Yet Another Head -driven Generator Of Natural Language

Dan Tufis

National Institute for R&D in Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA
e-mail:tufis@valhalla.racai.ro

Abstract: The paper discusses some basic issues in natural language generation (NLG) and describes a head-driven NLG system for HPSG-like language descriptions. We address mainly the aspects of surface natural language generation starting from a meaning representation of the message meant to be verbalized. The representation of the linguistic knowledge (grammar and lexicon) as well as the most important implementation issues (the generator is implemented in SMALLTALK) are described in detail, also commenting on the SMALLTALK code.

Dr. Dan Tufis is senior researcher and head of Computational Linguistics Research Laboratory at ICI Bucharest and director of the Romanian Academy Center for Artificial Intelligence (RACAI). He is Professor of Corpus Linguistics at the University of Bucharest.

He authored and co-authored 5 books, more than 80 papers in reviewed journals and more than 120 research reports on natural language processing, artificial intelligence and functional programming. He was awarded twice , in 1989 and in 1993, the Romanian Academy Prize for his contributions in the field of Natural Language Processing.

In 1997 he was elected Corresponding Member of the Romanian Academy.

Dr. Dan Tufis is an active member of various international professional associations (EACL, GLDV, ELSNET, TELRI, etc.) and is on the Editorial Board of several publications (Kluwer Book Series on Text, Speech and Language Technology, Computers& Humanities, International Journal on Corpus Linguistics, Studies in Informatics and Control, International Journal of Science and Technology of Information).

His recent research interests include corpus-based processing of natural language (he developed the tiered-tagging and the combined classifier methodology for accurately tagging highly inflectional languages with very large tagseyts), natural language learning by combining symbolic and statistical approaches.

1. Introduction

Several researchers have noted that NLG is a much less investigated area of natural language processing if compared with work on analysis, although many useful applications (machine translation, automatic summarization, question-answering, dialogue systems, CALL and Intelligent CALL, etc.) are supposed to take advantage of the developments in NLG (see for instance[Fawcet and Tucker,1989], [McDonald,1993], [Zock,1996], [Tufis and Zock, 1997], [Dale and Mellish, 1998], etc). This could be partially explained by the frequently met misconception that generation is just the reverse of analysis, but also by the fact that ad-hoc generation methods (pattern-based, for instance) proved useful enough for the specific purpose of various applications. It was mainly the study of the reversibility issues that showed that although it was possible to use the same linguistic information for both analysis and generation, the processes themselves differed a lot in terms of both their algorithmic complexity and control and only the shallow phases of these two processes (parsing and surface realization respectively) could be considered more or less the reverse of one another. In the mid80s, successful research was carried on the possibility of a common approach of the parsing and surface generation leading to the unification of the knowledge representation for both of these [Shieber,1988], [Dymetman and Isabelle, 1988] and for many others. The reversibility of linguistic descriptions became possible by adopting declarative formalisms, mainly those based on the unification of attribute-value structures.

Reusability of the linguistic knowledge makes a natural prerequisite due to the technological and conceptual advances over the last few years and also to those predictable for the future. If one considers the effort needed to implement a processing environment and the pains taken for a wide coverage language description (according to some estimates, this ratio is of at least 1:100), the reusability criterion in encoding language descriptions should normally prevail over efficiency. A linguistic description closely related to a particular formalism, whatever performant it might be, is at the risk of being partially or even completely reformulated, if the programming environment has to be changed. This is a first aspect of reusability. A second subtler one refers to avoiding radical decisions in linguistic phenomena modelling, based on a formalism or linguistic theory in fashion at a certain time. More difficult to realise, this aspect of reusability may be

facilitated through localizing very precisely all the elements specific to a certain formalism or linguistic theory.

The lexicalization trend in grammar modelling, a generalized practice nowadays, and, for sure, facilitated by the unification technology, is one possible answer to making reusability work in the large. Probably this is one reason why they are generally accepted. Our work shares this lexicalized, unification-based vision of the language processing.

When speaking about generation, it is usually the case that the distinction made between what has to be said and how it should be said is further refined into smaller tasks, such as text planning, sentence planning and linguistic realization [Dale and Mellish, 1998]. Coming closer to the implementation of a working architecture, Reiter and Dale (1997) identified 6 main problematic areas which, although overlapping to some extent, define at a finer grained level the strategy-tactics dichotomy [Dale and Mellish, 1998]:

- Content determination -decide on what information should be part of the text, and what information should stay out.
- Document structuring – decide on how the text should be organized and structured. It is often the case that the information representation out of which a generator is supposed to produce coherent text, is not structured at a sentence equivalent granularity.
- Lexicalization (lexical choice) – choose the particular words or phrases which best convey the intended meaning.
- Aggregation – somehow related to the problem of document structuring, this issue is concerned with congesting (whenever the case is) information into one sentence realization form, for the sake of fluency and stylistic relevance.
- Referring expression generation– determine what properties of any entity should be used in referring to that entity (a differentiation of potential similar referents).
- Surface realization – determine how the proper production of an NL text will be done based on the information provided at the previous five processing levels. This includes production of adequate morphological forms of the selected words and their serialization according to the syntactic restrictions specified in the grammar.

One pervasive problem is that of the nature and the format of the input an NL generator is supposed to produce texts from. This problem, which is known as the mapping problem or the problem of logical equivalence in computational linguistics [Shieber, 1993], [McDonald, 1993], can be raised with respect to each of the six phases mentioned above. The problem can be split into two subproblems: to define a loss-less mapping and to define a computationally efficient mapping.

We address here only the process of surface realization, and assume an input representation consistent with the linguistic knowledge representation, thus avoiding the mapping problem.

The structure of this paper is as follows: the first Part of the paper presents the general issues put by natural language surface generation, shows the limitations of the classical top-down and bottom -up generation methods, and introduces the head-driven generation method. Further we discuss the linguistic representation of the input, the grammar and the lexicon, and dwell on the implementation of our generator. The last Section presents preliminary results, current limitations and future work.

2. Surface Generation of Natural Language

A declarative view of grammars as promoted by the unification formalisms, predicts that grammars should themselves be interpretable without regard to direction that they should be equally suitable for use by parsers and generators. To put it more simply, in such an environment the generator works by taking a feature-structure (FS) and searching for grammar rules and lexical entries, which can relate subparts of the FS to words.

Among the different approaches to surface generation of natural languages, the head-driven models appear to be the most successful. From this point of view, our system is yet another head -driven generator not committed to a specific linguistic formalism, and just requiring a unification-based description of the linguistic knowledge which would unambiguously identify a logical form supposed to represent the semantics of the embedding feature structure. However, as the reader will notice, the grammar and the lexicon descriptions are close to an

HPSG style description. We have selected this notation for the same reasons Wilcock and Matsumoto [1998] selected HPSG, namely that a head-driven generator is supposed to take full advantage of linguistic descriptions expressed in a head-based manner. As HPSG is head -driven not only syntactically, but also semantically, it is natural to expect that a head-driven generator be an appropriate design option. In [Wilcock and Matsumoto, 1998] there were shown some inherent difficulties when a head -driven generator was supposed to work with full HPSG textbook semantics [Pollard and Sag, 1994]. The difficulties turned up due to the usual phrasal amalgamation of the quantifier storage and contextual background conditions[1]. With a lexical amalgamation of the quantifier storage as proposed by [Pollard and Yoo 1995], and of the CONTEXT feature as proposed by [Wilcock, 1997], the difficulties in using a head- driven generator for an HPSG grammar are, to a large extent, almost surpassed. The other obstacles in Wilcock's generator count with the limited power of the implementation means (template expansion and difference lists) provided by the programming environment, and are due to the lack of a dynamic constraint checking mechanism.

The surface (tactical [Thompson, 1977]) generator, to be presented here, is based on an efficient head-driven algorithm similar to the one used in BUG1 [van Noord, 1988] but, contrary to most other head -driven generators, and taking advantage of the language expressivity and computational properties, this one is implemented in Smalltalk. The generator is a mixed variant of van Noord's BUG1 and Shieber's Head -driven Generation Algorithm [Shieber, 1988], which incorporates advantages and copes with deficiencies of both of them. The extensions proposed by Wilcock and Matsumoto [1998] have also been observed.

For more efficiency , several implementation issues have been considered (indexing the lexicon on the logical structures, parallel search, copying/destructive unification of feature structures, etc.).

1 Wilcock (1997) shows that the Semantics Principle needs reformulation and it is replaced by three other principles: Semantic Head Inheritance Principle, Quantifier Inheritance Principle and Contextual Head Inheritance Principle. The rationale for this reformulation of the Semantics Principle is the reconciliation of the HPSG definition of semantic head (based on adjunct daughter or syntactic head daughter) and the one used by a head-driven algorithm (based on the identity of the logical forms). This reconciliation is achieved by including the unscoped quantifiers and the background conditions in the logical form out of which a head-driven generator is assumed to produce the surface string.

In the following we will briefly discuss some computational problems inherent to simple control strategies (top-down or bottom-up) which have motivated the combined strategy used in head -driven generators (top-down and bottom-up). We will also discuss completeness and coherence [Wedekind, 1988] and how they are possible to achieve in our generator. Then we will detail the Smalltalk implementation of the generator.

We distinguish, as to the input to the surface generator, between the *content* specification, the one that would head-drive the generation and the *functional* specification, the one that would goal-drive the restrictions application. The content specification will reflect in the lexical options, while the functional specifications will manifest in the final realization of the selected lexical entries. For instance, a logical form such as *write (john, novel)* provides merely the content specification and unless some functional *defaults* are assumed (cat: s, tense: present, number: singular, definiteness: indef, etc.) the logical form is grossly under-specified.

This approach provides the surface generator with a feature-structure within which the logical form represents just the value of a feature called *sem,* supposed to exist among the first-level features of the feature structure. The functional specifications are supposed to be there, embedded into the structure representing the value of the first-level feature *syn*. The graphemic realizations of the lexical items (currently the lexicon is a word-form one, with all the values responsible for the inflected form being fully instantiated) are to be found in the first-level feature called *phon*. We assume that whatever information is provided on quantification (QSTORE) as well as on presuppositions and other discourse information (CONTEXT), it is embedded into the *sem* attribute structure. This would account for a lexical amalgamation of the QSTORE and the CONTEXT sets as proposed by Wilcock and Matsumoto [1998].

Whereas in the representation of the input to the surface generator described in this paper the syntactic and "semantic" information is mix-up, it is relatively simple to design an interface which separates the two types of information. In [Tufis et al, 1994] there is described such an interface called KRIL (Knowledge Representation to InterLingua), used to link the immersive language tutoring system FLUENT [Hamburger et al, 1993], with the natural language generator of the multilingual natural language processing system, ATHENA

[Felshin, 1993]. The structure in Figure 1a describes the input specification for the sentence *"Does every person love Mary?"* while the structure in Figure 1b corresponds to the sentence *"Every person loved her"*.

```
(kril (to-say
      :action :love
      :agent(object :class :person
                    :attribute :every
                    :arole :quantity)
                    :object1 Mary)
       (how-to
         :s-type :interrogative
         :tma'(:present)
         :how((agent :exophoric))
```

```
(kril (to-say
      :action :love
      :agent(object :class :person
                 :attribute :every
                 :arole :quantity)
                 :object1 Mary)
     (how-to
     :s-type :affirmative
     :tma(:past)
     :how((agent :exophoric)
          (object1 :anaphoric)
```

Figure 1. Separation of A Logical Form from the Realization Specifications

This description is mapped (via lexicon inspection) onto a more linguistically -oriented representation (called Interlingua) and then passed to the underlying surface generator.

3. Why Top-Down or Bottom-Up Generators Are Not Enough

As the theory of parsing [Aho, 1972] shows, in case of top-down, left-right approaches, the grammar rules have to be free of left-recursion, otherwise the process will not stop. Not only will the direct recursiveness (such as VP→VP COMP) be forbidden, but also will the indirect one be (such as A→B C…Z; B→A D…Y). To put it more formally, if we denote by α a string of terminal grammar symbols, by Ω a string of either terminal or non-terminal grammar symbols and by FIRSTNT(X) =[Y1 Y2 … Yk] the list of non-terminals so that they are the left-most non-terminal of whatever sentential form $\alpha Y \Omega$ that can be generated by the grammar starting with X as a start symbol, then for the top-down generators the relation $X \notin$FIRSTNT(X) should exist. For a grammar involving feature-structure descriptions, the precondition should be modified by replacing the equality in the membership test by a unification condition, that is X should not get unified with any Y in FIRSTNT(X).

The use of a rule such as VP→VP COMP in modelling natural languages is not an accident, but a common construction in many languages for specifying the complements of a verb. The intent is that the rule should apply recursively until the subcategorization list of the head verb is empty (saturated) or, to put it otherwise, until all the complements of the verb are consumed.

In [Shieber et al, 1988] an example of Dutch cross-serial verb constructions is discussed, where subcategorization lists may be appended by syntactic rules, resulting in indefinitely long lists. One could argue that syntactic information might block infinite recursion, as any *subcat* list has a fixed length (this length is specified in the lexical entries) but, this information is not available for top-down generators because they would look-up the lexicon only at a final stage when preterminal categories are dealt with. A more detailed discussion and more examples of how a simple top-down generator would fail with tiny left-recursive grammars are to be found in [van Noord, 1988] and [Shieber et al, 1989].

Another kind of difficulty a simple top-down generator would face with is discussed in [Shieber et al]. The problem is illustrated by a rule like: s/S → np/NP, vp(NP)/S. Consider that the generation process is initiated with the goal *s/love(john, mary)*, which will fire the rule **s/S→np/NP, vp(NP)/S**. If the generation proceeds with the left-most non-terminal (the subgoal **np/NP**), the search space for the binding of NP will become huge (since any NP covered by a grammar would get unified with the unrestricted semantics of **np**). Thus, in a large system, although not properly an infinite recursion, this kind of a problem may cause the same hanging-around as a genuine left-recursion. And this, in spite of the **vp** semantics being (due to the variable sharing) bound to the semantics *love(john, mary)*. Several solutions to this problem are to be considered: use of a kind of goal freezing as proposed by [Colmerauer, 1982], a static goal ordering as proposed by [Dymetman and Isabelle, 1988] or dynamic ordering (based on checking the instantiation of semantics of the candidate daughters) as proposed by Wedekind [1988].

Bottom-up generators do not have termination problems with the rule recursiveness since whatever lexical information will be needed, this is available from scratch. Yet, they present other difficulties to be briefly mentioned below.

Most bottom-up generators based on Shieber's algorithm [Shieber, 1988] use a deduction controlled by an Earley-type chart structure. While this approach works nicely in parsing, with non-determinism limited by the data -driven conduct of the algorithm, to reverse it for generation (that is to make it goal -driven, applying the rules bottom-up and keeping the results in the chart) is not straightforward and might be the source of high computational inefficiency.

Shieber solved this problem by imposing on the grammar the semantic monotonicity condition. This restriction asks that the semantics of each daughter node of a rule subsumes some part of the semantics of the mother node of that rule. Apparently, this is a natural restriction in dealing with a compositional semantics approach (each sub-component of a phrase is supposed to contribute the semantics of the phrase), but on a closer look it reveals severe limitations with respect to the coverage of naturally occurring texts: phrasal verbs (*call* some friends *up*), idioms (*kick the bucket*), expletive constructions (*it is* ...), etc.

Also, the adaptation of the Earley -type of deduction for generation purpose, is not as efficient as it is in parsing, because the left-to-right scheduling of the algorithm (more structure -oriented than meaning -oriented to the string) imposes some delayed decisions. A notorious example is the incertitude on some attribute values (such as *case*) for an NP preceding the head- verbal phrase.

Most of the (surface) generators we are aware of, construct explicitly or implicitly a syntactic tree (with specific syntactic or semantic restrictions) that the underlying grammar accepts, and according to which the syntactic validity of the output string is provided. Depending on how this syntactic tree (whether virtual or real) is constructed, a traversal order is defined over the tree (the most usual, but not the only possible, are depth-first and left-to-right or breadth-first and left-to-right). However, as we have seen, for generation purpose, a different traversal order would be needed: one that should be goal- driven but which will also consider the information in the lexicon whenever this would be necessary. Certainly, this is particularly relevant to lexicalized grammars.

4. Head-driven Generation

A straightforward way to meeting the requirements stated at the end of the previous Chapter would be the use of information in the logical form of the generator's input in a top-down way of control, to effectively produce partial sub-strings (assembled in the final output) in a bottom-up manner, controlled by

Generator's input: category A and its logical form (LF)

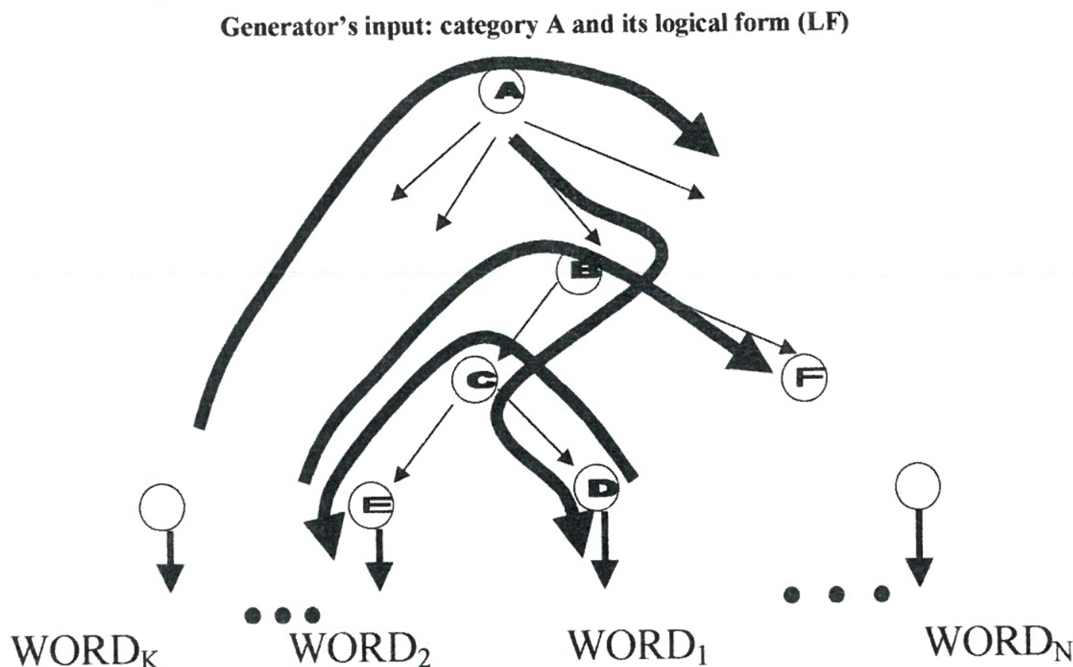

Figure 2. The final N words text is not generated left to right

the information available in the dictionary. If put this way, it is obvious that the actual segments of texts will be produced from neither left-to-right nor right-to-left, but in a combined way.

Figure 2 tries to suggest a head-driven approach of (surface) generation of natural language. So, providing the generator with a semantic representation of the intended output (LF) and a grammatical category to start with, the process will go downwards until a lexical entry (corresponding to the head of the syntactic structure to generate) is found, to subsume the input information. In Figure 2, this lexical entry (a particular grammar rule with no right-hand side) is labelled as D.

The result of unification (restrictions from the input LF and from the lexical entry) will be percolated upwards into the mother-node C. From there, C's non-head daughters are one - by- one submitted to the generation procedure. The generation of each daughter of C will contribute to further restricting the feature-structure of C (that is adding more information).

In the following, we will exemplify our implementation with a simplified version of the SMALLTALK code and therefore, it is assumed that the reader has some general knowledge of the object programming paradigm.

5. The Dictionary and the Grammar

The generator does not commit itself to a particular linguistic theory, but relies on a unification-based formalism. However, the lexical entries and the grammar rules which exemplify the functioning of the surface generator are encoded using an HPSG-flavoured notation, but once more we need to emphasize the idea that the basic functionality of the generator is not dependent on this formalism. The indexes showing the structure-sharing information are represented in the notation @n: followed by a feature structure FS and as @n for any reference to FS. A feature structure may contain only references (@n) that means that all co-indexed references refer to the same unbound feature structure.

The top -level structure of a lexical entry is shown in Figure 3.

phon	<a string>
syn	<a featurestructure>
sem	<a sem - featurestructure>|<an atom>

Figure 3. The Top-Level of A Lexical Entry

The features *phon* and *reln* take atomic values (strings of characters, representing the graphemic form of the word encoded by the entry and the name of the semantic predicate underlying the basic meaning of the word in case).The feature *syn* is non-atomic, having feature structures as values. In principle, *sem* takes non-atomic values, but in this current version we allowed also atomic values in order to let the generator work with simpler grammars and lexicons. Given that the *sem* feature is obligatory for all words in the lexicon, usually atomic *sem* values are associated with the description of functional words and constants (such as proper names).

The general structure of a *sem* feature structure is as follows:

reln	semantic_predicate
< argument1 - name >	< argument1 - value >
...	
< argumentn - name >	< argumentn - value >
qstore	< list >
conx	< list >

Figure 4. The General Structure of A *Sem* Feature-Structure

For instance, the semantics of the words in the paradigmatic family of *hate* is represented by the following structure: (reln hate hater @1 hated @2)

For building lexical entries, the system object Lex must be sent the method withValues with a FeatureStructure object as parameter as shown below.

```
A := Lex withValues #(phon beautiful
                      syn (head (mod agr @1 def indef )
                           subcat  (first  (syn  (head  (n  agr  @1  def
indef) )
                                            sem @2)
                           rest NIL))
                      sem (reln beautiful arg @1)).
```

The next examples show the skeletons of the lexical entries for the words *loves, I* and *Julie*, based on which the generator, with an appropriate grammar, is able to produce the strings: "I love Julie" or "Julie falls in love" from the logical structures love(i, julie) and love(julie) respectively.

```
(phon loves
 syn (head (v vform (finite tns prs)
      subcat (first (syn (head (n agr (per a3 num sg)))
                          sem @1)
                    rest (first (syn (head (np)) sem @2)
                           rest NIL)))
 sem (reln love lover @1 lovee @2)).
(phon "falls in love"
  syn (head (v vform (finite tns prs))
       subcat (first (syn (head (np agr (per a3 num sg)))
                          sem @1)
                    rest NIL))
  sem (reln love lover @1)).
(phon I  syn (head (n agr (per a1 num sg)))  sem i)).
(phon Julie  syn (head (n agr (per a3 num sg))) sem julie)).
```

Grammar is an ordered set of rules (OrderedCollection). A rule is a feature structure pretty much the same as a lexical entry, having an additional (built-in) attribute called *dtrs* (daughters) which, in its turn, contains three (built-in) attributes: *l_dtrs* (left-daughters), *head_dtr* (head-daughter) and *r_dtrs* (right -daughters). *l_dtrs* and *r_dtrs* have as values ordered lists of feature structures (they may be empty), corresponding to the daughters of the rule that precedes and respectively follows the head -daughter. If the value of a list of daughters is an empty list, then the corresponding attribute is, as usual, omitted. The *head_dtr* has as value a feature structure, describing the head -daughter of the rule, that is the one that shares the **sem** value with the mother node.

Two rules, corresponding to the more familiar format S→X VP and, respectively VP→VP X^+, are shown in Figure 5 (the notation {A . B} represents a list of which first element is A and the rest is B):

```
R1: (syn (head (s))
     sem @1
     dtrs ( l_dtrs @2
            head_dtr  (syn  (head
(vp vform (finite)) subcat @2)
                      sem @1))).
R2: (syn  (head @1 subcat  @2)
     sem  @3
     dtrs   (head_dtr (syn (head
@1:(vp  vform  (finite))  subcat
{@2 . @4}) sem @3)
           r_dtrs @4))).
```

Figure 5. Two Grammar Rules

A new rule is added to the grammar by sending to the system object Rule the method withValues with a FeatureStructure object as parameter as shown below.

```
A := Rule withValues: #(syn (head (np def indef @1 nil))
                        sem @2
                       dtrs(r_dtrs @3:(first(syn(head(np def indef agr
@1))
                                     sem @4)
                                   rest NIL)
                  head_dtr (phon @5
                            syn (head (mod)
                               subcat (first @3   rest NIL))
                            sem (reln @5 arg  @4)))).
```

It is important that the generator shall actually distinguish among different words in the lexicon and find all the rules that are applicable in a specific point of the generation. Given that the generator is driven by the semantics of the generation goal as well as by the lexical items that will be inserted in the output string, the dictionary (object of the type OrderedCollection) was indexed over the *sem* attribute. The dictionary accessor, withSemantics: aSem, accepts as search key (aSem) either an atomic value (as *i* and *julie* in the previous examples) or a proper *sem* structure (containing the *rel* attribute as in the example below:

```
(sem (reln love lover @1 lovee
@2)).
```

The lexicon semantic index is a hash- code table, which, if given a *sem* value, returns all the lexical entries whose semantics *strongly unifies* with it. *Strong unification* as implemented in our generator, requires that two unifiable feature structures have the same features the values of which should be pair-wise unifiable. Therefore, under *strong unification* the feature structure (Feature1 x Feature2 y) would not unify with (Feature1 x). From this point of view, the *sem* feature structure is interpreted as a predicate and therefore two *sem* structures with the same *reln* value but with a different number of features, would be considered distinct and for ever nonunifiable. Except for the lexical access, the generator uses the usual (and more permissive) unification which assumes only non-conflicting values for the same atttributes.

Grammar is searched for applicable rules on the basis of a head -feature structure. The grammar accessor, withHead: aHead, returns all the rules of which head unifies with the key (aHead)..

In the current version of the system, the lexicon contains all the inflectional variants of a given lemma. Therefore, the value of a key in the semantic index is usually a list of feature structures corresponding to the members of the inflectional family of the lemma or its synonyms. The list of lexical entries returned by the semantic index will be filtered out during the generation process due to unification failures (for instance a plural form will be filtered out by a unification with a feature-structure waiting for a singular form). However, with synonyms, this syntactic filtering will not work and here is one point where additional information source is needed to make a choice.

6. The Generator

The core of the generator is implemented by the methods **generate, generateDaughters** and **upTowards:havingGenerated:**, which operate under co-routine regime.

To keep track of the potential multiple solution in the generation process, we have implemented a specialized object called *StringVariants* (which is a type of chart, containing all possible sub-strings generated towards reaching the final solution), and an appropriate method (called **variance**) which adds new sub-strings and combines (adding to the left and to the right) partial chunks of text into larger ones.

The generation procedure is initiated by looking up in the lexicon for the entries that could serve as *generation pivots* for the input goal, that is the entries whose *sem* values unify with the semantics of the current goal. In case the lexicon contains several such entries, all of them will then be selected for further processing (given that the words in the lexicon are indexed over their *sem* feature, all the potential pivots are collected during one lexicon look-up).

The unification underlying this implementation is a destructive one, i.e. structure modifying. However, in order to avoid permanent data modification (grammar and lexicon), original data are copied before using them. This way the algorithm could considerably be accelerated because a lot of frequent tests and auxiliary data structures needed to keep the language resources unmodified, became superfluous. As a side -effect, this method eliminates the potential incompleteness of the generation algorithm (we will address this issue later). A successful unification operation modifies the arguments as a side-effect. The return value is a *block of assignments* which, when executed, will recreate the original arguments. This value is useful for undoing some modifications that led to a dead-end and for resuming the computation from other possible decision points.

```
generate
      | pivots result self_copy
|
      result   :=  StringVariants
new.
      pivots   :=  self   lexicon
withSemantics: (self at: #sem).
      pivots do: [ :p |
            self_copy   :=   self
copy.
            result addAll:
                  (p  upTowards:
self_copy havingGenerated:

                  (StringVariants
                  with:  ((p  at:
                  #phon)    value
                  asString)))].
      ^result
```

With a copy of the initial goal feature structure and each of the potential goals, the generator proceeds bottom-up (the method upTowards:havingGenerated:) looking for appropriate rules that will be applicable to the selected pivot. As in Shiber's and van Noord's generators, the pivot plays the role of a semantic head of the grammar rules, that is the daughter of a local tree (representing one grammar rule) whose semantics is identical (structure- sharing) to the semantics of the top of the tree (the mother node, or the left -part of the rule).

The potential multiple generated texts are collected and appropriately stored in the *result* object which is returned as the output of the *generate* method.

```
upTowards: top havingGenerated: someStringVariants
      | rules result left right all |
      (self subsumed: top) isNil
        ifFalse: [^someStringVariants]
        ifTrue:
          [result := StringVariants new.
            rules := self grammar withHead: self.
            rules do:
              [:r | left := r lefts isNil
                      ifTrue: [StringVariants with: '' ]
                      ifFalse: [r lefts generateDaughters].
                      right := r rights isNil
                      ifTrue: [StringVariants with: '' ]
                      ifFalse: [r rights generateDaughters].
                            all := (left variance: someStringVariants)
variance: right.
                               result addAll: (r mother upTowards: top
havingGenerated: all)
          ] ].
      ^result.
GenerateDaughters
    "generates the text corresponding to a list of daughters"
         self value == #NIL
      ifTrue: [^StringVariants with: ' ']
      ifFalse: [^(((self at: #first) generate)
                 variance:
                 ((self at: #rest) generateDaughters))].
```

When the top- goal subsumes the input pivot - (self subsumed: top) -, the recursion in generation is stopped (the recursion takes place via generateDaughters method). This condition test reaches at the top of the generation tree (obtaining a string accepted by the grammar), and also checks whether all the information provided in the initial goal has been used. The method simply returns the text that has been assembled by previous computations.

The indexing over the grammar rules allows the identification of the rules having a head-daughter that unifies with the input pivot. The mechanism used in our implementation is similar to the *link* (precompiled) predicate used in van Noord's BUG1 generator.

So, if the top goal has not been reached, the relevant rules for the current goal are collected, and each of them (rules := self grammar withHead: self.) is recursively applied for generating an alternative segment of text (rules do:[:r |...]). With the text corresponding to the head -node generated at the previous step, the **generate** method is sent to each of the left -daughters (processed in the order given by the precedence order specified in the rule) getting the text(s) that is (are) to be inserted to the left of the text(s) generated for the head -node (left := r lefts isNil ifTrue:[StringVariants with: ''] ifFalse: [r lefts generateDaughters]). The same , the generator computes the text(s) to be inserted to the right of the generated(s) from the head- node (right := r rights isNil ifTrue:[StringVariants with: ''] ifFalse:[r rights generateDaughters]). Finally, the texts corresponding to the left -daughters, the head-daughter and the right -daughter are assembled (all := (left variance: someStringVariants) variance: right) and the mother of the currently processed rule is set as a new pivot in the bottom-up proceed of the algorithm (r mother upTowards: top havingGenerated: all). The methods **left**, **right** and **mother** are selectors which, if sent to a rule, return the list of feature-structures for the left -daughters, the list of feature-structures for the right- daughters and, respectively the feature-structure of the mother -node pertaining to the rule concerned. If the rule has no left- or right- daughters, the corresponding segment of text would be the empty string.

As any head-driven algorithm, the present one has no problem with the left -recursiveness of the grammar rules. Given that this is argued at length elsewhere [van Noord,1988], [Shieber, 1988] we will not enter into more details.

For building generation goals, one should send the system object Generator the constructor method withValues with an appropriate feature

structure (that is the one which contains at least the *sem* attribute).

The basic assumption here is that whatever goal is given as input to the generator, there will be at least one lexical entry the semantics of which would unify with the semantics of the input. Otherwise, the generation procedure will produce an empty string. Seemingly, this condition raises the same problems as Shieber's grammar monotonicity requirement, but in fact this is not true. The insertion of words which are stipulated either lexically (as in the case of particles or idioms), or syntactically (for instance, the expletive expressions) is always possible given that the stipulation is done in terms of **sem** restrictions. For instance, the lexical entry [2] will be selected when the generator is invoked with a goal such as *die(john)* and the idiom *kick the bucket* is eventually generated. The non-idiomatic *kick* will not be considered since its semantics (reln kick kicker @1 kickee @2) does not unify with the initial goal.

```
(phon kicked
 syn (head (vp vform (finite tns past)
      subcat (first (syn (head (n agr (per a3 num sg))) sem @1)
                     rest (first (syn (head (n def definite agr (per
a3 num sg)))
                                  sem bucket))
                      rest NIL))))
 sem (reln die defunct @1)).
```

Thus the sequence below will yield two sentence-strings licenced by the grammar and the lexicon:

```
A := Generator withValues:  #(syn (head (s)
                              sem (reln die
                                   defunct john)).
A generate. → StringVariants('John died' 'John kicked the bucket')
```

From among the problematic issues connected with the head-driven generation algorithm, both van Noord and Shieber discussed Wedekind's notions of coherence and completeness of a generator [Wedekind, 1988]. In its original form, Shieber's algorithm is both incoherent (that is overgenerating for under-specified input -unbound variables- with a non-terminating threat) and incomplete (that is, undergenerating by leaving out some arguments of the input logic). A later version of Shieber's generator adopted van Noord's solution from BUG1: using *numbervars* to *freeze* the unbound variables in the input logical form (for ensuring coherence) and checking the subsumption of the final generated feature structure(s) by the initial input goal (for ensuring completeness).

The specific solutions adopted in BUG1 were imposed mainly by the implementation of the underlying PROLOG unification.

The under-specification of *sem* structures does not create special problems in our generator because, as we said before, the *sem* structures are interpreted in a predicate-like manner. However, one could produce an underspecified *sem* structure which will not be filtered out by the lexical look-up, namely proving a nil (unbound) value for one or more features:

a) *sem (reln love lover john lovee nil)*

As these features are supposed to be referenced in the *syn* structure of the selected entries, the applicable grammar rules will eventually need

---

2 A much simpler solution for generating the idiomatic expression would have been the use of a simplified entry as the one given below but we wanted to exemplify the proper lexical insertion, that is one that would pick up whatever stipulated insertion from the lexicon:

```
(phon kicked_the_bucket
 syn (head (vp vform (finite tns past)
      subcat (first (syn (head (n agr (per a3 num sg))) sem @1)rest NIL))
 sem (reln die defunct @1)).
```

these unbound features and, if they fail, in the end the whole generation will fail. So they will always produce an empty string.

There might be, however, a real need to under-specify a certain feature. In such a case the generator allows a "don't care" value for any attribute (_) which is recorded in the lexicon and has an empty string as the value for the *pron* feature. Therefore, if the intended text is *"John loves", then the proper *sem* structure of the generation top goal should be in *sem (reln love lover john lovee _)*.

Let us assume that the dictionary contains the following entries:

```
Entry1:
(phon loves
 syn (head (v vform (finite tns prs)
      subcat (first (syn (head (n agr (per a3 num sg)))
                          sem @1)
                   rest (first (syn (head (np)) sem @2)
                          rest NIL)))
 sem (reln love lover @1 lovee @2)).
Entry2:
(phon falls_in_love
 syn (head (vp vform (finite tns prs))
       subcat (first (syn (head (np agr (per a3 num sg)))
                          sem @1)
                  rest NIL))
 sem (reln love lover @1)).
```

With these lexical entries, asking the generator to produce a text from the following semantic specification (*sem love lover john)* only Entry2 will be retrieved from the lexicon (due to a strong unification used by the lexical look-up). Based on it, the generator will produce the string "John falls in love".

On the other hand, if the generation will start with the semantic specification (*sem love lover john lovee_)* the actual generator will produce not the acceptable sentence "John falls in love" (as one might expect) but the questionable one *"John loves".

Completeness is the property of the algorithm which makes sure that all the information specified in the input is included in the generated output, that is when fed with a semantic representation like *sem (reln eat eater John food pie),* the generator should not provide as a result *John eats,* but *John eats a pie*. This property of the algorithm is achieved by the same copying and final subsumption mechanism as in BUG1.

The proper generation is launched by constructing a generation object (sending the system object *Generator* the constructor method `withValues` with a `FeatureStructure` object as parameter) and sending to it the method `generate`.

```
Generator withValues: #(syn (head (s))
                        sem
(reln love

lover i

lovee julie)) generate.
➔    I love Julie.
Generator withValues: #(syn (head (s))
                        sem
(reln love

lover julie

lovee NIL)) generate.
➔    Julie falls in love.
```

7. Conclusions and Further Work

The generator presented so far has been tested on very small grammar and lexicon. More realistic language resources are needed for a full evaluation of the generator (both linguistic and computational) performance. Strategic aspects of generation, that is how and out of what a ***sem*** feature structure as required by the generator is constructed, have not been discussed. Also, we have not discussed the generation of multi-sentence natural text, planning the text structuring. A general solution to the strategic generation, if possible, requires further investigation, but it is very likely that a useful one (especially from a computational performance point of view) considers the client-orientation of the surface generator, at least its knowledge representation and domain -specific knowledge.

Acknowledgments

The work reported here was supported by a NATO grant and done at CNRS/LIMSI, Orsay, France. It was part of a joint project carried out together with Michael Zock, and many of his stimulating ideas were included in this surface generator. Mihai Ciocoiu wrote an initial

Smalltalk version of the generator and most of the test grammar and dictionary.

Joseph Mariani and Joseph Sabah deserve special thanks for making my stay at LIMSI extremely pleasant and stimulative.

REFERENCES

AHO, A.V. and ULMAN, J. D., **The Theory of Parsing, Translation and Compiling**, PRENTICE -HALL SERIES IN AUTOMATIC COMPUTATION, 1972.

COLMERAUER, A., **PROLOG II: Manuel de réference et modèle théorique**, Technical Report, Groupe d'Intélligence Artificielle, Marseille, France, 1982.

DALE, R. and MELISH, C., **Towards Evaluation in Natural Language Generation**, Proceedings of the 1st International Conference on Language Resources and Evaluation, Granada, Spain, 1998, pp. 555-562.

DYMETMAN, M. and ISABELLE, P., **Reversible Logic Grammars for Machine Translation**, Proceedings of the 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages, Pittsburgh, PA, USA, 1988.

FAWCET, R. P. and TUCKER, G. H. , **Prototype Generators 1&2**, Report No.10 of COMMUNAL, Computational Linguistic Unit-University of Wales College of Cardiff, 1989, 30p.

FELSHIN, S., **A Guide to the Athena Language Learning Project Natural Language Processing System**, copyright, Massachusetts Institute of Technology, USA, 1993.

HAMBURGER, H., TUFIS, D. and HASHIM, R., **Structuring Two-medium Dialog for Learning Language and Other Things**, Proceedings of the Association for Computational Linguistics Workshop on Intentionality and Structure in Discourse Relations, Columbus, Ohio, USA, June 20, 1993.

MC DONALD, D., **Description Directed Control: Its Implications for Natural Language Generation**, COMP. & MAT. WITH APPLS. Vol. 9, No. 1, 1983, pp. 111-129.

MC DONALD, D., **Issues in the Choice of A Source for Natural Language Generation**, COMPUTATIONAL LINGUISTICS, Vol. 19, No. 1, 1993, pp. 191-197.

POLLARD, C. and YOO, E. J., **Quantifiers, Wh-phrases and A Theory of Argument Selection**, HPSG Workshop, Tübingen, Germany, 1995.

POLLARD, C. and SOG, I. A., **Head-driven Phrase Structure Grammar**, CSLI PUBLICATIONS and UNIVERSITY OF CHICAGO PRESS, 1994.

REITER, E. and DALE, R., **Building Applied Natural Language Generation Systems**, NATURAL LANGUAGE ENGINEERING, No.3, 1997, pp.57-87.

SHIEBER, S. M., VAN NOORD, G., MOORE, R. C. and PEREIRA, F.C.N. , **A Semantic Head-driven Generation Algorithm for Unification-based Formalisms**, Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver, Canada, 1989.

SHIEBER, S., VAN NOORD, G., PEREIRA, F.C.N. and MOORE, R.C., **Semantic Head-driven Generation**, COMPUTATIONAL LINGUISTICS, Vol.16, No.1, 1990, pp. 30-42.

SHIEBER, S. M., **An Introduction to Unification-based Approaches to Grammar**, Lecture Notes CSLI, No. 4, 1986.

SHIEBER, S.M., **A Uniform Architecture for Parsing and Generation**, Proceedings of the 12th International Conference on Computational Linguistics, Budapest, Hungary, 1988.

SHIEBER, S. M., **The Problem of Logical Equivalence**, COMPUTATIONAL LINGUISTICS, Vol. 19, No. 1, 1993, pp. 179-190.

THOMPSON, H., **Strategy and Tactics: A Model for Language Production**, papers presented at the 13th Regional Meeting, Chicago Linguistic Society, Chicago, Illinois, USA, 1977, pp. 651-668.

TUFIS, D., HAMBURGER, H., HASHIM, R. and PAN, J., **Generation of Natural Language in An Immersive Language Learning System**, in M. Browse (Ed.) Basics of Man-Machine Communication for the Design of Educational Systems, SPRINGER-VERLAG, 1994.

TUFIS, D. and ZOCK, M., **Maximal Cohesion, Minimal Commitment and Lexical Pattern Matching**, LIMSI Research Report, December 1997, 56p.

VAN NOORD, G., **BUG: A Directed Bottom Up Generator for Unification Based Formalisms**, Utrecht-Leuven Working Papers in Natural Language Processing, 1989.

WEDEKIND, J., **Generation As Structure Driven Derivation**, COLING '98, Budapest, Hungary, 1998.

WILCOCK, G. and MATSUMOTO, Y., **Head-driven Generation with HPSG**, Proceedings of COLING-ACL '98, Montreal, Canada, August 1998, pp. 1393-1337.

ZOCK, M., **The Power of Words in Message Planning**, Proceedings of COLING'96, Copenhagen, Denmark, 1996, pp. 990-995.