An Overview of Head driven Bottom-up Generation

Gertjan van Noord
OTS RUU
Trans 10 Utrecht
vannoord@hutruu59.BITnet

November 24, 1994

Abstract

In this paper I will discuss the properties of a *tactical* generation approach that has become popular recently: head-driven bottom-up generation.

It is assumed that bidirectional grammars written in some unification- or logic-based formalism define relations between strings and some representation, usually called *logical form*. The task for a generator is to generate for a given logical form the strings that are related to this logical form by the grammar.

In the paper it will be shown that the 'early' approaches to this conceivement of the generation problem such as [21], [31] and [7] are not entirely satisfactory for general purposes. Furthermore I will define a simple bottom-up generator, called *BUG1* for reference, as prototypical for the head-driven bottom-up approach as defended by for example [30, 5, 23, 24]. I will argue that head-driven bottom-up generation is to be preferred because the order of processing is directed by the input logical form and the information available in lexical entries; moreover the algorithm puts fewer restrictions on grammars than some other generators.

1 Introduction

Since a few years several authors have been working on the *tactical* generation problem in logic- or unification-based grammar formalisms such as *DCG* [17] and *PATR II* [22] (or linguistic theories that use similar formalisms such as *LFG* [3], *UCG* [32] and *CUG* [28]). In most natural language generation systems there are two different parts: the *strategic* part, 'deciding what to say', and the *tactical* part, that is, 'how to say it'. For a discussion of this division of labour see for example [1], and also [14, 26].

A bidirectional unification grammar defines relations between strings and an abstract representation that is sometimes called the *logical form*. The parsing problem is to determine the corresponding logical form(s) for a given string; similarly the (tactical) generation problem is to determine the corresponding string(s) for a given logical form. This conceivement of the generation problem is different from the approaches of Buseman [4] and Saint-Dizier [20]. In these two works the generation method relies on the parsing of a control structure by a specialized grammar in order to construct a syntactic representation. Moreover it is different from the approach of [25] in that I assume a clear distinction between the declarative definition of the relation between strings and logical forms (the grammar) and the different processes that are used to compute this relation (the parser and/or the generator).

None of the 'early' approaches to the tactical generation problem in this context (such as [21], [31] and [7]) were entirely satisfactory for general purposes. In section 2 I will summarize some of the criticisms.

Because of these criticisms other approaches were investigated, some of which turned out to be very similar in nature. For example the work reported in [23, 30, 5] [1] can be seen as variations on one 'theme': the head driven bottom-up approach. In this approach the order of processing is geared towards the logical form of the input (head driven) and the information available in lexical entries (bottom-up). In section 3.1 I will define a simple version of such a generator: *BUG1*, to explain the properties of head-driven bottom-up generation. The shortcomings of *BUG1* are discussed together with some possible extensions. Furthermore I clarify the relation between *BUG1* and the simple left-corner parser as described in [16].

Throughout this paper I will define examples and algorithms using the programming language *Prolog*. I thus assume that readers have some knowledge of *Prolog*; excellent introductions within the NLP context are [8, 16].

In this paper I will hardly touch upon lexical issues. In fact I will assume that a lexical entry is just a unification grammar rule, accidentally without any daughters. For a more realistic approach see for example [11] and [24]. I will assume that such approaches can be incorporated in the framework described here.

2 Problems with early approaches

This section gives an overview of some of the problems that led to the head driven bottom-up approach.

Problems that a generator can face are divided in two types.

The first type of problem is the following. It is quite easy to define grammar rules which most known generators will fail to handle (eg. a *DCG* rule such

[1] Martin Kay presented a similar generator at the MT workshop 'Into the Nineties', Manchester, august 1989

as $vp \rightarrow [to], vp$). In fact, the generation problem in (unrestricted) unification grammars can easily be proved to be undecidable. Therefore each generator is usually restricted to a particular class of grammars for which it is useful, although this class is not always clearly defined. I am interested in a generator that will work for linguistically relevant unification grammars, thus I prefer some generator over another if it handles some linguistically motivated grammar whereas the other does not.

It is clear that people may disagree on the notion 'linguistically relevant'. Consequently implementations of different linguistic theories may use different generation algorithms. In this paper I am interested in the family of *lexical* and *sign-based* linguistic theories, for example theories such as *HPSG* [18], and unification-based versions of categorial grammar [32, 28].

The second type of problem is simply the performance of the generator. Although it is hard to give any complexity results for generation from a unification grammar (for example because the problem is undecidable), it seems to be possible to prefer generators that are more *goal-directed* than others — resulting in dramatical differences in performance.

First I will describe the principal problem for top-down generators such as the *LFG*-generator of Wedekind [31] and the generator for *DCG*'s of Dymetman and Isabelle [7]. Then I will discuss the chart-based bottom up generator of Shieber [21].

2.1 Top-down generators and left recursion

To illustrate the principal problem for top-down generators I will define a very simple top-down generator in *Prolog*. Assume that grammars are defined as *Prolog* clauses of the form

```
Node ---> Nodes
```

where $Node$ is a term of the form $node(Syn/LF, P0 - PN)$; $Nodes$ is a *Prolog*-list of such nodes. LF represents the logical form; $P0 - PN$ is a difference-list represention of the string and Syn can be any *Prolog* term representing syntactic information. As an example consider the grammar in figure 1 [2]. A suitable parser for this simple grammar will instantiate LF in the call

$$parse(node(s/LF, [john, kisses, the, boy] - []))$$

to $kisses(john, boy)$. Note that in this grammar a lot of 'interesting' feature percolations are defined in the lexical entries. Now consider the 'naive' top-down generation algorithm in figure 2. The algorithm simply matches a node with the mother node of some rule and recursively generates the daughters of this rule. Because of the left to right search strategy of *Prolog* this algorithm will not terminate if we give it the semantics $sleeps(boy)$, because when it applies rule

[2] For the sake of simplicity the sample grammars will define very simplistic logical forms.

Figure 1: A grammar for a fragment of English

```
node(s/LF,P0-PN) --->                  % s rule
    [ node(Np,P0-P1),
      node(vp(Np,[])/LF,P1-PN) ].

node(vp(Subj,T)/LF,P0-PN) --->         % vp complementation
    [ node(vp(Subj,[Obj|T])/LF,P0-P1),
      node(Obj,P1-PN) ].

node(np/LF,P0-PN) --->                 % np rule
    [ node(det/LF,P0-P1),
      node(n/LF,P1-PN) ].

node(np/john,[john|X]-X) --->
    [].                                % john
node(np/mary,[mary|X]-X) --->
    [].                                % mary
node(n/boy,[boy|X]-X) --->
    [].                                % boy
node(n/girl,[girl|X]-X) --->
    [].                                % girl
node(det/_,[the|X]-X) --->
    [].                                % the

node(vp(np/Np,[])/sleep(Np),           % to sleep
    [sleeps|X]-X) ---> [].

node(vp(np/Np,[np/Np2])/               % to kiss
    kisses(Np,Np2),[kisses|X]-X) ---> [].

node(vp(np/Np,[np/Np2,p/up])/          % to call up
    call_up(Np,Np2),[calls|X]-X) ---> [].

node(p/up,[up|X]-X) ---> [].           % up
```

Figure 2: A naive top-down generator

```
generate(LF,String) :-
    gen(node(_/LF,String-[])).

gen(Node) :-
    ( Node ---> Nodes ),
    gen_ds(Nodes).

gen_ds([]).
gen_ds([Node|Nodes]) :-
    gen(Node),
    gen_ds(Nodes).
```

s, it will try to generate a node with an unknown logical form and an unknown syntactic category. The algorithm thus will apply rule s for this node, and will go into an infinite loop because each time it chooses rule s for the first daughter of rule s.

Therefore, the order in which nonterminals are expanded is very important, as was noticed in [7, 31]. If in the foregoing example the generator first tries to generate the second daughter, then the first daughter can be generated without problems afterwards. In the approach of *Dymetman and Isabelle* the order of generation is defined by the rule-writer by annotating *DCG* rules. In the approach of *Wedekind* this ordering is achieved automatically by essentially a version of the goal-freezing technique [6]. Put simply, the generator only generates a node if its logical form is instantiated; otherwise the generator will try to generate other nodes first.

A specific version of the first approach is an approach where one of the daughters has a privileged status. This daughter, that we might call the 'head' of the rule will always be generated first. The notion 'head' may be defined by the rule writer or by some other method. I will simply asssume that for all rules the first daughter of the list of daughters represents the head [3]. For example, the *vp complementation* rule will now look as follows:

```
node(vp(Subj,T)/LF,P0-PN) --->         % vp complementation
    [ node(vp(Subj,[Obj|T])/LF,P0-P1), node(Obj,P1-PN) ].
```

Now without any modification to the original definition of *gen_ds* this change will imply that heads are generated first.

The resulting simple top-down generation algorithm is equivalent to the

[3] Note that this does not imply that this daughter is the leftmost daughter; the information about the left-ro-right order of daughters is represented by the difference-list representation of strings in each node. Moreover note that this particular representation may be the result of some compilation step from a representation that is more conveniant for the rule writer

approaches of [7] and [31] with respect to one major problem: the left-recursion problem.

Suppose this generator tries to generate a sentence for the logical form *sleeps*(*boy*). As required the generator will first try to generate a verb phrase after the selection of rule s, assuming that the second daughter is appropriately chosen as the head. However, to generate this verb phrase the generator will try to apply the *vp complementation* rule. For this rule to apply it will try to apply the same rule for its first daughter. Each time the same rule can be applied (predicting longer and longer lists of complements), resulting in non termination, because of left recursion.

Now the question is whether this problem is a linguistically relevant problem. According to Klaus Netter (personal communication) the foregoing type of rules can not be written in *LFG* and therefore Wedekind's generator has been used without any problems for *LFG*-grammars. I want to argue that at least for linguistic theories such as *UCG*, other versions of unification based categorial grammars and some versions of *HPSG* left-recursive rules such as the *vp complementation* rule occur very frequently. Moreover in [30, 23] we have argued at length that the problem can not be solved by an ad-hoc restriction on the length of this list of complements. Several Germanic languages require rules where this list of complements can grow during a derivation (in case of cross-serial dependencies) without a linguistically motivated upper bound. The conclusion of this section therefore is that top-down generators have problems with some linguistically motivated left recursive analyses [4].

2.2 Shieber's chart-based generator

In [21] Shieber proposes a chart-based generator where rules are applied in a bottom up fashion. Results are kept on an Earley type chart. To make this process goal driven there is only one restriction: the logical form of every subphrase that is found must subsume some part of the input logical form. This restriction results in the *semantic monotonicity* requirement on grammars; this restriction requires that the logical form of each daughter of a rule subsumes part of the logical form of the mother node of that rule. An example will clarify the strategy. Assume we want to generate a string for the logical form

$$kisses(boy, girl)$$

with grammar 1. As the generator starts it will try to select rules without any daughters (as the chart is still empty) whose logical form subsume part the input logical form.

[4] It is not necessarily the case that this left recursion is caused by the leftmost daughter of a rule; the left recursion is caused by the node that is generated first. Thus the German and Dutch equivalents of the *vp complementation* rule where the order of the head and the argument is switched present exactly the same problem.

First it can apply the rules *boy*, *girl*, *kisses* and *the*. After entering these entries on the chart the noun phrases *the girl* and *the boy* can be built. Now a VP dominating *kisses* and *the girl* will be constructed as well, resulting in a VP with the logical form

$$kisses(\_, girl)$$

Finally a rule applies that combines the NP dominating *the boy* and the VP dominating *kisses the girl* resulting in the sentence *the boy kisses the girl* with the required logical form. Note that no other rules can apply, because their resulting logical form does not subsume part of the original logical form.

The requirement that every rule application yields a logical form 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."

In fact the grammar 1 is not semantically monotonic, because it can assign the logical form

$$call\_up(boy, girl)$$

to the sentence *the boy calls the girl up*. Note that the logical form of the particle *up* does not subsume any part of the resulting logical form (*call_up* is an identifier without any internal structure). Other examples where semantic monotonicity is not obeyed are cases where semantically empty words such as 'there' and 'it' are syntactically necessary, and prepositional verbs such as 'count on'. Furthermore analyses of idioms usually will be semantically non-monotonic. As an example consider the case where a sentence like *john kicks the bucket* has a logical form $kick\_bucket(john)$.

Another disadvantage of this generator is the nondeterministic style of processing. The requirement that only rules can be applied of which the logical form subsumes some part of the input logical form does not direct the generation process very much. Furthermore subsumption checks (for example to check whether a result already is present in the chart) lead to much processing overhead. These efficiency considerations also led to the head driven bottom-up family of generators to be discussed in the next section.

Summarizing, the principal problem of top-down generators is left-recursion. This problem is solved in a chart-based bottom-up generator at the cost of severe restrictions on possible grammars, and rather inefficient processing.

3 Head driven bottom-up generation

In this section I want to discuss the merits of the family of head driven bottom-up generators. In the first section I will define *BUG1*, a simple member of this family. In the second section I will argue why the head driven bottom-up approach is favorable. The third section discusses some problems with *BUG1* and discusses some extensions to *BUG1*. The fourth section shows how parsing and generation can be incorporated in a general architecture. In the fifth section I will give some more ambitious examples and some experimental results.

3.1 *BUG1*: a head driven bottom-up generator

In this section I will present a simple variant of a head driven, bottom-up generator, called *BUG1*, which I use as prototypical for the approaches presented in [5, 23, 30, 24].

As a simplifying assumption I will require for the moment that rules have a head (unless of course a rule does not have any daughters). Moreover, the logical form of this head must be *identical* to the logical form of the mother node; i.e. the mother node and the head *share* their logical form. Note that for each rule in grammar 1 it is possible to choose a head that will fulfill this requirement.

The algorithm consists of two parts. The *prediction* part predicts a lexical entry. The *connection* part tries to build a parse tree matching the top goal and starting from this lexical entry. The algorithm *BUG1* proceeds as follows. Its input will be some node N with some logical form LF. First *BUG1* tries to find the *pivot*, a lexical entry whose logical form unifies with LF (the *prediction* step). Note that the logical form of this lexical entry will be instantiated by the prediction step. Now *BUG1* is going to build from this pivot larger entities as follows. It selects a rule whose head unifies with the pivot. The other daughters of this rule are generated recursively. For the mother node of this rule this procedure will be repeated: selecting a rule whose head unifies with the current node, generate the daughters of this rule and connect the mother node. This part of the algorithm, called the *connection* part, will end if a mother node has been found that unifies with the original node N. In *Prolog* this can be defined as in 3. As an example, consider what happens if this algorithm is activated by the query

$$bug1(node(s/kisses(mary, john), String - []))$$

First the clause *predict_word* will select the pivot, a lexical entry with a logical form that can unify with $kisses(mary, john)$. The definition for $kisses$ is a possible candidate. This results in a node

$$Small = node(vp(np/mary, [np/john])/kisses(mary, john), [kisses|X] - X)$$

This node is going to be connected to the original node by the *connect* clauses. To be able to connect the *vp* to (ultimately) the *s* a rule will be selected of which

Figure 3: The *Prolog* definition of *BUG1*

```
bug1(Node):-
    predict_word(Node,Small),
    connect(Small,Node).

connect(Node,Node).
connect(Small,Big) :-
    predict_rule(Small,Middle,Others,Big),
    gen_ds(Others),
    connect(Middle,Big).

gen_ds([]).
gen_ds([Node|Nodes]):-
    bug1(Node),
    gen_ds(Nodes).

predict_word(node(_/LF,_),node(S/LF,P)):-
    ( node(S/LF,P) ---> [] ).

predict_rule(Head,Mother,Others,_):-
    ( Mother ---> [Head|Others] ).
```

the *vp* can be the head. The *vp complementation* rule is a possible candidate. If this rule is selected the following instantiations are obtained:

$$Others = [node(np/john, X - PN)]$$

$$Middle = node(vp(np/mary, [])/kisses(mary, john), [kisses|X] - PN)$$

The list $Others$ is generated recursively, instantiating $X - PN$ into $[john|Y] - Y$. Therefore the next task is to connect

$$Middle = node(vp(np/mary, [])/kisses(mary, john), [kisses, john|Y] - Y)$$

This node can be unified with the head of rule s, thereby instantiating the logical form of the *np*. This *np* again is generated recursively and the $Middle$ node of the s rule will become:

$$node(s/kisses(mary, john), [mary, kisses, john|Y] - Y)$$

This node can easily be connected to the s by the first clause for *connect*, because it can be unified with the s node; the variable $String$ in the query will be instantiated into $[mary, kisses, john]$.

As another example consider the case where the logical form is built in a semantically non-monotonic way:

$$bug1(node(s/call\_up(mary, john), String - []))$$

The predictor step will select the lexical entry for *calls*, after which the generator will try to connect this *vp* to the *s* node, as in the foregoing example. After the generation of the first element of the complement list, *john*, the first entry of the complement list of the dominating *vp* is fully instantiated, so the particle can be generated without problem. In the connect step the *s* rule can be applied after which the connect step terminates, instantiating *String* as $[mary, calls, john, up]$. In section 3.5 I will give some more interesting examples.

3.2 Why head driven bottom-up generation?

Here I will argue why head driven generation is to be preferred.

First note that the order of processing is not left-to-right, but always starts with the head of a rule. The logical form of this head is always known by the prediction step. This constitutes the top-down information of the algorithm. Apart from the top-down logical form information the algorithm is directed by the information of the lexicon because the order of processing is bottom-up. Head driven bottom-up generators are thus geared towards the semantic content of the input on the one hand and lexical information on the other hand. Of course this is especially useful for grammars that are written in the spirit of *lexical* linguistic theories. These two sources of directedness yield generators with acceptable performance.

Apart from considerations of efficiency the major reason for constructing *bottom-up* generators has been the left-recursion problem summarized in section 2. If the base case of the recursion resides in the lexicon the bottom-up approach does not face these problems. Typically in grammars that are based on theories such as *HPSG*, *UCG* and *CUG* these cases occur frequently, but are handled by *BUG1* without any problems.

3.3 Problems and Extensions

Of course the simple architecture of *BUG1* faces a number of problems. However several extensions to *BUG1* can and have been proposed to deal with some of them.

3.3.1 Restrictions on heads

The assumption that heads always share their logical form with the mother node will be too restrictive for several linguistic or semantic theories. Some extensions to *BUG1* are possible that handle more sophisticated grammars. For example it is possible, as proposed in [30], to enlarge the power of the prediction step. By inspection of the grammar it may be possible to precompile possible relations between the logical form of some top node and the logical form of the pivot of that node.

Another extension is the architecture advocated in [23, 24], where rules are divided into two types. The first type of rules are the ones where the head indeed shares its logical form with the mother node. In the second type of rule there is no such node. The algorithm does not necessarily predict a lexical entry, but it predicts a rule of the second type, or a lexical entry. The daughters of this rule are then generated in a top-down fashion after which the mother node of the rule is connected to the top node bottom-up, as in *BUG1*. In case all rules of a grammar are of the first type the algorithm behaves similar to *BUG1*. In case no rules have a head, the algorithm reduces to a top-down generator.

In the generator for *UCG* [5] it is also assumed that heads share their logical form with the mother node ('all logical forms are projected from the lexicon'); as an extension to this, a special arrangement is made to allow for typeraising as well.

3.3.2 Nodes which share their logical form

Another problem is posed by rules where the logical form of the head is shared with the logical form of some other daughter. In fact the *np* rule of 1 is such a rule:

```
node(np/LF,P0-PN) --->          % np rule
    [ node(n/LF,P1-PN),node(det/LF,P0-P1) ].
```

In principle, this situation can lead to nontermination of the algorithm. A simple (partial) solution to the problem is to augment the algorithm as follows. In *BUG1* only semantic top-down information is used (in the prediction step). The prediction step can easily be augmented to use some syntactic information as well. Assuming that the predicate $link(SynMoth, SynHead)$ is a precompiled table of the transitive closure of possible syntactic links between mothers and heads, similar to the link predicate in the *BUP* parser [13] between mothers and left-most daughters, the definition of *select_word* and *select_rule* can be changed as follows:

```
select_word(node(M/LF,_),node(S/LF,P)):-
    ( node(S/LF,P) ---> [] ),
    link(M,S).
select_rule(Head,node(M/S,P),Others,node(Syn/_,_)):-
    ( node(M/S,P) ---> [Head|Ds] ),
    link(Syn,M).
```

In most practical cases this technique solves the problem.

3.3.3 Delayed evaluation of nodes

Although I have argued that the *heads first* approach usually implies that the logical form of a node is instantiated at the time this node has to be generated it is possible to write reasonable grammars where this is not the case. For example *raising-to-object* constructions can be analysed in a way that is problematic for *BUG1*. Assume that the logical form for the sentence *john believes mary to kiss the boy* will be $believes(john, kiss(mary, boy))$. Furthermore assume that *mary* is the syntactic object of *believes*. A reasonable definition of the lexical entry *believes* in the spirit of 1 then is the following:

```
node(vp( np/Np, [ np/Np2, vp(np/Np2,[])/Vp ])/believes(Np,Vp),
    [believes|X]-X) ---> [].
```

This lexical entry has two complements, a noun phrase and a verb phrase. Furthermore it is stated that the logical form of this noun phrase is identical to the logical form of the subject of the embedded verb phrase.

Such an analysis can be defended as follows. Assume that passive is accounted for by some lexical rule that, intuitively speaking, takes the object from the subcat list and makes it the subject (such as in *LFG*). Now if *believes* is defined as above then this passive rule will also naturally account for sentences such as *mary is believed to kiss the boy*.

Now, as the generator proceeds bottom-up it will try to generate the object noun phrase *before* the embedded verb phrase has been generated, i.e., before the link between the embedded subject and the object is found. As a result the logical form of the object is not yet instantiated and therefore *BUG1* will not terminate in this case. Assuming that the analysis of raising-to-object is correct then it might be necessary to augment *BUG1* with some version of *goal freezing*. If the generator comes across a uninstantiated logical form, then the execution of that node is suspended until the logical form is instantiated. In the case of *believes* this will imply that the embedded verb phrase will be generated first, after which the object can be generated. An implementation of this technique is reported in [29].

3.3.4 Verb Second

BUG1 can correctly handle analyses that make use of empty elements such as in cases of *gap threading* analyses of *wh-movement*, *topicalization* and *relativization* [15]. In such cases a possible pivot for, say, a nounphrase can be the empty noun phrase whose semantics obviously unifies with the input; this pivot can easily be connected to the noun phrase goal. As usual the threading of information will check whether the gap is properly related to an antecedent. In section 3.5 I will give some experimental results of a grammar that includes topicalization.

In case the *head* has been 'moved' however there will be a problem for *BUG1*. Consider the analysis of *verb second* phenomena in Dutch and German. In most traditional analyses it is assumed that the verb in root sentences has been 'moved' from the final position to the second position. Koster [12] convincingly argues for this analysis of Dutch. Thus a simple root sentence in German and Dutch usually is analysed as in the following examples:

Vandaag kust$_i$ de man de vrouw ϵ_i

Today kisses the man the woman
Vandaag heeft$_i$ de man de vrouw ϵ_i gekust
Today has the man the woman kissed
Vandaag [ziet en hoort]$_i$ de man de vrouw ϵ_i
Today sees and hears the man the woman

In *DCG* such an analysis can easily be defined, by percolating the information of the verb second position to some empty verb in the final position. Consider the following simple grammar for Dutch in 4. In this grammar a special empty element is defined for the empty verb. All information of the verb in the second position is percolated through the rules to this empty verb. Therefore the definition of the several vp-rules is valid for both root and subordinate clauses. This grammar does not handle topicalization but assumes that all root sentences start with the subject. In the appendix a more realistic grammar for Dutch is presented that handles topicalization. There is some freedom in choosing the

Figure 4: A grammar for a fragment of Dutch

```
node( s/Sem, P0-PN) --->                % s -> Subj, v, vp
    [ node(vp(Np,[],V)/Sem,P2-PN),
      node(Np,P0-P1),
      node(V,P1-P2)].

node( vp(Subj,T,V)/LF, P0-PN) --->      % vp complement
    [ node(vp(Subj,[H|T],V)/LF, P1-PN),
      node(H,P0-P1) ].

node( vp(A,B,C)/D, String) --->         % vp_v
    [ node(v(A,B,C)/D, String)].

node( vp(A,B,C)/Sem, P0-PN) --->        % vp_mod
    [ node(adv(Arg)/Sem,P0-P1),
      node(vp(A,B,C)/Arg,P1-PN) ].

node( v(A,B,node(vp(A,B,_)/Sem,String)/
    Sem, P-P) ---> [].                  % empty verb

node( np/john,[john|X]-X) --->
    [].                                 % john
node( np/mary,[mary|X]-X) --->
    [].                                 % mary
node(adv(Arg)/today(Arg),
    [vandaag|X]-X) ---> [].             % vandaag (today)
node( v(np/S,[np/O],nil)/
    kisses(S,O), [kust|X]-X) ---> [].   % kust (kisses)
```

head of rule s. If it is the case that the verb always is the semantic head of s then *BUG1* can be made to work properly if the prediction step includes information about the verb second position that is percolated via the other rules. In general however, the verb will not be the semantic head of the sentence, as is the case in this grammar. Because of the *vp_mod* rule the verb can have a different logical form compared to the logical form of the s. This poses a problem for *BUG1*. The problem comes about because *BUG1* can (and must) at some point predict the empty verb as the pivot of the construction. However in the definition of this empty verb no information (such as the list of complements) will get instantiated (unlike in the usual case of lexical entries). Therefore the *vp complement* rule can be applied an unbounded number of times. The length of the lists of complements now is not known in advance, and *BUG1* will not terminate.

In [30] an ad-hoc solution is proposed. This solution assumes that the empty verb is an inflectional variant of a verb. Moreover inflection rules are only applied after the generation process is finished (and has yielded a list of lexical entries). During the generation process the generator acts as if the empty verb is an ordinary verb, thereby circumventing the problem. However this solution only works if the head that is displaced is always a lexical entry. This is not true in general. In Dutch the verb second position can not only be filled by (lexical) verbs but also by a conjunction of verbs. Moreover it seems to be the case that some constructions in Spanish are best analysed by assuming the 'movement' of complex verbal constructions to the second position (*vp second*). For example in the case of *V-preposing* [27], and *Predicate Raising* ([2] and references cited there; but see also [9]).

Here I will propose a more general solution that requires some cooperation of the rule writer. In this solution it is assumed that there is a relation between the empty head of a construction and some other construction (in the case of verb second the verb second constituent). However the relation is usually implicit in a grammar; it comes about by percolating the information through different rules from the verb second position to the verb final position. I propose to make this relation explicit by defining an empty head as a *Prolog* clause with two arguments as in:

```
gap( node( v(A,B,nil)/Sem, String),
     node( v(A,B,node(v(A,B,nil)/Sem,String))/Sem,P-P)).
```

This definition can intuitively be understood as follows: once you have found some node A (the first argument of *gap*), then there could have been just as well the (gap-) node B (the second argument of *gap*). Note that a lot of information is shared between the two nodes, thereby making the relation between the antecedent and the empty verb explicit [5]. Another way to understand this is by writing the gap declaration as an ordinary definite clause:

[5] This may suggest that the relation between the antecedent and gap is now fully acounted for by this definition (as in for example Extraposition Grammars [15]); however in this solution the rulewriter is still responsible for the percolation of the verb second information in each

```
node( v(A,B,node(v(A,B,nil)/Sem,String))/Sem,P-P) :-
    node( v(A,B,nil)/Sem, String).
```

The use of such rules can be incorporated in *BUG1* by adding the following clause for *connect*:

```
connect(Small,Big) :-
    gap(Small,Gap),
    connect(Gap,Big).
```

Note that the problem is now solved because the rule for the gap will only be selected after its antecedent has been built. Some parts of this antecedent are then unified with some parts of the gap. The subcat list for example will thus be instantiated in time.

3.3.5 Other extensions

The generator defined so far will generate sentences with logical forms that are *compatible* with the input logical form. It is possible to define a stricter generator that will only generate sentences whose logical form subsumes the input logical form (*coherence*) and is subsumed by the input logical form (*completeness*). For a discussion see [31, 30, 23].

The generator as it stands may be rather non-deterministic with respect to the particular choice of the inflectional form of lexical entries. For example the agreement features of a verb may only be known after the generation of the subject. Some uninteresting non-determinism can be eliminated by postponing this choice to a postprocess. Instead of yielding a string the generator will yield a list of lexical entries for which inflectional rules select the apropriate form. For more details see [24].

It is also possible to allow for extra *Prolog* predicates in *DCG* rules. For example in [23] this possibility is used to implement quantifier storage. In general there may be some problems with this technique as the order of selection of literals is different than in the usual top-down *DCG* case. Again delayed evaluation techniques will prove useful here.

3.4 A uniform architecture

Shieber argues for a uniform architecture for parsing and generation [21]. I will argue that *BUG1* and a simple version of a left-corner parser [16] can be regarded as two instantiations of a simple general proof procedure for Horn clauses. Such a general procedure can be defined as in figure 5. In this definition the two clauses *predict_unit* and *predict_rule* are parameterized for a specific task such as parsing or generation. In the case of parsing *predict_unit* simply

rule.

Figure 5: A general proof procedure

```
prove(Call) :-
    predict_unit(SubCall,Call),
    connect(SubCall,Call).

connect(Call,Call).
connect(Child,Ancestor):-
    predict_rule(Child,Ancestor,Parent,Siblings),
    prove_ds(Siblings),
    connect(Parent,Ancestor).

prove_ds([]).
prove_ds([H|T]):-
    prove(H),
    prove_ds(T).
```

finds a representation of the first word of the sentence, in the case of generation it finds a representation of the pivot of the sentence. The clauses for *predict_rule* similarly are parameterized to define different orders of selection of the literals of the clause. For parsing a left-to-right selection is useful; for generation a head-first selection. Thus the above general proof procedure constitutes a bottom-up proof procedure where top-down filtering is defined via the *predict_unit* and *predict_rule* predicates. In [19] a generalized version of this proof procedure is defined where *predict_unit* is replaced by a predicate that predicts a non_chain_rule.

3.5 Some experimental results

In this section we will give some experimental results achieved with a specific version of the generation algorithm described here. This version is implemented in Sicstus Prolog by Herbert Ruessink and the author and runs on a DEC RISC workstation 3100. In the appendix a DCG grammar for a fragment of Dutch is defined. This grammar is compiled into a bottom up parser and a head driven bottom up generator. The small grammar handles the following types of construction:

- topicalization
- verb second
- subcategorization
- cross serial dependencies

- subject and object control verbs
- auxiliaries
- some idioms
- extraposition

The time taken to generate some sentences from their corresponding logical forms is given in appendix A.

The algorithm is also in use in a more elaborated system: the MiMo2 research MT system. Usual generation times for this system that contains grammars for Dutch, English and Spanish with a much broader coverage and larger lexicon are satisfactory. However generation times can increase dramatically if analyses are defined where the logical form is build in a non-compositional way.

4 Summary

In this paper I have given an overview of algorithms for tactical generation within unification based formalisms. I have argued that head driven bottom-up generators are useful for two reasons. First the order of processing is geared towards the semantic content of the input and the information in the lexicon. This results in goal-directed algorithms with reasonable performance in practice. Second this order of processing puts less restrictions on grammars than top-down generators and Shieber's chart-based bottom-up generator. Especially for grammars written in the spirit of lexical, sign-based linguistic theories head driven bottom-up generators are useful.

Acknowledgements

It is clear that this report bears heavily on work by and discussions with the authors of [23, 31, 7, 5, 21]. Moreover I would like to thank my collegues of the MiMo2 project - a research MT-project of some members of Eurotra-Utrecht. I was supported by the European Community and the NBBI through the Eurotra project.

References

[1] Douglas E. Appelt. Bidirectional grammars and the design of natural language generation systems. In *Theoretical Issues in Natural Language Processing 3*, 1987.

[2] Ivonne Bordelois. Causatives: from lexicon to syntax. *Natural Language and Linguistic Theory*, 6, 1988.

[3] Joan Bresnan, editor. *The Mental Representation of Grammatical Relations*. MIT Press, 1982.

[4] Stephan Busemann. Generation strategies for GPSG, 1989. Paper presented at the Second European Workshop on Natural Language Generation.

[5] Jonathan Calder, Mike Reape, and Henk Zeevat. An algorithm for generation in unification categorial grammar. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, 1989.

[6] Alain Colmerauer. *PROLOG II: Manuel de réference et modèle théorique*. Groupe d'Intelligence Artificielle, Faculté, des Sciences de Luminy, 1982.

[7] Marc Dymetman and Pierre Isabelle. Reversible logic grammars for machine translation. In *Proceedings of the Second International Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages*, 1988.

[8] Gerald Gazdar and Christopher S. Mellish. *Natural Language Processing in Prolog; an Introduction to Computational Linguistics*. Addison Wesley, 1989.

[9] Anneke Groos and Reineke Bok-Bennema. The structure of the sentence in spanish. In Ivonne Bordelois, Helen Contreras, and Karen Zagona, editors, *Generative Studies in Spanish Syntax*. Foris Dordrecht, 1986.

[10] Barbara Grosz, Karen Sparck Jones, and Bonny Lynn Webber, editors. *Readings in Natural Language Processing*. Morgan Kaufmann, 1986.

[11] Pierre Isabelle, Marc Dymetman, and Elliott Macklovitch. CRITTER: a translation system for agricultural market reports. In *Proceedings of the 12th International Conference on Computational Linguistics*, 1988.

[12] Jan Koster. Dutch as an SOV language. *Linguistic Analysis*, 1, 1975.

[13] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom up parser embedded in prolog. *New Generation Computing*, 1(2), 1983.

[14] Kathleen McKeown. *Text Generation*. Cambridge University Press, 1985.

[15] Fernando C.N. Pereira. Extraposition grammars. *Computational Linguistics*, 7(4), 1981.

[16] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford, 1987.

[17] Fernando C.N. Pereira and David Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980. reprinted in [10].

[18] Carl Pollard and Ivan Sag. *Information Based Syntax and Semantics*. Center for the Study of Language and Information Stanford, 1987.

[19] Herbert Ruessink and Gertjan van Noord. Remarks on the bottom-up generation algorithm. Technical report, Department of Linguistics, OTS RUU Utrecht, 1989.

[20] Patrick Saint-Dizier. A generation method based on principles of government and binding theory, 1989. Paper presented at the Second European Workshop on Natural Language Generation.

[21] Stuart M. Shieber. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, 1988.

[22] Stuart M. Shieber, Hans Uszkoreit, Fernando C.N. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In B. J. Grosz and M. E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge.* SRI report, 1983.

[23] Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. A semantic-head-driven generation algorithm for unification based formalisms. In *27th Annual Meeting of the Association for Computational Linguistics*, 1989.

[24] Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 1990. To appear.

[25] Koenraad J.M.J. De Smedt. IPF: an incremental parallel formulator. this volume.

[26] Henry Thompson. Strategy and tactics: a model for language production. In *Papers from the Thirteenth Regional Meeting*, 1977.

[27] Esther Torrego. On inversion in spanish and some of its effects. *Linguistic Inquiry*, 15, 1984.

[28] Hans Uszkoreit. Categorial unification grammar. In *Proceedings of the 11th International Conference on Computational Linguistics*, 1986.

[29] Maarten A. J. van Hemel. A delayed generation strategy for a semantic-head-driven generation algorithm. Technical report, Department of Linguistics, OTS RUU Utrecht, 1989.

[30] Gertjan van Noord. BUG: A directed bottom-up generator for unification based formalisms. *Working Papers in Natural Language Processing, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht*, 4, 1989.

[31] Jürgen Wedekind. Generation as structure driven derivation. In *Proceedings of the 12th International Conference on Computational Linguistics*, 1988.

[32] Henk Zeevat, Ewan Klein, and Jo Calder. Unification categorial grammar. In Nicholas Haddock, Ewan Klein, and Glyn Morrill, editors, *Categorial Grammar, Unification Grammar and Parsing.* Centre for Cognitive Science, 1987. Volume 1 of Working Papers in Cognitive Science.

A Generation times for a DCG grammar for Dutch

The first line for each entry is the input logical form, then follows a sentence that is generated with its English translation. The fourth line for each entry gives the time in seconds cputime to generate the first sentence and the total time to compute all possible sentences. Most multiple results come about because different topicalizations are possible according to the grammar. Also different attachments are possible for extraposed constituents (that do not correspond to any differences in logical form according to this grammar).

1.
   - $perf(try(jane, kiss(jane, tarzan)))$
   - jane heeft tarzan proberen te kussen
   - jane has tried to kiss tarzan
   - 1 sec. — 4 sec. (4 results)
2.
   - $perf(see(pl(man), help(tarzan, kiss(sg(vrouw), jane))))$
   - de mannen hebben tarzan de vrouw jane zien helpen kussen
   - the men have seen tarzan help the woman to kiss jane
   - 1 sec. — 2 sec. (8 results)
3.
   - $fut(perf(try(tarzan, catch(tarzan, jane))))$
   - tarzan zal jane op hebben proberen te vangen
   - tarzan will have tried to catch jane
   - 1 sec. — 4 sec. (4 results)
4.
   - $perf(say(pl(man), see(jane, in\_trouble(pl(woman)))))$
   - de mannen hebben gezegd dat jane de vrouwen in de puree ziet zitten
   - the men have said that jane sees the women are in trouble
   - 4 sec. — 13 sec. (8 results)
5.
   - $force(pl(men), jane, see(jane, help(tarzan, kiss(pl(woman), pl(mary)))))$
   - de mannen dwingen jane om tarzan de vrouwen mary te zien helpen kussen
   - the men force jane to see tarzan help the women to kiss mary
   - 10 sec. — 35 sec. (24 results)

B a simple DCG for Dutch

Each node is a term $x(Cat/LF, Topic, Verb2, Deriv)$ where $Topic$ is used for a gap threading analysis of topicalization; $Verb2$ percolates information on verb second; and $Deriv$ is a simple derivation tree. Cat is a term representing syntactic information and LF represents the argument structure, input for generation.

Arguments of $partial\_ex$ are predicates that are executed during compilation. Thus some of these rules will become 'metarules', in case these executable calls have several solutions. The 'object' grammar is thus somewhat larger.

```
partial_ex( extra/2 ).  partial_ex( verb/4 ).
partial_ex( aux/5 ).    partial_ex( member/2 ).
partial_ex( topicalizable/1) .

% main -> topic v0 s
x(main/LF,X-X,nil,main(T,V,S)) -->
  x(Topic, Y-Y, nil, T),  { topicalizable(Topic) },
  x(v0(VT,fin,B,C)/VS,Z-Z,nil,V),
  x(s(fin)/LF,Topic-nil,v0(VT,fin,B,C)/VS,S).

topicalizable(np(_)/_).   topicalizable(pp/_).

% sbar -> comp s
x(sbar(fin)/LF,I-I,nil,sbar(C,S)) -->
  x(comp(fin,SS)/LF,O-O,nil,C),
  x(s(fin)/SS,P-P,nil,S).

x(sbar(infin,Sj)/LF,I-I,nil,sbar(C,S)) -->
  x(comp(infin,SS)/LF,O-O,nil,C),
  x(vp(te,Sj,[])/SS,P-P,nil,S).

% s -> np vp
x(s(VF)/LF,I-O,Verb,s(SS,VV)) -->
  x(np(Agr)/Sj,I-O2,nil,SS),
  x(vp(VF,np(Agr)/Sj,[])/LF,O2-O,Verb,VV).

% vp -> compl vp
x(vp(VF,S,T)/LF,I-O,V,vp(C,VP)) -->
  x(H,I-O2,nil,C),  { extra(H,no) },
  x(vp(VF,S,[H|T])/LF,O2-O,V,VP).

% vp -> vp compl (extraposition)
x(vp(VF,S,T)/LF,I-O,V,vp(VP,C)) -->
  x(vp(VF,S,[H|T])/LF,O2-O,V,VP),
  x(H,I-O2,nil,C),   { extra(H,yes) }.

extra(sbar(_)/_,yes).  extra(sbar(_,_)/_,yes). % must be extraposed
extra(np(_)/_,no).     extra(part/_,no).       % may not
extra(pp/_,_).                                 % can be extraposed

% vp -> v1
x(vp(VF,Sj,Sc)/LF,I-O,V,vp(VV)) -->
  x(v(VF,Sj,Sc-[])/LF,I-O,V,VV).
```

```
% v1 -> v0
x(v(VF,Sj,Sc)/LF,I-O,V,v(VV)) -->
  x(v0(main,VF,Sj,Sc)/LF,I-O,V,VV).

% v1 -> aux v1
x(v(F,Sj,Sc)/LF,I-O,V,v(A,B)) -->
  x(v0(aux,F,Sj,v(_A,_B,_C)/_S+Sc)/LF,I-I2,V,A),
  x(v(_A,_B,_C)/_S,I2-O,nil,B).

% gap -> v0
gap(x(v0(VT,fin,Sj,Sc)/LF,_,nil,Tree),
  x(v0(VT,fin,Sj,Sc)/LF,X-X,v0(VT,fin,Sj,Sc)/LF,vgap)).

% np -> det n
x(np(Nm)/LF,I-I,nil,np(Det,N)) -->
  x(det(Ns,Nm)/LF,nil-nil,nil,Det),
  x(n(Nm)/Ns,nil-nil,nil,N).

% pp -> p np
x(pp/LF,I-I,nil,pp(P,NP)) -->
  x(p(Np)/LF,J-J,nil,P),
  x(np(_)/Np,O-O,nil,NP).

% topic ->
x(Topic,Topic-nil,nil,topic_gap) -->
  { topicalizable( Topic ) }.

x(np(sg)/john,I-I,nil,np(john)) --> [jan].
x(np(sg)/mary,I-I,nil,np(mary)) --> [marie].
x(np(sg)/jane,I-I,nil,np(jane)) --> [jane].
x(np(sg)/tarzan,I-I,nil,np(tarzan)) --> [tarzan].
x(n(sg)/sg(man),I-I,nil,n(man)) --> [man].
x(n(pl)/pl(man),I-I,nil,n(mannen)) --> [mannen].
x(n(sg)/sg(woman),I-I,nil,n(vrouw)) --> [vrouw].
x(n(pl)/pl(woman),I-I,nil,n(vrouwen)) --> [vrouwen].
x(n(sg)/sg(mashed_potatoes),I-I,nil,n(puree)) --> [puree].
x(det(Ns,_)/Ns,I-I,nil,det(de)) --> [de].
x(det(Ns,pl)/Ns,I-I,nil,det(e)) --> [].
x(p(Np)/naar(Np),I-I,nil,p(naar)) --> [naar].
x(p(Np)/in(Np),I-I,nil,p(in)) --> [in].
x(p(Np)/van(Np),I-I,nil,p(van)) --> [van].
x(p(Np)/aan(Np),I-I,nil,p(aan)) --> [aan].
x(part/op,I-I,nil,part(op)) --> [op].
x(comp(fin,S)/dat(S),O-O,nil,comp(dat)) --> [dat].
x(comp(fin,S)/because(S),O-O,nil,comp(dat)) --> [omdat].
x(comp(infin,S)/S,O-O,nil,comp(om)) --> [om].
x(comp(infin,S)/S,O-O,nil,comp(om)) --> [].

% verbs and auxes pick out an inflectional variant:
x(v0(main,Fin,np(Agr)/A1,Sc)/LF,I-I,nil,v0(Form)) -->
  [ Form ],   { verb(LF,np(Agr)/A1,Sc,List),
  member(Fin/Agr/Form,List) }.
```

```
x(v0(aux,Fin,np(Agr)/A1,Verb+Sc)/LF,I-I,nil,aux(Form)) -->
  [ Form ],   { aux(LF,Verb,Sc,np(Agr)/A1,List),
  member(Fin/Agr/Form,List) }.

verb(sleep(A1),_/A1,X-X,
  [fin/sg/slaapt,fin/pl/slapen,infin/_/slapen,part/_/geslapen]).

verb(catch(A1,A2),_/A1,[part/op,np(_)/A2|X]-X,   %particle verb
  [fin/sg/vangt,fin/pl/vangen,infin/_/vangen,part/_/gevangen]).

verb(in_trouble(A1),_/A1,[pp/in(sg(mashed_potatoes))|X]-X, %idiom
  [fin/sg/zit,fin/pl/zitten,infin/_/zitten,part/_/gezeten]).

verb(kiss(A1,A2),_/A1,[np(_)/A2|X]-X,
  [fin/sg/kust,fin/pl/kussen,infin/_/kussen,part/_/gekust]).

verb(say(A1,A2),_/A1,[sbar(fin)/dat(A2)|X]-X,
  [fin/sg/zegt,fin/pl/zeggen,infin/_/zeggen,part/_/gezegd]).

% object control
verb(force(A1,A2,A3),Sj/A1,[sbar(infin,Sj/A1)/A3,np(_)/A2|X]-X,
  [fin/sg/dwingt,fin/pl/dwingen,infin/_/dwingen,part/_/gedwongen]).

verb(try(A1,A2),Sj/A1,[sbar(infin,Sj/A1)/A2|X]-X,  % subject control
  [fin/sg/probeert,fin/pl/proberen,infin/_/proberen,part/_/geprobeerd]).

verb(give(A1,A2,A3),_/A1,[np(_)/A2,np(_)/A3|X]-X,
  [fin/sg/geeft,fin/pl/geven,infin/_/geven,part/_/gegeven]).

verb(give(A1,A2,A3),_/A1,[pp/aan(A3),np(_)/A2|X]-X,
  [fin/sg/geeft,fin/pl/geven,infin/_/geven,part/_/gegeven]).

aux(perf(X),v(part,np(Agr)/A1,Sc)/X,Sc,np(Agr)/A1,
  [ fin/sg/heeft,fin/pl/hebben,infin/_/hebben, part/_/gehad]).

aux(fut(X),v(infin,np(Agr)/A1,Sc)/X,Sc,np(Agr)/A1,
  [ fin/sg/zal,fin/pl/zullen,infin/_/zullen ]).

aux(X,v(infin,Sj,Sc)/X,Sc,Sj,[ te/_/te ] ).

aux(try(A1,A2),v(te,_/A1,Sc)/A2,Sc,np(Agr)/A1, %raising
  [ fin/sg/probeert,fin/pl/proberen,infin/_/proberen,part/_/proberen ] ).

aux(see(A1,A2),v(infin,Sj,A-[Sj|E])/A2,A-E,np(Agr)/A1, % aci raising
  [ fin/sg/ziet, fin/pl/zien, infin/_/zien,part/_/zien ] ).

aux(help(A1,A2),v(infin,Sj,A-[Sj|E])/A2,A-E,np(Agr)/A1, % aci raising
  [ fin/sg/helpt, fin/pl/helpen,infin/_/helpen,part/_/helpen ] ).
```