Text Generation

Michel Boyer Guy Lapalme

September 1989

12.1 Introduction

Writing texts involves many complex cognitive processes. We are all aware of that because, for many of us, writing is not always an easy task and our ability to efficiently convey information needs education and experience to be developped. Is it then possible to automatically generate non trivial good quality texts by other means than predefined formats or templates? This chapter argues that it can be done and shows some examples.

Automatic text generation is often divided into two distinct parts: the "what to say" module that extracts the important facts from all the available information and the "how to say" module that finds appropriate rhetoric techniques or syntactic choices in order to express the information previously selected. Though it has been argued that such a separation is artificial it is almost always present in some way or another in actual text generation systems. In most applications, it is the referential function of the text that prevails and that will be our only concern in this chapter. That is obviously a limitation since, for example, in poetry we may be more interested in the effect on the hearer or on some aspects of the communication channel than on the message content.

We do not address those important issues here. Our approach is practical: we introduce and illustrate basic principles through working Prolog programs. We first show how Definite Clause Grammars (DCG) can be used for generation in simple cases where only the "how to say" part is important and also when some non trivial processing is necessary to find out "what to say". We then illustrate how a Prolog implementation of a functional grammar allows a "rational reconstruction" of *Ana*, a system for generating stock exchange reports. After that, an alternative way for generating text is given using a semantic network covering approach that we developped in the context of the Meaning Text Model. We then give an example of a system for generating questions for a medical information system. These systems illustrate how logic programming can be used for text generation either to represent and manipulate the basic informations or to describe the grammar of the generated text.

While there are many well-established methods for parsing and building semantic structures from a natural language sentence, nothing equivalent exists for text generation. Why? One of the main reasons is surely that parsing starts from a well defined state: a natural language string. On the other side, the input of a text generator can be anything from a parsing tree, a logical formula, a semantic graph, an expression in a data-base query language, a series of logical deductions or even raw data like those we can get from the "Dow-Jones News Service" database. For parsing, we have the dual problem: the starting point is precise but complexity varies with the "deep structure" sought.

Another reason for this seeming lack of interest in generation stems from the fact that much effort was necessary to make humans understandable by computers, with natural language query systems for example, whilst humans can still make do with rudimentary outputs, tabular forms for instance. However as users get used to higher standards they become more demanding and the basic technique of *canned-text* is not sufficient any more. There are even applications where natural language has to be used; summaries or cooperative answers are examples of such advanced uses of text generation.

Mann (1982) was one of the first general survey on the topic of natural language generation. McDonald (1983) , McKeown (1985) , Danlos (1985) and Appelt (1985) describe interesting approaches to this problem. McDonald and Bolc (1988) have brought together a collection of recent and interesting works.

12.2 Random generation

At the most simple level, grammars can be used to freely generate sentences without any constraint on meaning. That can be useful as a tool for "debugging" a grammar or for generating personalized sentences to be used in spelling or grammar drills.

Here follows a Prolog program that generates such random strings; the sentences are not very informative but they show how a DCG can be used to get a sentence skeleton containing information that is afterwards managed by the morphology component in order to get a readable sentence.

The phrase structures are chosen using predicates that succeed or fail with a given probability. We use the following evaluable predicates: `random/1` returns a random floating point value in $[0, 1)$ and `random/2` returns an integer value in $[1, N]$ where N is the first parameter. In order to use these random numbers in a more intuitive way in grammar rules, we have defined two operators: the prefix operator `?/1` succeeds with probability of $1/n$ and the infix operator `?/2` chooses a random element in a list. We give a $1/n$ probability to a clause of a predicate where n is the number of remaining clauses of the same predicate[1]. DCGs build the sentence structure containing words with possibly free logic variables that later take values when some more information becomes available. We see that, as in the case for parsing, the DCG rules are only a transcription of the grammar rules. In this way, for example, the number of a noun phrase `np` can depend on the number of the noun even if the latter occurs after the determiner and the adjective. Once the structure with logic variables appropriately linked together is built, it is given to a morphology component that writes the "right words".

```
s --> np(Number), vp(Number).

np(Number) --> [det(Number)],ap,[noun(Word,Number)],
               {[cat,mouse] ? Word, [singular,plural] ? Number}.

vp(Number) --> ?2,[vrb(V,3,Number)], {[eat,run,love] ? V}    ;
               ?1,[vrb(V,3,Number)],np(_), {[eat,love] ? V}.

ap --> ?2,[adj(A)], {[fussy,grey,nervous]?A} ;
       ?1, [].

generate :- s(X, []), write(X),nl, morphology(X,Y), write_the_words(Y).
```

We now give a few examples of the generated structures and sentences by calls to `generate`:

```
[det(plural),noun(mouse,plural),vrb(love,3,plural),
 det(plural),adj(nervous),noun(mouse,plural)]
mice love nervous mice

[det(plural),adj(fussy),noun(cat,plural),vrb(eat,3,plural),
```

[1] With these parameter values, we thus choose uniformly between the clauses because the ith clause (starting from the "bottom" of the predicate clauses) will then have as success probability

$$\left(1-\frac{1}{n}\right)\left(1-\frac{1}{n-1}\right)\cdots\left(1-\frac{1}{i+1}\right)\frac{1}{i} = \frac{n-1}{n}\frac{n-2}{n-1}\cdots\frac{i}{i+1}\frac{1}{i} = \frac{1}{n}$$

because it is the product of the failure probabilities of the preceding clauses with the success probability of the current clause

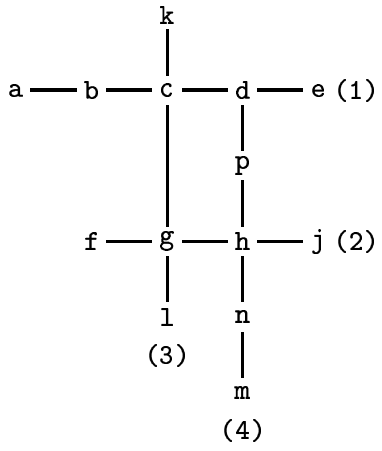
```
 det(singular),noun(mouse,singular)]
fussy cats eat a mouse

[det(plural),adj(nervous),noun(mouse,plural),vrb(eat,3,plural),
 det(singular),adj(fussy),noun(mouse,singular)]
nervous mice eat a fussy mouse

[det(singular),noun(mouse,singular),vrb(eat,3,singular)]
a mouse eats
```

12.3 A small application

Here is another illustration of the power and flexibility of Prolog for all aspects of text generation including "what to say". Suppose we want to give instructions on the the fastest way to go from `a` to `b` using the subway. The information about the subway network is given as Prolog clauses and `find_path(A,B,Path)` computes the shortest path tree from `A` to `B` using Dijsktra's algorithm and `find_lines/3` extracts the information about which lines to follow and for how long. Given the following network:

```
%%%
%%%   Network definition in terms of lines
%%%     line(linenumber,list of station names and distances between them)
%%%
line(1,[a,1,b,1,c,1,d,1,e]).
line(2,[f,1,g,1,h,1,j]).
line(3,[k,1,c,1,g,1,l]).
line(4,[m,1,n,1,h,1,p,1,d]).
```

it corresponds to the following network, the lines numbers are in parenthesis.



To find a way from `b` to `j`, we call

```
| ?- find_path(b,j,Path),find_lines(Path,Dist,Lines).
```

```
Path = [(j,15,2,h),(f,13,2,g),(m,11,4,n),(n,10,4,h),(h,9,4,p),
        (l,8,3,g),(p,8,4,d),(k,7,3,c),(g,7,3,c),(e,3,1,d),
        (d,2,1,c),(c,1,1,b),(a,1,1,b),(b,0,1,b)]
Dist = 15
Lines = [b,(1,2,2),d,(4,2,2),h,(2,1,1),j]
```

`Path` gives the shortest path tree taking into account a penalty of 5 units when there is a line change. Now this information can be used by the text generation module. DCGs allow us to integrate lexical, syntactic and even semantic processing within the grammar rules. We use the same kind of approach to achieve a variation in the output and for expressing the different alternatives. Here for simplicity and for showing the basic mechanisms and flexibility given by the DCGs for text generation, we have used very simple sentence templates, but the next examples show a much more sophisticated approach to find sentence structures.

```
:- reconsult('shortest_path.pro'),
   reconsult('randomdcg.pro').

%%%
%%%   Sent will be a list of words describing how to get from X to Y
%%%
go(X,Y,Sent) :-
        find_path(X,Y,Path),
        find_lines(Path,Dist,Lines),
        generate(Lines,Sent,[]).

%     A simple sentence when the destination is only a line away
generate([Orig,(Line,Nb_stations,Len),Dest]) --> !,
        (?2, ['This',is,easy] ;
         ?1, ['You',are,lucky,you,are,only,Nb_stations,St1,away],
             {sing_plur_st(Nb_stations,St1)}),
        (?2, [take,line,Line,towards,Direction]
           , {find_dir(Line,Orig,Dest,Direction)} ;
         ?1, [follow,line,Line,for,about,Len,minutes]).

%      A more complex case divided in three phases
generate(Path) -->
        introduction(Path,Path1),
        body(Path1,Path2),
        conclusion(Path2),!.

%      Boarding and following the first line
introduction([Orig,(Line,Nb_stations,Len),Dest|Rest],[Dest|Rest]) -->
        (?2,['First',take,line,Line,for,Nb_stations,St1] ;
         ?1,['Board',on,line,Line,towards,Direction]),
        (?2,[until,you,are,at,Dest] ;
         ?1,[up,to,Dest]),
        {sing_plur_st(Nb_stations,St1),
```

```
         find_dir(Line,Orig,Dest,Direction)}.

%      Describing all the lines until the last one
body([Orig,L,Dest],[Orig,L,Dest]) --> !.
body([Orig,(Line,Nb_stations,Len),Dest|Rest],Rest1) -->
        (?4,[after,that,take,line,Line,for,about,Len,minutes,up,to,Dest] ;
         ?3,[then,change,on,line,Line,until,Dest] ;
         ?2,[and,transfer,for,Dest,on,line,Line] ;
         ?1,[';',switch,on,line,Line,for,Nb_stations,St1],
            {sing_plur_st(Nb_stations,St1)}),
        body([Dest|Rest],Rest1).

%      Saying that it is the end of the trip
conclusion([Orig,(Line,Nb_stations,Len),Dest]) -->
        (?3,[finally,you,should,take,line,Line,to,arrive,at,Dest] ;
         ?2,[at,last,you,will,find,station,Dest,on,line,Line] ;
         ?1,['. Station',Dest,will,be,just,Len,minutes,away,on,line,Line]).


%%%
%%%  utility predicates for computing information
%%%

%     find the first or last station in the line giving the direction
find_dir(Line,Orig,Dest,Dir):-
              ....          %  auxiliary definitions not pertinent here

% return ``station'' or ``stations'' depending on the first parameter
sing_plur_st(1,station) :- !.
sing_plur_st(_,stations).
```

Now let us look at a few outputs:

```
| ?- go(b,j,Sent).
Sent = [First,take,line,1,for,2,stations,until,you,are,at,d,
        and,transfer,for,h,on,line,4,finally,you,should,take,
        line,2,to,arrive,at,j]

| ?- go(b,j,Sent).

Sent = [Board,on,line,1,towards,e,up,to,d,;,switch,on,line,4,
        for,2,stations,. Station,j,will,be,just,1,minutes,away,on,line,2]
```

This simple example shows how simply we can produce texts using DCGs which are ordinarily only used for parsing. The usual examples of "reversible grammars" are simplistic and not very useful as they often suffer from a combinatorial explosion (although one should see chapter 14 for a way around this problem). In our case, we make a heavy use of semi-colon and cuts as they

allow us to commit to choices in the generation process, although as we have shown earlier it is sometimes very useful to use backtracking to reconsider some choices. Using the schema given in our programs seeing the semi-colon as separating the alternatives and the interrogation mark as numbering them, we end up with a simple translation of the grammar where other control predicates can be embedded by putting them in braces.

12.4 Generation using a unification grammar

The program that we now describe produces a well written three paragraph stock market report from raw data. It was developped following ideas of Kittredge (1983) and reproduces in Prolog outputs similar to those obtained by Karen Kukich's ANA system described by Kukich (1983) . However our system is much faster than hers and, when written in terms of feature structures, it provides a concise and readable grammar. Analog produces the full report in less than 3 seconds on an Apollo workstation and takes 15 seconds on a Mac II (running C Prolog).

Definite Clause Grammars were intended to provide a readable input grammar that could readily be translated into Prolog clauses. It is very easy to design a DCG preprocessor so that the generated Prolog rule automatically builds a parse tree. More sophisticated variants of DCGs (namely Extraposition Grammars, Discontinuous Grammars) provide additional tools to deal with movements. However, to deal with agreements or build semantic representations, the grammar writer is required to use explicit arguments in the input rules and DCG based grammars eventually loose most of their intended readability, become cumbersome and hard to modify. An alternative is to use a grammar formalism with a more simple argument structure. That is what logical functional grammars described by Boyer (1989) are intended for. Functional structures simply take the form of a list of `attribute:value` pairs like

```
[ number: plural,
  person: third ]
```

and can be specified incrementally: after some additional analysis, one can further specify that the gender is feminine. The order of the pairs is irrelevant and to unify two structures we simply merge the lists provided there is no value conflict for some attribute. For instance the most general unifier of

`[number: plural, person:third]` and `[gender: feminine]`

is `[person: third, gender:feminine, number:plural]` (or any permutation of that list); as a counterexample, the structures `[gender:masculine]` and `[gender:feminine]` do not unify.

Given two functional structures `NP` and `VP`, we use the notation

```
NP:gender === VP:gender
```

to specify that the values of the gender attribute in `NP` and `VP` are to be unified. Keeping a syntax that closely resembles that of the most simple DCG rules, we can write

```
s(S) --> np(NP),vp(VP),
    {
           NP:gender === VP:gender,
           NP:number === VP:number
        }.
```

to express the fact that the functional structures `NP` and `VP`, agree in gender and number. Nothing prevents the structures `NP` and `VP` from containing other attributes. The above rule is obviously too simplistic: for instance it specifies nothing about `S`. If we further decide to use the person and forget to mention it in the rules defining `NP` and `VP`, then an additional condition like

```
NP:person === VP:person
```

(to be added between the brackets in the above rule defining s) would not fail but simply succeed with an uninstantiated person value. Such a behavior is much more user friendly when grammar debugging is necessary than with DCGs (that will return 'no' instead of returning an underspecified structure). Boyer (1987) shows how Functional Logic Grammars can be used to provide the same functionality as PATR.

12.4.1 Planning a Report

Artificial Intelligence provides a number of tools to represent meaning; conceptual dependency structures have been used for actions, whether mental or physical. Lambda representations or logical forms are used for quantifiers and determiners. Various forms of semantic networks, including conceptual graphs, aim at providing a unifying formalism. The usefulness of those tools increases with the complexity of *what to say*. In the case of stock market reports very simple tables can be used to specify what to say. Such tables are generally used as input for highly graphical representations. The problem now is *how to say it*. Facts have to be ordered and processed by a planner; appropriate messages, encoding more precisely *what to say* than mere tables, are to be built and given as input to a generation grammar with rhetoric abilities. Producing such a report requires planning and style. That is our concern here.

The input

Facts are written as Prolog clauses allowing functional structure arguments. They take the form:

```
fact(F-Structure).
```

Half hourly statistics `[fname:hrstat]` are kept as facts. The same holds for the closing status `[fname:clstat]`, advance-decline statistics `[fname:advec]` and volume statistics `[fname:volstat]`. Here is a fact describing closing statistics (`fname:clstat`) of the Dow Jones (`iname:dji`) composite index (`itype:compos`) on June 24 (1983).

```
fact([fname:clstat,  iname:dji,  itype:compos,   date:06/24,
      hour:close,     close_level:810.41,     cumul_deg:2.76,
      cumul_dir:dn,   high_level:821.63,    low_level:805.56,
      open_level:814.69]).
```

The input files we were given (Karen Kukich's sources are in OPS5) contained 17 such facts for a given day. These files had been automatically generated from Dow Jones News Service Data.

What to say

Determining what to say is finding the semantic chunks, or messages, to build a meaningfull and readable report. As is the case with facts, messages are also represented as functional structures

`message(`*functional structure*`)`

and determining what to say amounts to generating a list of such messages. Here is the main rule of the (semantic) grammar that performs that job:

```
text_as_messages -->
                market_status,
                mixed_market,
                volume_of_trading,
                market_fluctuations,
                dow_jones_ind_status,
                dow_jones_fluctuations,
                transport_index_closing_status,
                util_index_closing_status,
                nyse_volume_level,
                advances_declines_ratio.
```

Each of the non terminals (`market_status`, `mixed_market` etc.) generates a list of 0 or more messages by fetching information in the `facts` database. Out of the 17 original facts, a list of about 8 messages is generated with the call

```
text_as_messages(ListOfMessages,[]).
```

Such a list can be seen as an unstructured text whose "words" are messages.

To better understand how that list is built and how message attributes are fetched, let us look at one of the rules defining the non terminal `mixed_market` that appears in the definition of `text_as_messages`. As usual, Prolog calls are between brackets; lists of `attribute:value` pairs are to be interpreted as functional structures. The following rule expresses the fact that there is one `mixed_market` message that is generated when the Dow Jones (`iname:dji`) closing statistics (`fname:clstat`) are up (`cumul_dir:up`) whilst the New York stock (`iname:nyse`) exchange declines (`fname:advdec`) outnumber the advances (or the converse). Else there is none. The Dow Jones and the New York stock exchange information is obviously to be fetched in the appropriate facts.

```
mixed_market -->
                {
                  fact([fname:clstat, iname:dji, cumul_dir:Dir]),
                  fact([fname:advdec, iname:nyse,
                        advances:X, declines:Y],
                  ( (Dir == up, X < Y) ; (Dir == dn, X > Y) ), !
                }
                [ message([top:genmkt,subtop:mix, subjclass:mkt,
                           mix:mixed, time:close])
                ].
mixed_market --> [].
```

The goal `text_as_messages(Messages,[])`, unifies Messages with a list that still requires some processing to organize the discourse.

12.4.2 How to say it

To organize the discourse, messages are grouped into paragraphs by topic and ordered within the paragraph. Messages can also be "merged"; for instance if the messages concerning the transportation (`trn`) and the utility index (`utl`) show the same variations, they will be replaced by a single

message whose subject is "transportation and utility index" (`trnutl`). Given that the domain is very limited, useful combinations of subjects can be hand listed.

The report grammar

The grammar that produces reports uses full phrase templates as its terminals (v.g. `the transportation and utility indicators`, or `losers among New York Stock Exchange issues`). Those templates contain variables that can take care of numeric values for instance. A similar approach had been taken by Chantal Contant (1988) in OPS5. Our Prolog based grammar is much more readable, twice shorter and more than 10 times faster though.

The underlying context free grammar of the report generator is simple.

$$\begin{aligned} report &\rightarrow (adverb \mid end\_paragraph \mid ['.'] \mid phrase), report. \\ adv &\rightarrow 'yesterday' \mid 'at\ the\ final\ bell'. \end{aligned}$$

The non terminal *report* takes as arguments the list of messages and information to constrain the ongoing text. Here is the (slightly rephrased) rule corresponding to

$$report \rightarrow phrase, report$$

```
report([Message|Rest], Constraints) -->
        phrase([Message|Rest],Constraints,PhrFeat),
        { gen_constraints(Message, Constraints,PhrFeat,NewConstraints)},
        report(Rest,NewConstraints).
```

The constraining information bears both on syntax and rhetoric. Care is taken to keep cross sentence references to a same subject in increasing level of generality. Constraints furthermore specify if the subject is still to appear in the current sentence, how many syllables can still be added to it and how many messages were conveyed by the preceding two sentences (to avoid overloading a third one)

The Lexicon

The structure of the lexicon is very simple. Two sets of phrases are used, one for subjects, the other for verbs with their complements. After a statistical analysis of the Stock Exchange sublanguage, Karen Kukich identified eight classes of synonymous subjects; here they are each with one possible lexical choice:

utl	the utility index
dow	the Dow Jones industrial average
trn	the transportation index
dec	declining stocks
trnutl	the transportation and utility index
bigbd	big board volume
mkt	stocks
adv	gainers

For each of them, the lexicon gives a number of choices with a hyponymy level (level of generality), a number (sing or plur), and a cumulative probability value. The grammar takes care to choose cross sentence coreferences in increasing level of hyponymy.

Class	Lexical choice	Hypon	Number	Prob.
dow	['the Dow Jones industrial average']	1	sing	0.4
dow	['the Dow Jones industrials']	2	sing	0.7
dow	['the industrial average']	3	sing	0.9
dow	['the index']	4	sing	1.0

Lexical entries for verbs take a similar form; lexical choices depend on the attribute values of the current message (`MessAtts`) and also the subject (`SubjClass`) (French needs an additional argument for direct object agreement). Here is the general structure of a lexical entry for a `VP` with an example:

```
SubTop(MessAtts, SubjClass, CumulProb ,Form, Verb, Np, LengthVp)
```

`MessAtts` specifies the attributes of the message needed to select that `VP`, `CumulProb` is a probability threshold (that plays the same role as Prob for the noun phrases described above), Form is either `simple` or `prepp` (prepositional phrase) and `LengthVp` is a measure of the length, in syllables, of the `VP` encoded by the lexical entry. Here is an example of such an entry:

```
close( [dir:up, deg:nil, tim:close, vardeg:Vardeg, varlev:Varlev],
       dow, 0.25, simple,
       'to close', ['at', Varlev, ', up', Vardeg, 'points'], 10).
```

That entry gives rise to the phrase "closed at Varlev up Vardeg points" where Varlev and Vardeg are integer values (or "closing at ..." if a participle is asked for). That entry will be judged acceptable by `vp` (c.f. next section) only if the `VP` can still hold more that 10 syllables. The lexicon containts 280 such entries for verb phrases. As one can see, the role of the grammar is to put together in an elegant and coherent fashion full chunks of text.

Producing the Sentences

The underlying context free grammar for generation at the sentence level can be written as follows:

$$
\begin{aligned}
phrase &\rightarrow (simple \mid coorsent \mid suborsent \mid suborpartsent \mid \\
&\quad\ subortempspost \mid subortempspre \mid prepp).\\
simple &\rightarrow np, vp.\\
coorsent &\rightarrow conj, np, vp \mid and, vp.\\
suborsent &\rightarrow [',', as], np, vp.\\
suborpartsent &\rightarrow [',', with], np, vp.\\
subortimepost &\rightarrow [',', before], vp.\\
subortimepre &\rightarrow [after], vp, [','].\\
prepp &\rightarrow vp.\\
conj &\rightarrow [and] \mid [but].
\end{aligned}
$$

The symbols `np` and `vp` directly access the lexicon as described in the previous paragraph. To guarantee a good usage of conjunctions or constrasts, the topic, subtopic and subject class of the previous message is kept in the constraining environment. Here is the *suborsent* rule that can be applied if the previous message dealt with the "big board":

```
suborsent([Mess|Rest], Constr, Phrase) -->
        {
           Constr:lastmess:subtop === bigbd,
           ?2
        },
        comma, [as],
        np(Mess,Constr,suborsent, Phrase),
        vp(Mess,Constr,simple, past, Phrase).
```

For example, once the message concerning the big board has been uttered, one can add something of the form "as advancing stocks outpaced the loosers 772 to 660". Here the `np` "advancing stocks" is to be found verbatim in the lexicon. The `vp` entry (as described in the previous section) is almost as close to the actual text that will appear in the final report.

The functional structure `Phrase` used in the previous rule takes the form:

```
[subj:ChosenSubj, number:N, length:L]
```

where `L` is an upper bound for the length of the `vp`. The rule concerning coordinate forms looks at the topic and subtopic of the current and previous message as well as to the class of their subjects.

12.4.3 The Working System

The full working version of the program described above was one of our first systems using feature unification but had a much more complex argument structure than that advocated above. The French version contains only 271 clauses for the lexicon and 141 clauses for the grammar. The full English system has 63 clauses for subjects, and 500 clauses for the verb phrase lexicon. The system is fast, robust and produces consistently well written reports; here is an example:

```
                    STOCK MARKET REPORT


Friday june 25, 1982

     Stock prices crept upward early in the session, before slid-
ding  downhill  late in the day, with the indexes edging downhill
yesterday.  Prices turned in a mixed showing in moderate trading.

     The Dow Jones industrial average rose  slightly,  to  finish
the  day  at  810.41  up 2.76 points, with the transportation and
utility indexes being up slightly.

     Volume on the big board was 55860000  shares  compared  with
62710000 shares on Wednesday, with advancing stocks outpacing the
losers 772 to 660.
```

12.5 Paraphrase generation

Tables are often not enough to capture the meaning of a simple declarative sentence. When the complexity of what to say increases, networks are often used to represent meaning. We first show

how unification and covering methods can be used to generate syntactic structures from a net, allowing *semantic* paraphrasing on backtracking. Then we look at lexical transformations as a method of syntactic paraphrasing. The approach is a formalization of methods borrowed from Mel'čuk's Meaning Text linguistic model.

The aim of the Meaning Text Model (henceforth MTM) is to establish correspondences between meanings, represented by networks, and texts. MTM is made of several components and representation levels: semantic, deep syntax, surface syntax and morphology. The MTM can be described by the procedure

```
mtm(SemRep, Text) :-
    sem_comp(SemRep,DeepSynt),
    deep_synt_comp(DeepSynt, SurfSynt),
    surf_synt_comp(SurfSynt, DeepMorphology),
    deep_morph_comp(DeepMorphology, Text).
```

According to Mel'čuk, all those predicates should be reversible. Due to combinatorial problems, our system can only generate (i.e. it works if `SemRep` is instantiated and `Text` is a free variable).

Both Deep and Surface Syntax are represented by dependency trees, i.e. trees with labeled arrow. Labels of the deep syntax are taken in the set `1`, `2`, `3`, `4`, `attr`, `coord`, `append` where numeric labels indicate arguments, `attr` an attributive relation, `coord` a coordinate construction and `append` is used for parentheticals, interjections ...

Two trees can be merged at the root provided there is no duplication of numeric labels. Such a restriction reminds us of feature structure unification and, in fact, dependency trees can be represented by a term with a functional structure argument:

```
tree(Root,[Label1:Sub_tree1, Label2:Sub_tree2 , ... Labeln:Subtreen]).
```

provided functional unification is modified to allow multiple copies of `attr`, `coord` and `append`.

To build the `DeepSyntax` from the network, we use a dictionary relating subnet patterns and associated deep syntax structures. Those patterns can be interpreted as *meaning molecules*. Once the semantic network has been covered by such molecules, we simply merge their associated deep structures to get a full deep structure of the sentence. Semantic paraphrasing occurs when many coverings are possible.

The merge involves trees that do not have the same root. To do it we first define the procedure `sub_tree(X,Y)` which checks if `X` can be found somewhere in `Y` (the root of `X` can be any node of `Y`). The tree `T` obtained by gluing together `T1` ... `Tn` is such that the goals `sub_tree(T1,T)` ... `sub_tree(Tn,T)` hold. However the `T`is have to be reordered so that each of them is connected to the currently built tree. The procedure `merge` does the reordering when necessary. In the following, we assume that the functional structure `[att1:val1, att2:val2, attn:valn]` is represented by the incomplete list

```
[att1:val1, att2:val2, .. attn:valn | StillUnspecified]
```

Here is how merge can be defined:

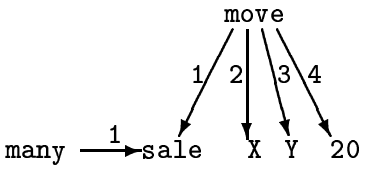
```
sub_tree(tree(Root,S1),
         tree(Root,S2))
         :-
         S1 === S2.
```

```
merge1(T1,T2,T2) :-
    sub_tree(T1,T2).
merge1(T1,T2,T1) :-
    sub_tree(T2,T1).
```

```
sub_tree(
    tree(R1,S1),
    tree(R2, S2) ) :-
    R1 \== R2,
    in_branch(tree(R1,S1),S2).

in_branch(T1,[Att:Val| R]) :-
    bound(Val),
    sub_tree(T1,Val).
in_branch(T1,[Att:Val | R]) :-
    bound(R),
    in_branch(T1,R).
```

```
merge([Tree],Tree).
merge([T1,T2|R], Tree) :-
    merge1(T1,T2,Merg1),
    !, merge([Merg1|R], Tree).
merge([T1, T2|R], Tree) :-
    append(T, [T2], R2),
    merge([T1|R2], Tree).
```

MTM semantic networks have labeled nodes and arrows. Each node is to be interpreted as a predicate, the numbered arrows leading to its arguments. The other arrows represent adjuncts. Nodes will be represented as pairs `nd(Key,Label)` where `Key` is an integer that uniquely identifies the node. We call branching at a node `Node` of a semantic network `SemN` the subnetwork `Br` of `SemN` whose nodes are `Node` and its immediate sons. Each branch is represented by the term `br(Node,sons( ... Soni ...))`. Given the network



where nodes are numbered from `1` to `6`, the branching at `move` is the term

```
br(nd(1,move), sons(nd(3,sale), nd(4,X), nd(5,Y), nd(6,20)))
```

Networks can be represented by the list of branchings at all their nodes. The above network thus becomes:

```
[ br(nd(1,move), sons(nd(3,sale), nd(4,X), nd(5,Y), nd(6,20))),
  br(nd(2,many), sons( nd(3,sale) )),
  br(nd(3,sale), nil),
  br(nd(6,20), nil)]
```

where `nil` is used when there is no son.

The dictionary is a set of clauses whose predicate is the word to be defined. The first argument gives the nature of the entry (a lexical function or a semantic definition). Here are a few examples of "lexical functions" in the sense of MTM:

```
analysis(oper1, perform).
increase(oper1, show).
shoot(syn, fire).
```

Indeed to *analyze* is to *perform an analysis*, to *increase* is *to show an increase* and *fire* is a synonym of *shoot*. These lexical functions are useful tools for paraphrasing because they give alternate wordings or expressions.

A semantic definition relates a net template to a deep structure tree henceforth called its syntactic definition. In order to produce a deep structure from a given semantic net, we have to cover it with subnets whose syntactic definition can be found in the dictionary and merge the syntactic trees so obtained. For example, we can replace the "concept" of `move` of `X` from `Y` to `Z` giving a change of `W` by the "word" `increase` of `X` by `W` provided `Y` and `Z` are not instantiated and that `W` is positive. So we define `move` with the clause:

```
move(semdef,
      [br(nd(N0,move), sons(nd(N1,X), nd(N2,Y),     % net template
                            nd(N3,Z), nd(N4,W)))],
      tree(nd(N0,increase(verb,A)),                 % synt. definition
            [1:tree(nd(N1,X),Br1),
             2:tree(nd(N4,Z),Br2) |_]
          )
      , (var(Y),var(Z),integer(W),W>0)              % conditions
      ).
```

The second argument is the subtree to be matched, the third is the associated tree template and the last is an evaluable list of conditions for the definition to apply. The preceding clause shows the actual representation of the "feature structure".

The semantic component boils down to the following:

```
sem_comp(SemNet, DeepSynt) :-
    tree_set(SemNet,Set),         % find a covering of the network
    merge(Set,DeepSynt1),         % merge the corresponding trees
    clean_up(DeepSynt1, DeepSynt).

tree_set([],[]).
tree_set(Net,[Tree|Rest]) :-
    is_word_of(Word,Net), !,         % find one word in the net
    translate(Word,SubNet,Tree,Conds),
    subset(SubNet,Net,RemainingNet), % extract the subnet
    call(Conds),                     % check conditions
    tree_set(RemainingNet,Rest).

is_word_of(Word,Net) :-
    member(br(nd(Key,N),Brnch),Net)
```

where `subset` is defined the obvious way (`RemainingNet` being the difference between `Net` and `SubNet`) and where we have

```
translate(Word,SubNet,Tree,Conds) :-
    atom(Word),
    Call =.. [Word,semdef,SubNet,Tree,Conds],
    call(Call).
```

The procedure `clean_up` erases all the keys (that have become useless and annoying for other components). Terms of the form `nd(Key,Label)` are replaced by `nd(Label)`.

Once a deep syntax structure is obtained, further paraphrasing is possible by simply replacing a word by a synonym or a construct by an equivalent one. That is what lexical functions are used for. They give useful information for transforming deep structures into equivalent ones conveying the same meaning. For instance they are responsible for the equivalence between `sales increas(ed) (by) 20 percent` and `sales show(ed) an increase (of) 20 percent`; prepositions between parentheses (`by`, `of`) are chosen by the surface syntax component and have no counterpart in the deep structure. As for the `ed` endings, they realize the feature "past" that already figures in the verb node.

There are about sixty such rules valid for any language that can be applied. They take the form:

```
syntax_transf(TreeTemplate1, ResultTemplate, Conditions).
```

The program that uses them to "paraphrase" deep structures is less than 20 lines long. What we have shown here is an adaptation of a program originaly written in Prolog-2 described in Boyer and Lapalme (1985) . Here is a sample of a few of the paraphrases generated from a single semantic network.

```
les ventes ont augmente de 20 tonnes pour totaliser 220 tonnes
les ventes ont accuse une hausse de 20 tonnes en atteignant 220 tonnes
une avance de 20 tonnes a ete enregistree en se chiffrant a 220 tonnes
```

12.6 Question generation

We now describe a text generation component (Desmarais 1989) for a computerized medical information system; it was developped in collaboration with the "Institut de Recherche Clinique de Montréal (IRCM)".

At present, there is a system, called *Dr De Garde*, that asks the user questions answered by menu selections. When the system has enough information about the symptoms of the patient, it makes recommendations on whether he/she should go to the hospital or to some other medical service; it can also give simple advice such as "take some medication, some rest and wait a day or two before consulting a doctor should the sickness persist".

This system has several goals:

- to help cut down on the number of visits to the hospital or to the doctor
- to prompt people to consult their doctor should the symptoms indicate a serious sickness
- to prepare the patient by giving some preliminary information about his condition, hoping that in this way the consultation with the doctor will be more productive

As this system is aimed at a diversified public, it is necessary to adapt the questions to the person asking for help: somebody can seek advice not only for him or herself but also for a child, a spouse or a friend. The user can either be a woman or a man, be an adult, a teenager or an elderly for which the questions (style and content) and recommendations can be very different. Experiments have shown that the success of the system depends very much on the appropriateness of the dialog with the user, so the questions have been customized: the same medical information is repeated in different places in the system but with a different wording because *Dr De Garde* is implemented as a "network" of videotex pages. A consultation corresponds to a path in this

network where the pages displayed depend on menu items choosen by the user. We point out that this is not an "expert medical consulting system", the content of the questions is already predetermined and so are all possible paths following the answers.

To help in reducing this duplication and also in order to be more flexible in dealing with the interaction, we developped a text generation module that can modify the questions according to the following criteria:

user criteria

- with whom the system interacts (interlocutor)
- for whom the system is asked recommendations (patient)
- the sex and the age of the interlocutor and the patient
- the relation between the interlocutor and the patient

grammatical criteria there are many ways of asking a question depending either on the syntax of the sentence or on the words used

contextual criteria

- what questions were previously asked and how do these (and their answers) modify the next questions
- what syntactic choices have already been made

Desmarais (1989) describes the new generation component (written in Prolog) that takes as input coded questions, that leads the user through questions which are generated on the fly and that finally gives the recommendation (in our first prototype, recommendations are taken "as is" from the database and are not generated or customized).

The wording of a question depends on a user model implemented as Prolog clauses:

```
locutor(age,X).      locutor(sex,Y).
patient(age,X).      patient(sex,Y).
relationLocPat(Z).
```

The system asks some preliminary questions in order to instantiate the variables. A question is encoded as a predicate calculus expression that captures the important aspects of the pain which prompted the consultation. We identified the following parameters as being important in this context:

- location on the body
- qualification according to its importance, its intensity and its extent
- circumstances and activating factors
- aggravating or attenuating factors
- time span (for how long have the symptoms appeared)

Here is a few examples of Prolog clauses of these representations:

```
questionAbr(14, respiration(sifflant)).

questionAbr(132, couleur(crachat, disjonction(brun, jaune, blanchatre))).

questionAbr(199, douleur(qualif(excessive),
                disjonction(lieu(dans, oeil),
                            lieu([autour,de], oeil)))).
```

Starting from the user-model and that initial representation, the system goes through two sets of transformations (from a deep syntactic representation to a surface syntactic representation and finally to the utterance). When there is more than one possibility, a random choice is made so that even if a user answers twice the same way, the questions may be different.

Our experiment was done on questions dealing with breathing problems; we give here a sample of generated questions once it has been determined that the user is a male adolescent consulting for his mother (for this paper, we give an english translation of the question)

- `Est-ce que la difficulté à respirer de ta mère est récente?`
  Is your mother's breathing difficulty recent?
- `A-t-elle une douleur à la poitrine?`
  Does she have pain to the chest?
- `Ta mère a-t-elle une respiration sifflante?`
  Has your mother a whistling breathing?
- `A-t-elle beaucoup de difficulté à avaler sa salive?`
  Does she have much difficulty to swallow her saliva?
- `Ta mère a-t-elle une fièvre de plus de 38 degrés?`
  Has your mother a fever of more than 38 degrees?
- `Les crachats de ta mère sont-ils colorés brun, jaune ou blanchatre?`
  Are the spittles of your mother colored brown, yellow or white?

Then follows a set of "canned text" giving the recommendations.

One can appreciate (provided one understands the subtleties of French...) that the questions are aimed at a young person – since "tu" is used instead of "vous" – and feminine is used to take into account the fact that the patient is a woman. Given that systems of this kind are expected to be more and more widespread we think that they will be a very fertile ground for applications of text generation because a good interaction keeps alive the metaphor of a good helping hand which is very important for interactive and cooperative computer systems.

12.7 Conclusion

Using systems developped at the Université de Montréal, we have shown that text generation spans a wide variety of problems starting from knowledge representation, whether at the domain or the application level, to more linguistic ones like making adequate syntactic choices or finding the right utterances. Text generation is not only a simple juxtaposition of sentences and user expectations and inferences must be taken into account.

We have concentrated on the main ideas and taken care to illustrate basic techniques through Prolog program excerpts. Those programs are mere examples and have no pretention at covering all the state of the art techniques. There is unfortunately very few examples of "real code" in the published litterature on text generation and our intent was to help fill that gap in giving a few practical techniques at the expense of comprehensiveness. Our feeling is that many of the latest ideas in text generation are still too often hidden in "thousand of lines of Lisp".

12.8 Acknowledgement

We first want to thank Richard Kittredge who introduced us to the problem of text generation through many fruitful discussions. We also thank Igor Mel'čuk and Chantal Contant for ideas on this problem. Syvie Giroux and Evelyne Millien produced a Prolog version of Ana. Patrick St-Dizier also gave useful comments on previous versions of this paper.

Bibliography

[1] D. E. Appelt. *Planning English Sentences.* Cambridge University Press, Cambridge, 1985.

[2] M. Boyer. Towards functional logic grammars. In P. St-Dizier, editor, *Proceedings of the Second International Workshop on Natural Language Understanding and Logic Programming*, pages 46–62, Vancouver, 1987.

[3] M. Boyer and G. Lapalme. Generating paraphrases from meaning-text semantic networks. *Computational Intelligence*, 1(3), 1985.

[4] C. Contant. Génération automatique de rapports boursiers français et anglais. *Revue québécoise de linguistique*, 17(1):197–222, 1988.

[5] L. Danlos. *Génération automatique de textes en langues naturelles.* Masson, 1985.

[6] S. Desmarais. Génération de questions adaptées à l'usager dans un système de consultation médicale. Thèse de maîtrise, Université de Montréal, 1989.

[7] R. Kittredge. Semantic processing of texts in restricted sublanguages. *Computers and Mathematics with Applications*, 9(1):45–58, 1983.

[8] K. Kukich. Design of a knowledge-based report generator. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 145–150, 1983.

[9] W. Mann. Text generation. *American Journal of Computational Linguistics*, 8(2):62 – 69, 1982.

[10] D. D. McDonald. Natural language generation as a computational problem : an introduction. In M. Brady, editor, *Computational Models of Discourse.* MIT Press, 1983.

[11] D. D. McDonald and L. Bolc (eds). *Natural Language Generation Systems.* Springer-Verlag, 1988.

[12] K. R. McKeown. *Text Generation.* Cambridge University Press, Cambridge, 1985.