Counting dependencies and Minimalist Grammars

Maxime Amblard

Counting dependencies and Minimalist Grammars. *

Maxime AMBLARD †

Minimalist Grammars (MG) are a formalism which allows a flexible syntactic analysis of natural languages. It was introduced by Stabler in [St 97]. Its generative capacity has been studies in [Ha 01].

This article describes the existence of a MG generating the counting dependencies $L_m = \{1^n 2^n \cdots m^n, n \in \mathbb{N}\}$, and an algorithm of construction of the lexicon Lex_m producing this language. It is a generalization of the Stabler presentation with $n = 5$ [St 97].

This class of languages belongs to the context-sensitive languages in the hierarchy of Chomsky. In a linguistic way, we could find example of this structure in sentence like : "Peter, Mary and Charles had respectively 14, 12 and 6 in math, history and sport".

1 Stabler's MG

Stabler's Minimalist Grammars are lexicalised grammars. Therefore the generated language is the transitive closure of the lexicon under the generating functions. Each lexical entry is a list of features. The features are of two different natures and take part in the release of two distinct operations.

Different types of feature :

The set of base features is noted BF. The following features are also defined :
- select : $\{=d \mid d \in BF\}$.

The set of move features is noted MF. The following features are defined :
- licensors : $\{+k \mid k \in MF\}$.
- licensees : $\{-k \mid k \in MF\}$.

Generating functions :
- Merge : unification of a base feature with the corresponding selector. The result is the concatenation of the other feautures.
- Move : unification of a licensor with a licensee. It corresponds to the move of the features to the components carrying the licensees in front of the structure.

We use the following notation : e stand for a feature of an arbitrary type and E for a sequence of features.

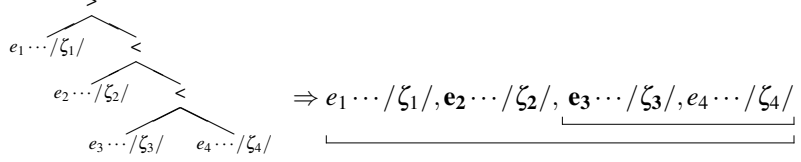
A lexical entry is made of a list of features and the associated phonological form, noted between oblique bars : e1…/z1/. The word generated is recognized by a left-right-hand side reading of the phonological forms in the analysis.

The phonological form will be called "terminal" and the other elements of the list of features "non-terminal".

Traditionally, the analyses are finite, binary and ordered trees with projections - which preserve the position of the head of the component. This order is marked on the nodes of the tree by '$<$ ' or '$>$ ' - for the direction of the head. In this article, we will use list ordered from left to right. A component will be delimited by an under-brace and the head of this last will be marked in bold. To simplify the graphical representation, the group containing only one element and those containing only a phonological form will not be marked by a under-brace and the head will take back a normal *font*.

The linear representation contains less information than the tree form but this information is sufficient to describe the mechanisms of our paper.

Here an example of translation of an analysis in tree form to a linear representation :



*LACL 2005 - poster for the student session - april.

†SIGNES team, LaBRI, université de Bordeaux 1- INRIA - CNRS

Graphical representation of rules :

– *Merge* results in an addition of a component into first position in the list during the derivation. Indeed it occurs between two entities such as in first position in the list of the features of the head one finds a basic feature in one and a selector in an other element (often a lexical entry).

$$\frac{\text{d } E_1/\zeta_1/ \quad =\text{d } E_2/\zeta_2/}{\underbrace{\mathbf{E_2}/\zeta_2/,\ E_1/\zeta_1/}}$$

The element carrying the selector will be the new head.
When a merge occurs between two lexical entries, the head will be placed on the left, in the other cases, it is the new lexical item which will be placed on the left.

– *Move* corresponds to placing the list for the component whose the licensee is the head in first position.

$$\frac{\underbrace{U,\ +\text{k } E_1\ /\zeta_1/,\ V,}\ \underbrace{\quad\vdots\quad}\ Y \qquad\qquad S,\ \underbrace{W,\ -\text{k } E_2/\zeta_2/,\ X,}\ T}{S,\ T}$$

$$\underbrace{W,\ E_2/\zeta_2/,\ X,}\ U,\ E_1\ /\zeta_1/,\ V,\ \underbrace{\vdots\ ,}\ Y$$

Only the internal order of the elements and the head of the moved element are modified if : $W = \varepsilon$ et $X = \varepsilon$.

2 Example of counting dependencies : 1^n2^n

To build the word $1^n2^n, n \in \mathbb{N}$, we use these lexical entries - a proof will be explain in the next section.

type : 1	2 -2 /2/	*type* : 4	=1 +2 2 -2 /2/
type : 2	=2 1 -1 /1/	*type* : 5	c
type : 3	=2 +1 1 -1 /1/	*type* : 6	=1 +2 +1 c

Sketch of derivation :

The entries of type 1 et 2 start the derivation. They add one of each terminal respectively.

Those of type 3 et 4 form the iterative part by adding a non-terminal and moving the group of this non-terminal to form a new entity.

The entry of type 5 allows the analysis for $n = 0$.

The last (6) finishes the derivation while putting the groups of terminals in the right order.

Example derivation :

1. Lexical entry of type 1 : 2 -2 /2/
   and one of type 2 : =2 1 -1 /1/
2. Merge : **1 -1 /1/**, -2 /2/
3. At this time, there are as many elements /1/ as /2/ elements. We could either finish the derivation with an entry of type 6 and obtain /1/,/2/, or take on iterative phase to build 1^22^2. Let us continue the derivation with
   a lexical entry of type 4 : =1 +2 2 -2 /2/
   and merge with the previous element : **+2 2 -2 /2/**, **-1 /1/**, -2 /2/
4. Move : /2/, **2 -2 /2/**, -1 /1/
5. There have one /2/ too many, it is necessary to add one /1/, which is done by a lexical entry of type 3 :
   =2 +1 1 -1 /1/
   – second part of the iteration – and a merge : **+1 1 -1 /1/**, /2/, **-2 /2/**, -1 /1/,
6. Move : /1/, **1 -1 /1/**, /2/, **-2 /2/**
7. Now, we have the same structure as in stage 2, with one /1/ and one /2/ more. The same choice is proposed : reiterate or conclude. Let us reiterate once more : lexical entry of type 4 :

=1 +2 2 -2 /2/

and merge : **+2 2 -2 /2/**, /1/, **-1 /1/**, /2/, **-2 /2/**,

8. Move : /2/, /2/, **2 -2 /2/**, /1/, **-1 /1/**

9. Lexical entry of type 3 : =2 +1 1 -1 /1/

   and merge : **+1 1 -1 /1/**, /2/, /2/, **-2 /2/**, /1/, **-1 /1/**

10. Move : /1/, /1/, **1 -1 /1/**, /2/, /2/, **-2 /2/**

11. After this new iteration, there are three $/1/$ and three $/2/$. Let us finish derivation. Lexical entry of type 6 : =1 +2 +1 c

    and merge : **+2 +1 c**, /1/, /1/, **-1 /1/**, /2/, /2/, **-2 /2/**

12. Move : /2/, /2/, /2/, **+1 c**, /1/, /1/, **-1 /1/**

13. Move : /1/, /1/, /1/, /2/, /2/, /2/, **c**

3 Generalization

This section presents a general algorithm to construct a lexicon generating a language of an N counting dependencies : $1^n 2^n \cdots N^n$, and outlines the proof of the language generated by the grammar with this lexicon.

Algorithm *Construction of the lexicon.*

It will suppose $S_1 < S_2 < \cdots < S_{N-1} < S_N$ where :

– $/S_i/$ are the terminals of the derivation, ordered according to appearance in the word

– S_{acc} is the accepting symbol of the grammar.

type 1 : $\langle S_N \text{ -}S_N\ /S_N/\rangle$

type 2 : for i from 1 to (N-1) $\langle =S_{i+1}\, S_i \text{ -}S_i\, /S_i/\rangle$

type 3 : from j from 1 to (N-1) $\langle =S_{j+1} +S_j\, S_j \text{ -}S_j\, /S_j/\rangle$

type 4 : $\langle =S_1 +S_N\, S_N \text{ -}S_N\, /S_N/\rangle$

type 5 : $\langle S_{acc}\rangle$

type 6 : $\langle =S_1 +S_N +S_{N-1} \ldots +S_1\, S_{acc}\rangle$

Theorem *Minimalist Grammars generate all counter languages.*

Proof *The previous part presents how to obtain 2 counting dependencies. Let us see how to extend it to N terminals with the algorithm above.*

The synopsis of the analysis is done according to three phases : start-iteration-conclusion. We will take a type of lexical entry according to the different phases :

The first type of lexical entry will combine with the last entry of type 2 ($S_{i+1} = S_N$ pour $i = N-1$) using merge. Thereafter this structure will combine with the preceding one of the type 2 and so on, until the start phase is finished, i.e. until we have accumulated a terminal of each letter. This is made possible by the structure of the elements of the type 2 because following the selector we find a basic feature with an index decreased by 1 (from where merge with the precedent). Once this phase is finish, a basic feature S_1 is in first position : $S_1\ -S_1\ /S_1/, \cdots, -S_N/S_N/$

The choice is thus either to pass directly to the conclusion phase, or to pursue with an iteration.

Iteration phase : it starts with a merge of a lexical entry of type 4 designed for this purpose. This new head immediately moves all the elements $/S_N/$ to the front . Then we find the same structure as in the start phase, which enables us to continue the iteration.

The action, in this phase, is, in addition to accumulating a phonological form, to move all elements carrying the same phonological form in first position : $+\mathbf{S_N}\ \mathbf{S_N}\ -\mathbf{S_N}/\mathbf{S_N}/, \cdots, /S_N/, \cdots, -\mathbf{S_N}/\mathbf{S_N}/$ *becomes :* $/S_N/, \cdots, /S_N/,\ \mathbf{S_N}\ -\mathbf{S_N}/\mathbf{S_N}/, \cdots$

At the end of the this phase, the derivation reaches again in the same configuration as at the end of the start phase. We could either start an iteration again, or conclude.

To conclude, the derivation is merged with an entry of the type 6, which orders all group of the same phonological form. $\underbrace{+S_N \ \ldots \ +S_{init}\ S_{acc}, \cdots,\ \underbrace{/S_1/, \cdots, -S_1\ S_1}, \cdots, \underbrace{/S_N/, \cdots, -S_N\ /S_N/}}$ *Thus, successive moves reorder the derivation according to each terminal by using the last licensee remaining with phonological forms. As we always added a series of terminal on each iteration phase, they all occur the same number of times.*

This grammar generates exactly the counter languages with N terminals : $1^k \cdots N^k$ *because only the analyses following the synopsis above can succeed. Any variation with in this synopsis will not return an accepting analysis because this kind of derivations are deterministic except at points that we will discuss :*

Starting the iteration phase without completing the start phase.

We can start a derivation by merge between an entry of type 1 and one of type 2, by an entry of the type 1 and one of type 3. Into this second case, we introduce a feature '$+k$*' into derivation. There is no element in derivation carrying the equivalent licensee. Therefore, the derivation fails.* $\underbrace{+\mathbf{S_{N-1}}\ \mathbf{S_{N-1}}\ -\mathbf{S_{N-1}}\ /\mathbf{S_{N-1}}/,\ -S_N\ /S_N/}$

If that occurs later in the start phase, the problem will be the same.

Returning from the iteration phase to the start phase.

The derivation uses a merge with an entry of type 2 instead of one of type 3. In this case, it misses one ' $+k$*', *$\forall k \in MF$ *in derivation. But the only moment in a derivation where there are two features '*$-k$*', is followed by a merge operation with an entry of type 3, but one of them will be unified immediately with the introduced feature '* $+k$*'.*

In this case, there are two '$-k$*' in the derivation, but only one of them can be unified in the conclusion phase. The analysis will finish with this additional feature '*$-k$*' and could not yield a successful derivation :* $\underbrace{+S_N \ \ldots \ +S_1\ S_{acc}, \cdots,\ \underbrace{/S_1/, \cdots, -S_1\ S_1}, \cdots, \underbrace{/S_N/, \cdots, -S_N\ /S_N/, -S_N\ /S_N/}}$

All the other stages of derivation are deterministic, therefor we obtain correctly the words on a counter.

Conclusion and prospects

The languages generated by Minimalist Grammars contain the counter languages. This is the point that distinguishes these grammars from other linguistic formalisms.

A version of $a^{(2^n)}$, $\forall n \in \mathbb{N}$ counts is presented in [Mi 05].

An MG of the *nested counters* is in progress. The *nested counters* are the sentences of the following shape : $1^n 2^k 3^n 4^k \cdots N^k$, $\forall n \in \mathbb{N}$, $\forall k \in \mathbb{N}$ which is a context-sensitive language, as counter languages with more than two terminals.

In this respect MG (strongly) differs from other derivational formalizations of NL syntactic structures.

They provide an account for linguistic analysis and we could show these complex syntactic structures by theoretical exploration. The main open question is whether it is possible to generate languages outside the class of natural languages.

Références

[Ha 01] Harkema H. (2001). A Characterization of Minimalist Languages. *Logical Aspect of Computational Linguistics 2001*. Springer-Verlag.

[Mi 05] Michaelis J. (2005). A Note on the Complexity of Constraint Interaction : Locality Conditions and Minimalist Grammars. *Logical Aspect of Computational Linguistics 2005*. Springer-Verlag.

[Mi Mö Mo 00] J. Michaelis, U. Mönnich and F. Morawietz (2000). Algebraic Description of Derivational Minimalism. *Algebraic Methods in Language Processing*. 125-141

[Mö Mo Kep 01] U. Mönnich, F. Morawietz and S. Kepser (2001). A Regular Query for Context-Sensitive Relations. *IRCS Workshop Linguistic Database*. 187-195

[St 97] Stabler Ed. (1997), Derivational Minimalism, *Logical Aspect of Computational Linguistics 1997*. vol 1328, Springer-Verlag.