended in syntax resulting in indefinitely long subcat lists, e.g. in analyses of Dutch crossing dependencies phenomena (Evers 1975, Huybrechts 1984), but also in analyses of Verb Clusters in other languages (Evers 1987). Consider the Dutch sentence <3a>. In this construction verbs can combine their subcat lists, as can be seen from <3b>. Therefore subcat lists can have any length. It is also impossible to predict from a semantic structure that contains e.g. 'kiss' the size of its subcategorization list by looking in the lexicon directly. It can be the case that the verb will be combined with another verb, resulting in a longer subcategorization list.

1.2 Bottom up generators

The only bottom up generator I currently know of is the one proposed in Shieber 1988. In Shieber's generator rules are applied in a

```
<3a> dat [Jan [Marie [de oppasser [de
     olifanten [zag helpen voeren]]]]]
     that John Mary the keeper the
     elephants saw help feed
     (that John saw  Mary help the
     keeper feed the elephants)
```

```
<3b>           V [a,b,c,d]

           V [c,d]    V [a,b,c]

                  V [b,c]     V [a,b]

           zag    helpen      voeren
           saw    help        feed

     a = elephants
     b = keeper
     c = Mary
     d = John
```

bottom up fashion. Results are kept on an Earley type chart. To make this process goal driven there is a restriction that the semantics of every subphrase that is found must subsume some part of the original semantics (the input). This restriction results in the *semantic monotonicity* requirement on grammars; this restriction requires that the semantics of a part of a derivation must subsume part of the semantics of the complete derivation. An example will clarify the strategy. Assume we want to generate a string for the semantic formula <4> with a grammar comparable to sample grammar 3 of Shieber 1986 [3] . As the generator starts it will try to select

```
<4> kiss(john,mary)
```

[2] It is not necessarily so that this left recursion is caused by the leftmost daughter of a rule; the left recursion is caused by that daughter that is generated first

[3] In this paper I will abbreviate semantic forms. In fact the semantics is a feature structure as every other.

rules (and lexical entries). First it can select lexical entries that subsume part of <4>. The lexical entries 'john', 'mary', 'kiss' will be put on the chart as they subsume part of <4>. A NP can be constructed dominating 'john' and another one dominating 'mary'. A VP dominating 'kiss' will be constructed as well. Now a rule similar to <2> can apply, resulting in a VP with the semantics as in <5>. At last a

```
<5> kiss(_,mary)
```

rule applies that combines the NP dominating 'john' and the VP dominating 'kiss' and 'mary' resulting in the sentence 'john kisses mary' (ignoring inflection for the moment) with semantics <4>. Note that no other rules can apply if their resulting semantics does not subsume part of the original semantics. In the foregoing example it is e.g. not possible to apply a rule to construct a PP 'to mary' or anything similar to that.

The requirement that every rule application yields a semantics that subsumes part of the input only results in a complete generator if the grammar is semantically monotonic. Shieber himself admits that this requirement is too strong (op. cit. section 7):

> "Perhaps the most immediate problem raised by the methodology for generation introduced in this paper is the strong requirement of semantic monotonicity. (...) Finding a weaker constraint on grammars that still allows efficient processing is thus an important research objective."

For the moment I will assume that semantic monotonicity is too strong. In the following section I will introduce an alternative generator, BUG, that uses a different constraint on grammars. In section 3 I will discuss some analyses of prepositional verbs and idioms that are not semantically monotonic but can be handled by BUG. Furthermore I will give some evidence that suggests that BUG "allows efficient programming".

2 BUG: a Directed Bottom Up Generator

In this section I will introduce BUG, a directed bottom up generator, and I will discuss some of its properties. The task for the generator can be seen as follows. The generator will receive a feature structure Top as its input. Part of this feature structure is the semantic structure S_Top. For a given grammar, the generator derives all strings W for which the condition <6> holds (for a formal definition of this concept within LFG cf. Wedekind 1988). As will become clear in the following para-

<6> Completeness and coherency

> A feature structure Top with semantics S_Top derives string W w.r.t. grammar G iff W is a terminal string admitted by G with feature structure Top2 and semantics S_Top2, Top can be unified with Top2, S_Top subsumes S_Top2 *(coherency)* and S_Top2 subsumes S_Top *(completeness)*.

graphs, a few conditions on grammars apply for BUG to work properly. For each grammar rule there is one daughter that is the semantic head (specified as such by the rule writer). For the moment I will assume that if the semantics of the mother node is instantiated then the semantics of the semantic head is instantiated too. I will define the notion 'semantic head' more precisely in section 2.3. First I will introduce the basic structure of BUG.

2.1 The basic structure of BUG

The basic structure of BUG that will be explained in a moment can be summarized by the set of Prolog clauses <7>. This simple

```
<7> /* BUG version 1 */
    generate(Top,P0-PN):-
        link(Top,Down),
        word(Word,Down),
        up(Down,Top,[Word|PI]-PI,P0-PN).

    up(Top,Top,String,String).
    up(Down,Top,PI-PJ,P0-PN):-
        rule(Down,Mother,Lefts,Rights),
        link(Top,Mother),
        generate_list(Lefts,PH-PI),
        generate_list(Rights,PJ-PK),
        up(Mother,Top,PH-PK,P0-PN).

    generate_list([],P0-P0).
    generate_list([H|T],P0-PN):-
        generate(H,P0-P1),
        generate_list(T,P1-PN).
```

Prolog program, that bears a close connection to the lc parser described in Pereira & Shieber 1987, represents the basic structure of the generator I will develop in the remainder of this paper. It is assumed that the grammar consists of clauses 'rule(Head, Mother, Lefts, Rights)' as a result of compiling PATR rules into Prolog, where Head is the feature structure of the semantic head of this rule, Mother the feature structure of the mother node, Lefts is a list of feature structures that describe the daughters that precede the head and Rights the feature structures that are preceded by the head. The lexicon simply consists of clauses 'word(Word,F)' where Word is the terminal string and F its feature structure.

In the program the clause 'link' is used as a top-down oracle. A *basic* link between A and B exists if and only if there is a rule whose mother's semantics unifies with A's semantics and whose semantic head's semantics unifies with B's semantics. All possible basic links are precompiled. A *link* between A and B exists if there is a basic link between A and B or if there is a basic link between A and C, and a link between C and B. Usually the number of different basic links is very small. The link predicate is comparable to the reachability tables in parse theory (cf. Kay 1980) and the link predicate within BUP (Matsumoto et al. 1983).

Viewed procedurally, the generation process defined in <7> proceeds bottom-up from semantic head to semantic head. The surface string grows to the left and to the right. First a 'seed' of the semantics is computed by the 'link' predicate. From this seed the predicate 'up' builds a feature structure that can be unified with the original feature structure. A rule is selected if a seed unifies with the semantic head of that rule. Other daughters of this rule are then generated recursively; their resulting strings are concatenated to the left and to the right of the string dominated by the seed. The feature structure of the mother becomes the new seed. It dominates the modified string. Now the 'up' predicate is called with this new seed. Eventually a seed can be unified with the original feature structure, yielding the final string; this case is defined in the first clause of 'up'.

An example will clarify this strategy. Suppose we start with semantic structure <4>. Suppose moreover that we have the following (abbreviated) rules, where the H categories represent the semantic heads <8>. This simple grammar only allows one type of link: a link exists between feature structures that share their semantics. The generation process is started by a call to the generate predicate, where Top is a feature structure whose semantics is instantiated; the second argument of

```
<8a> S -> H X
     <S sem> = <H sem>
     <H subcat first> = <X>
     <H subcat rest> = nil

<8b> VP -> H X
     <VP sem> = <H sem>
     <VP subcat> = <H subcat rest>
     <H subcat first> = <X>

<8c> john
     <sem pred> = john

<8d> mary
     <sem pred> = mary

<8e> kisses
     <sem pred> = kiss
     <sem arg1> =
         <subcat rest first sem>
     <sem arg2> =
         <subcat first sem>
```

'generate' represents the string that is to be found in difference list notation. The predicate 'link' relates feature structures whose semantics are shared; therefore Down will be a feature structure with the same semantics as Top. The predicate 'word' tries to find a lexical item whose feature structure unifies with Down. In this example the lexical item 'kisses' is the only candidate. Now the syntax of Down will be instantiated as well. The predicate 'up' will now be called with the two feature structures Down and Top, and two strings in difference list notation. The first string represents the string that is found up to now (the word 'kisses'); the second string is the resulting string. The predicate 'up' will select a rule whose semantic head unifies with Down. In this example the rule <8b> is a possible candidate. A link is computed between the mother of this rule Mother and the feature structure Top. The predicate 'generate_list' generates a string for daughters of this rule to the left of the semantic head, and a string for daughters to the right of the head. The first call to 'generate_list' succeeds trivially as there are no daughters to the left of the head. The second call to 'generate_list' yields the string 'mary'. The resulting string 'kisses mary' will be the third argument to the recursive call to 'up'; the feature structures Mother and Top will be the first and second argument. In this next call to 'up' rule <8a> is a possible candidate. Now the first call to 'generate_list' will yield 'john', whereas the second succeeds trivially. Finally the predicate 'up' is called with the instantiation of the mother of <8a>, Top, and the string 'john kisses mary'. As the two feature structures can unify the recursion bottoms out by unifying the strings. The string 'john kisses mary' will therefore be the final result. Note that words, rules and links are selected nondeterministically; this nondeterminism is handled by Prolog's built-in backtrack mechanism.

2.2 Making BUG complete and coherent

The generator defined in <7> is not coherent and not complete (cf. <6>), because <7> derives feature structures whose semantics unifies with the semantics of the original feature structure. Note furthermore that generator <7> can easily fail to terminate because of this lack of coherency. An example of this is <9a>. <9a> may be the semantics for the

```
<9a> eat(john)
<9b> eat(john,banana)
<9c> eat(john,nice(yellow(banana)))
```

sentence 'john eats'. Generator <7> will also try to build all feature structures containing a second argument, e.g. <9b>. For a realistic grammar the number of ways to do this will be infinite (e.g. by adding modifiers, <9c>); therefore BUG will not terminate.

Coherency is achieved by 'freezing' all variables occuring in the semantics of the original feature structure (e.g. by the numbervars predicate built in in some implementations of Prolog). These 'atomic place holders' will not unify with any augmentations to the semantics. In order to do so BUG is redefined as in <10>.

```
<10> /* BUG version 2 */
     bug(Top,String):-
         sem(Top,Top_S),
         numbervars(Top_S,0,_),
         generate(Top,String).
```

This leaves us with the completeness problem. It is still possible that <10> derives strings like 'john eats' for semantics <9b>. The solution is to test at the end of the generator procedure whether the feature structure that is found is complete with respect to the original feature structure. However, because of the way in which top-down information is used, it is unclear what semantic information is derived by the rules themselves, and what semantic information is available because of unifications with the original semantics. For this reason so called 'shadow' variables are added in BUG that represent the feature structure derived by the grammar itself. Furthermore a copy of the semantics of the original feature structure is made at the start of the generation process. Completeness is achieved by testing whether the semantics of the shadow subsumes this copy. This technique is used in the final version of BUG (see the Appendix).

2.3 Semantic heads

The success of the generation strategy that is used by BUG very much depends on the definition of *semantic head* of a rule. The semantic heads are used to make it possible to compute links between a feature structure and a seed. Furthermore, by using these links, it is assumed that the semantics of other daughters of a rule get instantiated, to make a directed generation process possible. Therefore, the following condition <11> on semantic heads applies. The first part of this condition states

<11> Condition on rules - (a) and (b)

> For all rules R of grammar G with semantic head S, mother node M and daughter nodes Di
> (a) if the semantics of M is instantiated, then the semantics of S is instantiated (*connectedness of the semantic head*); and
> (b) if S is instantiated by a feature structure admitted by G, and the semantics of M is instantiated, then the semantics of Di are instantiated (*connectedness of the daughters*).

that the semantics of the semantic head of a rule must be predictable from the semantics of the mother node of this rule. The second part of <11> states that the semantics of the daughters of a rule must be predictable from the semantics of the mother and the syntax of the semantic head. The first part of the condition is a condition on rules, whereas the second part of the condition is a condition on grammars, because it is not clear what the syntax of the head can turn out to be, without looking at the complete grammar.

2.4 Auxiliaries and function words

A problem for BUG is posed by rules where a daughter (that is not the semantic head) has the same semantics as the mother node. An example is rule <12> In this type of rules,

```
<12> Sbar -> Comp  H :
     <Sbar semantics> =
          <H semantics>
     <Sbar semantics> =
          <Comp semantics>
```

that usually occur in the case of auxiliaries and function words, BUG will not terminate, because after the construction of H it will try to construct the Comp. However this process starts with the same goal as the construction of sbar. Therefore this same rule will apply again and again. I hypothesize that this only occurs in the case of function words, case markers and things like that. In this case the syntactic head often differs from the semantic head; moreover the syntactic head is a lexical entry. Therefore I add <11c> to <11>. BUG

<11> Condition on rules - (c)

> <c> A node Di is the syntactic head of R iff the semantics of Di and the semantics of M are equal.
> If a node D functions as the syntactic head, but not as the semantic head, then D must be nonbranching.

is modified so that it recognizes cases of syntactic heads. In that case only lexical lookup is possible, and BUG will terminate for grammars that obey <11> (except for grammars that use rules like <1>).

2.5 Empty semantic heads

An analysis that seems to be ruled out by <11b> is the analysis of verb second phenomena in several languages. Root sentences in German for example usually are analyzed as <12> (I do not really care about the labels of this tree here). In these cases the semantic

```
<12>         smax

       topic    sbar

             verb(i)   s

                subj     vp

                          v

                         e(i)
```

head seems to be a 'gap'. Therefore probably the instantiation of e.g. its subcat list will only be known after some relationship is established with the displaced verb. As a result <11b> is violated. However in the final version of BUG a solution to this problem is available. The 'generate' predicate in BUG will yield a list of lexical items instead of a list of words. To this list a small phonological frontend will be applied. This reduces a lot of uninteresting nondeterminism during the generation process (e.g. the choice between 'kisses', 'kiss', and 'kissed' is only made after the generation of the other lexical items). We can use this strategy to allow for empty semantic heads. During the generation process there is no distinction between gaps of semantic heads and their lexical realization yet (just as there is no difference between 'kisses' and 'kiss' yet). A possible phonological realization of a feature structure is the empty string, provided some features are specified locally (e.g. the value of a slash-like feature as in GPSG, Gazdar et al. 1985). In this strategy there is no difference between associating some inflected form of a verb (e.g. "kissed") with a feature structure containing the appropriate features (e.g. agreement and tense) and associating the empty string with a feature structure containing the appropriate features (e.g. slash). This strategy furthermore reduces a lot of ambiguity in the generation of main (verb second) and subordinate (verb final) clauses in languages like German and Dutch.

3 A comparison between BUG and Shieber's generator

In this part of my paper I will compare the generator presented by Shieber with BUG. The first difference is that Shieber's generator is part of a general architecture for parsing and generation. I have presented BUG as an independent Prolog program. It is possible to define some overall architecture for parsing and generating within Prolog that abstracts away from the differences between BUG and certain Prolog bottum up parsers like 'lc' and BUP, but I will not pursue this matter here. In the remainder of this section I will discuss two other points. In the first part Shieber's semantic monotonicity requirement will be compaired with the semantic head requirement. The second part will give some indications about the efficiency of both programs.

3.1 Semantic monotonicity vs. semantic heads

The semantic monotonicity requirement entails that analyses that require some noncompositionality are impossible. A first example of this are prepositional verbs like 'count on'. Assume that sentences like <13a> have a semantics like <13b>, and are built like <13c>. This means that at a certain stage in the analysis a pp exists with, probably, its own semantics. This semantics will look like <13d> or <13e>, which both do not subsume any part of <13b>. Therefore this analysis is not semantically monotonic. A semantically monotonic analysis of these examples will need a rule similar to <14>. So prepositions that can occur in prepositional verb constructions need to be analyzed as function words without significant semantics. This leads to redundancy and ambiguities, as sometimes these prepositions *must* contain specific semantic information. This 'escape hatch' is not available for more complicated noncompositional examples as is the case for idioms. Suppose we want to analyze <15a> as <15b>. This analysis is entirely impossible within Shieber's framework because the noun phrase 'the bucket' can never

```
<13a> John counts on Mary
<13b> count_on(john,mary).
<13c>       s

        np       vp

      john     vp       pp

               v      p    np

             count   on      mary
<13d> on(_,mary)
<13e> on(mary)
```

```
<14> pp -> p np :
     <pp head> = <p head>
     <pp semantics> = <np semantics>
```

```
<15a> John kicks the bucket
<15b> die(john)
```

be generated since it does not subsume part of <15b>. It seems that nothing can be done about this. Both examples show that noncompositional cases cannot (or not without serious problems) be handled with grammars that are semantically monotonic. However, both examples can be analyzed in a way that observes the semantic head requirement. The lexical entry 'count_on' will contain the equations <16>. At the time a vp dominating 'count' is con-

```
<16> <subcat first cat> = pp
     <subcat first semantics pred>
                = on
     <subcat first semantics arg1>
                = <semantics arg2>
     <semantics pred> = count_on
```

structed in the generation process, the semantics for the pp is instantiated by a rule similar to rule <2> of section 1. Therefore no problems arise. The same applies to idioms. One reading of the verb 'kick' will contain the following equations <17>. Again this will force

```
<17> <subcat first cat> = np
     <subcat first semantics pred>
                          = bucket
     <subcat first semantics det>
                          = the
     <semantics pred> = die
```

an object 'the bucket' to be generated while the semantics is kept very simple. I conclude that several analyses are impossible with semantically monotonic grammars that are possible for grammars that observe the semantic head requirement.

One might wonder what the answer to the opposite question is, i.e. do there exist analyses in semantically monotonic grammars that are impossible within my approach. It is clear that these must exist (e.g. by choosing the 'wrong' daughters as the semantic head in a semantically monotonic grammar); however I did not yet come across examples that make a crucial use of this. For the moment it seems that the semantic head requirement is linguistically more relevant than Shieber's monotonicity requirement.

3.2 Efficiency considerations

In general I do not yet have anything decisive to say about the difference in efficiency of Shieber's generator and BUG. However there are a few reasons that suggest that my implementation is more efficient than Shieber's. First, BUG uses a top-down oracle. Therefore the generation process seems more goal directed. Instead Shieber uses a top-down test to destroy search paths afterwards. Second, BUG uses unification almost everywhere, whereas Shieber's generator often needs to check for subsumption. At least within Prolog unification allows more efficient implementations than subsumption. Therefore this may lead to a speed advantage of BUG. A third advantage is obtained by the delay of choosing between inflectional variants of words (including the empty semantic head in case of verb second). This reduces a lot of nondeterminism. In Shieber's system this reduction of local ambiguity does not exist. However, probably this same strategy can be incorporated within Shieber's framework without serious modifications.

Summary

In this paper I have shown that existing algorithms to generate strings from a semantic formula in unification based formalisms do face some problems. I argued that better results are achieved by using BUG, a directed bottom-up algorithm, that proceeds from semantic head to semantic head. Grammars that obey the semantic head requirement <11> can be handled by BUG. I compared the algorithm with Shieber's Earley-based generation algorithm and concluded that BUG is both linguistically more expressive and computationally less expensive.

References

A.Colmerauer, *PROLOG II: Manuel de reference et modele theorique* AI-group, Faculte des Sciences de Luminy, Marseille, 1982.

M.Dymetman,P.Isabelle: "Reversible logic grammars for MT" in: *Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language* , Carnegie Mellon Univ. Pittsburgh, 1988.

A.Evers, *The transformational cycle in German and Dutch* Ph.D. Utrecht 1975.

A.Evers, "Clause Union in French and German", Utrecht 1987.

S.Hirsch,"P-Patr, a compiler for unification based grammars" in: V.Dahl and P.Saint Dizier (eds): *Natural Language Understanding and Logic Programming, II* , Elseviers Science Publishers 1987.

R.Huybrechts, "The Weak Inadequacy of Context-Free Phrase Structure Grammars" in: G.de Haan, M.Trommelen en W.Zonneveld, *Van Periferie naar Kern* Foris Dordrecht 1984.

M.Kay, "Algorithm Schemata and Data Structures in Syntactic Processing", Xerox Palo Alto 1980. Appears in: B.Grosz, K.S.Jones, B.Lynn Webber, *Readings in Natural Language Processing* Morgan Kaufman Publ. Los Altos 1986.

Y.Matusumoto, H.Tanaka, H.Hirakawa, H.Miyoshi, H.Yasukawa, "BUP: a bottom-up parser embedded in Prolog" in: *New generation Computing* , 1(2):145-158, 1983.

F.C.N.Pereira, S.M.Shieber *Prolog and Natural Language Analysis* CSLI, Stanford, 1987.

C.Pollard,I.Sag: *Information-based syntax and semantics* CSLI, Stanford, 1987.

S.M.Shieber, *An introduction to unification based approaches to grammar* CSLI, Stanford, 1986.

S.M.Shieber, "A uniform parsing architecture for Parsing and Generation" in *Coling* 1988 Budapest.

J.Wedekind, "Generation as Structure Driven Derivation" in *Coling* 1988 Budapest.

H.Zeevat,E.Klein,J.Calder: "Unification Categorial
Grammar", in: N.Haddock,E.Klein,G.Morrill *Working papers in Cognitive Science* Volume 1, University of Edinburgh, 1987.

Appendix

```
/* BUG final version
   This version of the implementation serves mainly as
   documentation - probably faster implementations are
   possible
*/

bug(FS,String):-
        sem(FS,Sem),
        copy_term(Sem,Sem2),
        numbervars(Sem,0,_),    %coherent
        generate(FS,Shad,P0-[]),
        phonology(P0,String),
        sem(Shad,ShadSem),
        subsume(Sem2,ShadSem).  %complete

generate(FS,Shad,Mother,[FS|T]-T):-
        sem(FS,Sem),
        not not sem(Mother,Sem), !, %syntactic head
        word(Word,FS,Shad).

generate(FS,Shad,Mother,P0-PN):-
        link(FS,Down),
        word(Word,Down,Shad2),
        up(Down,FS,Shad2,Shad,[Word|PI]-PJ,P0-PN).

up(X,X,Shad,Shad,P0-PN,P0-PN).
up(Down,Top,DownShad,TopShad,PI-PJ,P0-PN):-
        rule(Down,Mother,Lefts,Rights,
             DownShad,MotherShad,LeftsShad,RightsShad),
        link(Top,Mother),
        generate_list(Lefts,LeftsShad,Mother,PH-PI),
        generate_list(Rights,RightsShad,Mother,PJ-PK),
        up(Mother,Top,MotherShad,TopShad,PH-PK,P0-PN).

generate_list([],[],_,P0-P0).
generate_list([H|T],[HShad|TShad],Mother,P0-PN):-
        generate(H,HShad,Mother,P0-P1),
        generate_list(T,TShad,Mother,P1-PN).
```