Generating with Minimalist Languages

Francesco Zamblera

August 28, 2011

Abstract

This article describes a free implementation of Computational Minimalist Grammar, and particularly the generator. The implementation is targeted to the subset of English and other languages known as the Natural Semantic Metalanguage.

After an introduction on Computational Minimalist Grammars (section 1), which sketches the particularities of this implementation, section 2 shows how the generator works, while section 3 discusses the use of feature variables in PF.

1 Introduction

This paper presents an implementation of Computational Minimalist Grammars, written in Python and freely available on the web. The program is part of a project directed to build automatic translators for the Natural Semantic Metalanguage, which I am developing in collaboration with Cliff Goddard and Anna Wierzbicka.[1]

By "Computational Minimalist Grammars" I refer to the computational version of Chomsky's Minimalist Program developed by Edward Stabler (see e.g. Stabler, 1997, 2011b,a). For computational purposes, I have taken as my point of depart Harkema's PhD dissertation (Harkema, 2001), which is based on Stabler's 1997 model.

The program implements a parser, a generator and a translator, which is simply obtained by piping the LF output by the parser into the generator.

This article describes my particular implementation of Stabler's Minimalist Grammars using a restricted lexicon. A preliminary account of the English NSM in this particular framework is given in Zamblera (Forthcoming).

In this paper, I will focus mainly on the generator, as the parser is basically an implementation of Harkema's CKY bottom-up parser (see Harkema, 2001, chapter 4).

[1] Information and literature for Natural Semantic Metalanguage (NSM) can be found at the website http://www.une.edu.au/bcss/linguistics/nsm/, of the University of New England, where a link to this program can be found (http://www.une.edu.au/bcss/linguistics/nsm/translator.php).

1.1 Features of the Implementation

My implementation works bottom-to-top, standing very close to the "traditional" definition of *merge* and *move*, both in parsing and in generating. The shortcomings of such an approach for *performance* models have been highlighted by Chesi (2004, 2007), who proposes an alternative top-down strategy. In this work I try to overcome the problems of a bottom-to-top approach by translating the dominance relations discovered by the "Cartographic approach" [2] into "level numbers" which are added to the base features:

- All the functional heads which belong, for example, to the nominal extended projection have the base feature `n`, as in Chesi (2004), but with an added level number (`n:1`, `n:2`, and so on).

  Crucially, *merge* is allowed when the numerical index of the selector is *greater or equal* than the one of the base. So, a selector like `=n:2` can merge with a `n:1` or a `n:2` base (but not with any `n:k` where $k \geq 3$).

  In this way:

  - We can directly represent the hierarchy of functional heads by giving progressive number, starting from the bottom of the hierarchy: so, for example, given the final part of Scott's hierarchy in the noun phrase (Scott, 2002, p. 114):

    $\ldots >$ COLOR $>$ NATIONALITY/ORIGIN $>$ MATERIAL $>$ COMPOUND ELEMENT $>$ NP

    we can represent the noun head base feature as `n:1`, the functional head which hosts the "compound element" modifier will be `n:2`, the head of "MaterialP", `n:3`, and so on, up to the determiner.[3]

    We will let a selector like `=n`, without level numbers, merge with `n:k` for any k. So, the selector feature of the determiner will be simply `=n`.

  - This representation also expresses the fact that functional heads belong to the same extended projection: so, nominal functional heads will all have the base feature n_i, while verbal ones will be v_i.

  - We can account for the optionality of adverbial modifiers: so, for example, the derivation of "(Probably) (suddenly) Napoleon died", discussed in Chesi (2004, 138-40), and Chesi (2007, 55-56) where the two optional adverbs are problematic, can be obtained given a (very simplified) lexicon as:

    [$_N$ Napoleon], [$N{=}\ V_1$ die], [$=V_1\ V_2$ suddenly], [$=V_2\ V_3$ -ed], [$=V_3\ V_4$ probably], [$=V\ C$].

    The complementizer C will be able to merge with any of the V_i, thus allowing the adverbials to be optional.

---

[2] Cf. Belletti (2004); Cinque (1999, 2002, 2006); Rizzi (1997, 2004).

[3] In the following discussion, `cat:number` will be alternatively represented as cat_{number}.

Other than by the use of level numbers, the hierarchies discovered by the cartographic studies are reflected in LF:

- I adopt a simple predicate-argument structure for LF and, crucially, elements which occur higher in the hierarchy act as predicates taking the lower structure as their argument.

  Another important addition to Stabler's model is the use of variable-sharing between syntactic features and PF representation:

- PF representations can contain *variables* which are crucially shared with syntactic features. These variables are instantiated by feature-checking in the usual probe-goal relation (cf. Chomsky, 2001, 2005; Hornstein *et al.* , 2005, p. 317 ff.). In this way, features can be erased from syntax when checked, but their PF effects remain visible for the PF interface.

  For example, the morphonological entry which covers the Spanish forms *bueno, buena, buenos, buenas* is `buen-$gen$-$num$`, where `$gen$` and `$num$` are variables.[4] Among the syntactic features (which are those to be checked by movement operations), there will be something as $-f_{gender}: \$gen\$$ and $-f_{number}: \$num\$$. A *move* operation triggered by a noun with, for example, $+f_{gender}: F$ and $+f_{num}: P$ will assign the right gender and number values, so that `buen-$gen$-$num$` becomes `buen-F-P`. This will be converted into *buenas* by morpho-phonetic rules.[5] This use of features is described more fully in section 3.[6]

  The minimalist grammar here developed uses feature-checking as a morphological device to instantiate e.g. agreement, much in the way as feature unification is used in other computational models.

- the problem of persistent features is solved simply by putting the same feature twice, once as a licensor and then as a licensee (see section 3.1). For example, in the Spanish noun phrase *estas personas buenas*, gender and number features may still be checked when the nuon phrase is integrated in a higher structure, e.g. *estas personas son buenas*. Let's therefore say that, simplifying, the determiner has the following feature structure:

$$\begin{array}{rll} =n & +f_{gender}: \$gen\$ & +f_{number}: \$num\$ \\ d & -f_{gender}: \$gen\$ & -f_{number}: \$num\$ \end{array} \tag{1}$$

[4] Variables have the same name of the feature surrounded by \$. If two different variables should be needed with the same name, numerical indices are added. So, for example, $\$gen_1\$$, $\$gen_2\$$. This operation is done automatically by the program when variables need renaming (see section 3.2).

[5] The process here described refers to the generation process. In parsing, the algorithm starts, of course, with an already instantiated form, as for example *buenas*. morpho-phonetic rules will change it into `buen-F-P`, with features $-f_{gender}: F$ and $-fnumber: P$. A movement operation will check these features against those of the nominal head.

[6] Cf. ?, p. 226, note 13: "Morphological agreement phenomena involve feature copying in PF".

Crucially, the same variables `$gen$` and `$num$` are shared among the relevant licensors and licensees. Let the feature structure of the phrase *personas buenas* be the following:

$$n{:}2 \quad -f_{gender} : fem \quad -f_{number} : pl \tag{2}$$

First, Merge({estas},{personas buenas}) will apply, erasing the selector `=n` from the determiner and the base `n:2` from *personas buenas*. Then, two instances of *move* will check first $+f_{gender}$ and then $+f_{number}$ of the determiner against the relevant features of the noun phrase. Checking will instantiate the variables `$gen$` and `$num$`. After these operations, the feature structure of the determiner will be:

$$d \quad -f_{gender} : fem \quad -f_{number} : pl \tag{3}$$

In this way, we get the effect of persistent features without any machinery such as optional deleting, argued against by Chesi (2004).

- I have adopted a reverse-Polish notation for LF, which allows for a very simple bottom-up building of LF (in the parser) and bottom-up processing of LF (in the generator).

  For example, let the LF of the sentence "this person moves" be something like this:[7]

$$DECL(PRESENT(MOVE(THIS(PERSON)))) \tag{4}$$

In reverse-Polish notation, this becomes

$$\text{PERSON} \quad \text{THIS} \quad \text{MOVE} \quad \text{PRESENT} \quad \text{DECL} \tag{5}$$

Here is how a traditional predicate-argument structure becomes in reverse-Polish notation:

Arity	Structure	Notation
0	arg	`arg`
1	$pred(arg)$	$\texttt{arg}_1$ `pred`
2	$pred(arg_1, arg_2)$	$\texttt{arg}_1$ $\texttt{arg}_2$ `pred`
3	$pred(arg_1, arg_2, arg_3)$	$\texttt{arg}_1$ $\texttt{arg}_2$ $\texttt{arg}_3$ `pred`

[7] The LF, output of the parsing process and input of the generator, is a very simple predicate-argument scructure.

Some examples of nested structures:

Structure	Notation
$pred_1(arg_1, pred_2(arg_2))$	$\texttt{arg}_1$ $\texttt{arg}_2$ $\texttt{pred}_2$ $\texttt{pred}_1$
$pred_1(arg_1, arg_2, pred_2(arg_3, arg_4))$	$\texttt{arg}_1$ $\texttt{arg}_2$ $\texttt{arg}_3$ $\texttt{arg}_4$ $\texttt{pred}_2$ $\texttt{pred}_1$
$pred_1(arg_1, pred_2(pred_3(arg_2, arg_3))$	$\texttt{arg}_1$ $\texttt{arg}_2$ $\texttt{arg}_3$ $\texttt{pred}_3$ $\texttt{pred}_2$ $\texttt{pred}_1$

After this cursory survey of the main characteristics of this implementation, the next section will briefly review Stabler's model of minimalist grammars.

1.2 Minimalist Grammars

As in categorial grammars (see e.g. Steedman, 2000), a minimalist grammar consists mainly of a *lexicon*; syntactic constructions are generated by the combinatorial properties of lexical items,[8] together with the two structure-building operations *merge* and *move*. These operation are triggered exclusively by the syntactic features of the lexical items.

A lexical item is essentially a bundle of features. Each item has three types of features:

syntactic features, which determine the morphosyntactic properties of the lexical item, and trigger the two operations of *merge* and *move*. For example, a transitive verb like *read* has the categorial feature `v`, and the selection feature `=d`, meaning that it will merge with a DP in a head-complement structure. An inflection head like *-s* will have among its features `=v`, so it will merge with a verb to form a structure $[_i i\ v]$;

phonetic (PF) features, represented by a string of characters. I prefer to call these features **morpho-phononetic**, because this implementation allows for "abstract" PF representations containing *variables*, which correspond to variables in the syntactic features.

semantic (LF) features, which will be represented as an uppercase LF predicates. *Merge* operations compose semantic features, so that the LF of the head takes as its argument the LF of the merged item (complement or specifier). As LF is in reverse-Polish notation, this means simply that both LFs are represented by strings, and LF_{head} is concatenated after LF_{compl}.

1.3 Syntactic features

There are four kinds of syntactic features, grouped in two sets:[9]

[8] The program here developed adds a morpho-phonetic rule component.

[9] In section 4, some possible refinements of this architecture are briefly discussed, which take into consideration more recent proposals.

1. **Categorial features**

   base features like `v`, `a`, `n`. Each lexical item has one and only one categorial feature;

   selectors like `=v`, `=a`, `=n`. An item with selector $=f$ can merge with an item whose categorial feature is f. Both f and $=f$ are deleted after merge. Furthermore, in my implementation, an item with selector $=f_i$ can merge with an item whose base is base f_j, if $i \geq j$, or with an item whose base is f, for any i. Again, both f_j and $= f_i$ are deleted after merge. If an item has more than one selector, the first one will trigger merge with a *complement*, all the others will select *specifiers*.

2. **Movement-related features**

   licensors represented as `+f` (or, in my implementation, also as `+f:val`), together with

   licensees trigger the *move* operation. An item with $+f$ on the top of the tree will attract an item lower in the syntactic tree, whose first feature is $-f$, if there is no other subtree with the feature $-f$ (*shortest move constraint*). In my implementation, an item with `+f:val1` on the top of the tree attract an item lower in the syntactic tree, whose first feature is `-f:val2`, if `val1` can be matched with `val2`. In that case, *move* applies and the two features are deleted. Each of the two values can be a variable or an actual value (for example, if the feature is f_{gender}, values could be *masc*, fem, or a variable \$*gender*\$).[10]

   To check the feature `val1` against `val2`, the following procedure is applied:

   - if neither `val1` nor `val2` are variables, they match only if they are identical (so e.g. $-f_{gender} : masc$ matches $+f_{gender} : masc$ but not $-f_{gender} : fem$);
   - if both are variables, they are unified: after *move*, `val1` = `val2`, and when either will be assigned a value (by later movement operations), the other will automatically assume the same value;
   - either one of `val1`, `val2` is a variable and the other an actual value, the variable is assigned that value. So e.g. $+f_{gender}$: \$*gen*\$ and $-f_{gender} : masc$ match, and the variable \$*gen*\$ is assigned the value *masc*.

When a variable is assigned a value, all instances of that variable in the morphophonological and morphosyntactic features will be assigned that value, as we have seen before in the brief discussion of Spanish adjective gender agreement. In this way, even if the two features get deleted by *move*, their values can persist if there are other instances of that variable.

[10]The program requires that possible values for each feature be declared in advance. In this way, ther parser can go back, for example, from `buen-F-P` to `buen-$gen$-$num$`.

1.4 The Lexicon

The program implements a lexicon as a Python dictionary[11], using the LF representation as key, and a list of strings as value. The first item of the list represents the PF, and all the following items are the morphological features.

An entry looks like this:

$$LF : [PF, f_1, f_2, ...] \tag{6}$$

Of the syntactic features f_1, f_2, ..., only the first of the list is active (that is, it can trigger some operation).[12] For example, consider the item:

$$LF_1 : [PF_1, =f_1, +f_2 : val_2, f_3, -f_4 : val_4] \tag{7}$$

which can be taken directly from the lexicon, or be the effect of previous derivations. The only possible operation for this item is a *merge* with some item of the form:

$$LF_2 : [PF_2, f_1, \ldots] \tag{8}$$

If *merge* applies, the top feature will be deleted, and our first item becomes:

$$LF_1 + LF_2 : [PF_1, +f_2 : val_2, f_3, -f_4 : val_4] \tag{9}$$

Now the active syntactic feature is $+f_2 : val_2$, so the item can only attract a movement candidate, whose first morphological feature is $-f_2 : val_j$, where val_j must match val_2 in the sense defined above. After movement, $+f_2 : val_2$ will be deleted, and f_3 will become the active feature, and so on, until all syntactic features have been deleted.

1.5 A Sample Derivation

Let us now see how the machinery described above works in the actual derivation of a simple sentence, *Titus praises Lavinia*.[13]

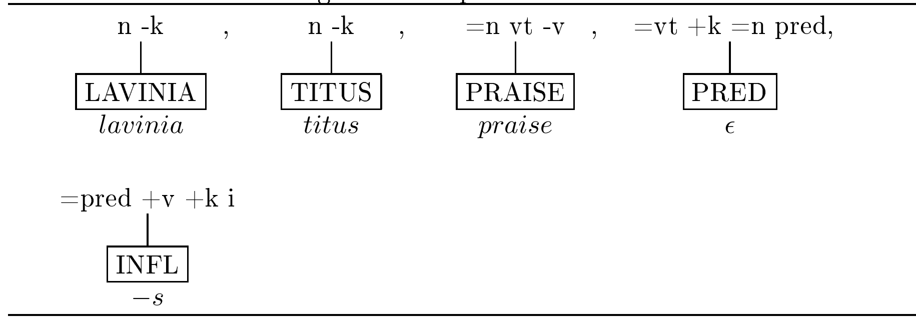
The derivation begins by selecting the lexical items with which the structure will be built. This set of selected items constitues a *numeration*. Figure 1 represents the numeration for the sentence *Titus praises Lavinia* (LF representations are boxed, PF are in italics. ϵ represent an empty category, that is, an item with null PF.).

[11] A dictionary in the programming language Python is essentially a list of ordered pairs <key,value>.

[12] It is the "edge feature" of Chomsky (2005, 6).

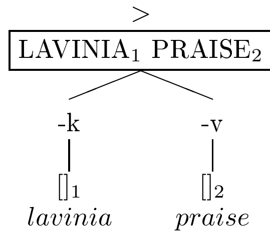
[13] The example is taken from (Harkema, 2001, 31-35), cf. also Stabler (1997). I have only added the LF representations.

Figure 1: Sample Numeration

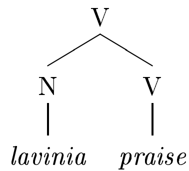


The derivation process selects an item at a time from the numeration, and adds it to the structure built so far, applying *merge*, then tries to apply *move* if possible.

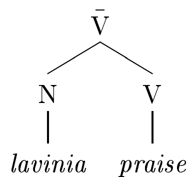
We start by selecting items 1 and 3 and merging:



after merge, the categorial features `n` and `=n` disappear. The tree is headed by the arrow > pointing to the head of the tree. (in a more traditional GB notation, this tree would look as



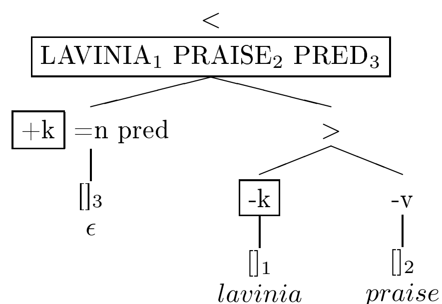
or, with bar levels,



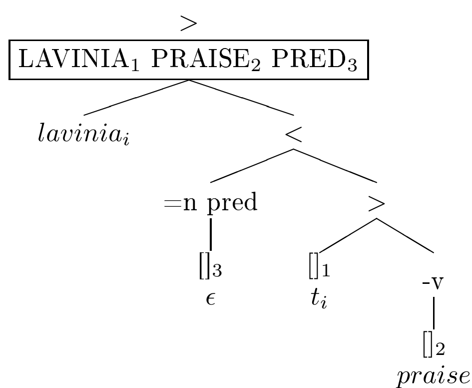
I adopt here the notation used in Harkema's and Stabler's works).

Note how the *merge* concatenates immediately the two LFs.[14]

In the next step, the fourth item is selected and merged with the tree built so far. The result is:

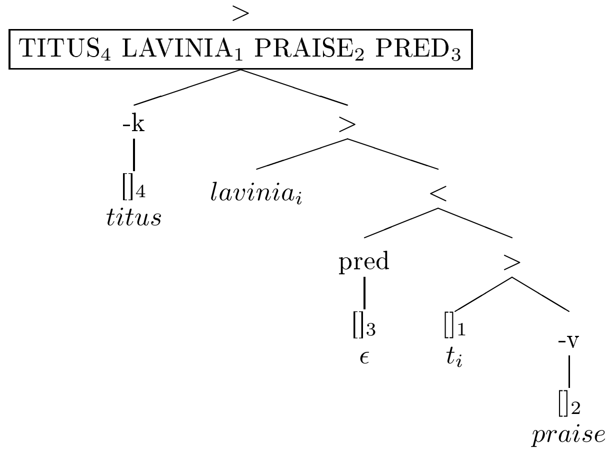


The head of the construction is now the phonetically empty `pred`. Now a *move* operation is possible: the head of the tree has a feature `k` to check, and the noun *lavinia* has the corresponding feature to be checked (`+k` and `-k` are boxed in the above tree). After movement, the tree becomes

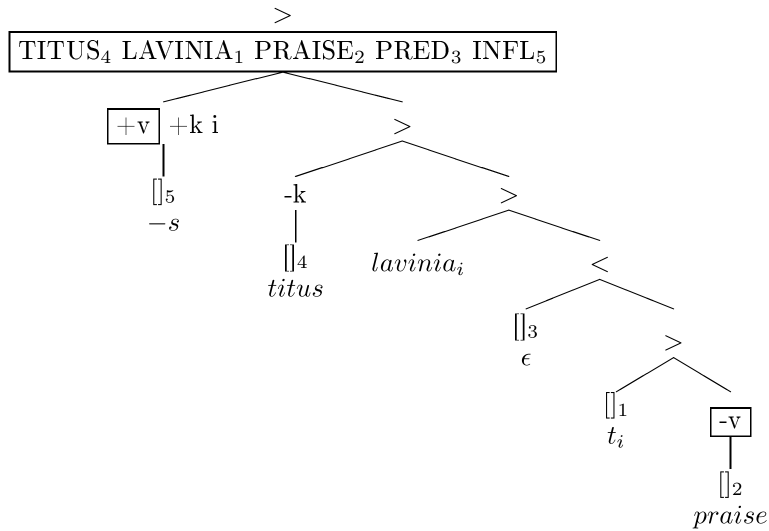


In the next step, the lexical item *titus* is picked out from the numeration and merged:

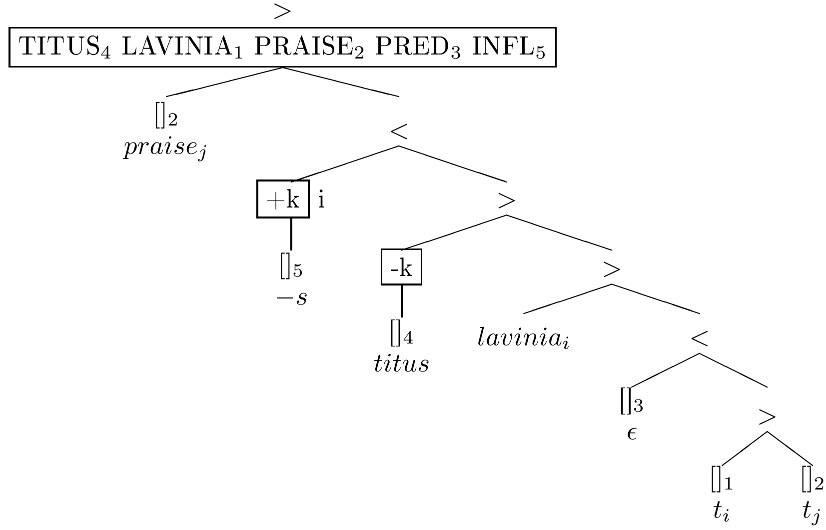
[14] *Move* has no effect on LF representation in this framework. This is a peculiarity of my implementation, whose consequences will have to be tested on larger samples of English than NSM.

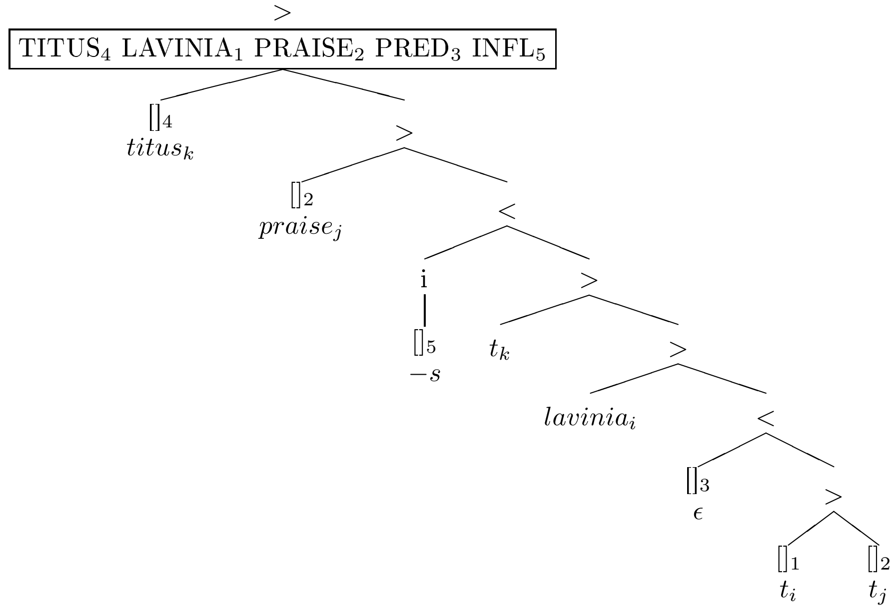


The head of the tree is the *pred* item (note how it is the only categorial feature present in the tree; hte other have been discharged after a successful *merge*). There remains item $[_i\ {-s}]$ to be selected, and merged with the structure:



The head i has still two features to check: $+v$, boxed in the above tree, and $+k$. These features are checked by the two last applications of *move*:





The final LF is equivalent to:

$$INFL(PRED(PRAISE(LAVINIA), TITUS)) \tag{10}$$

and the PF is *titus praise -s lavinia.*

1.6 Competence and Performance[15]

The derivation sketched above is an example of how minimalist grammars can model the speaker's *competence.* As Chomsky has always made clear, from the very beginning (e.g. Chomsky, 1957, 1965), a generative grammar is not involved in the production of specific sentences, but is to be understood as a computational device which generates, in the mathematical sense, the grammatical sentences of a language, assigning them a structural description.

A generator and a parser, on the other hand, produce and, respectively, analyse particular sentences. A generator produces a sentence starting from a specified LF, while a parser goes the opposite way, from PF to LF. Being so, a generator-parser can be thought of as a model of a speaker-hearer's *performance.*[16]

So the term *generate* means two different things:

- In the Chomskian competence-oriented sense, *generate* means enumerate all and only the grammatical sentences which can be derived by a grammar;
- In computational linguistics, to generate a sentence means to produce a specific sentence.

Given the enumeration above, the computational device can generate (in the first sense), beyond the sentence "Titus praises Lavinia", also "Lavinia praises Titus", by merging "praise" with "Titus" instead of "Lavinia" in complement position.

If we add to the numeration the following item:

=pred +v +k i
|
[INFL]
$-d$

we can also generate "Titus praised Lavinia" and "Lavinia praised Titus". And so on.

We can summarize the difference between competence and performance as in table 1.6.

The model of competence exemplified by the above derivation is similar to the one proposed in Hornstein *et al.* (2005), from which I take (pag. 328) figure 1.6, slightly adapted:[17]

[15] An illuminating discussion on this issue can be found in Chesi (2007).

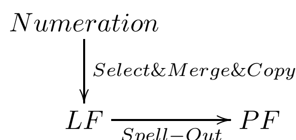
[16] This is true, of course, only to the extent that a generator-parser is not built simply as a computational device, but can claim to be models of the human generation and parsing processes.

[17] "Select" refers to selection of items from the numeration, while *move* is analyzed as a combination of two elementary operations, *copy*, which copies an item from a lower position in the tree, and *merge*.

Table 1: competence and Performance

	Competence	**Performance**	
		Parsing	Generation
Starting point	Numeration	PF	LF
Result	All possible pairs {PF,LF}	LF	PF

Figure 2: a model of Minimalist Grammar



The main difference concerns the *spell-out* operation. In my implementation (as in Stabler (1997) and Harkema (2001), if I understand them correctly, as well as Chomsky (2005, p. 9)), LF is not mapped unto PF by a (single) spell-out operation; instead, LF and PF are built in parallel: each application of *merge* and *move* operates on all the three kinds of features (syntactic, semantic and morpho-phonetic) at the same time. In particular:

- syntactic features are matched and deleted by both *merge* and *move*;
- PFs are concatenated, again by both *merge* and *move*, provided the relevant items will not move further. This could be seen as a particular form of *spell-out* happening at each step in a derivation. A similar model is proposed in Wojdak (2005), and exemplified with extensive documentation from the Wakashan language Nuu-chah-nulth;
- in my implementation, *merge* (alone) also composes LFs.

This is not yet the whole story, however, because in my implementation PF and LF, once built, still have to undergo some transformation, in the form of *PF-rules* and *LF-rules* respectively. *PF-* and *LF-rules* are implemented as a cascade of regular expressions.

1.7 PF- and LF-rules

The use of variables and features in PFs produce "abstract" PFs, which will have to be instantiated by PF-rules. For example, in the Spanish fragment that we will consider in section 3, the syntactic component produces PFs like *est-F-P dos persona-P*, which will have to be turned into *estas dos personas*.

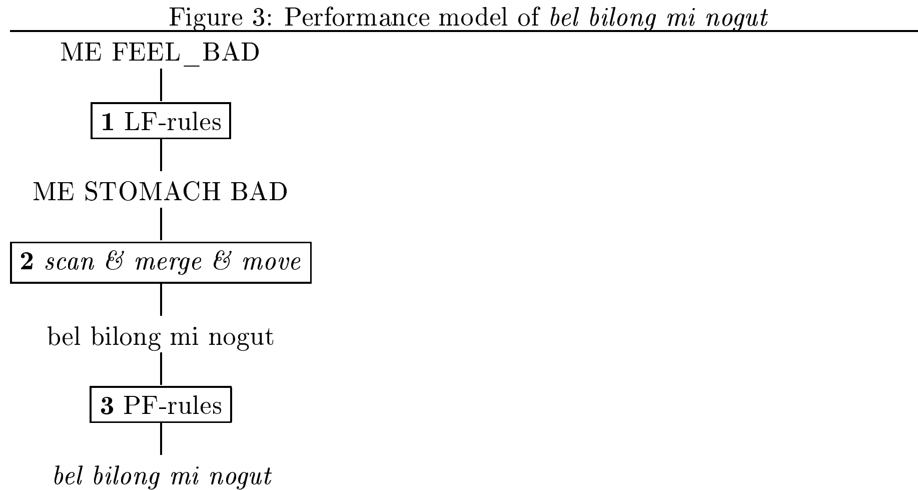
As for LF-rules, their need is felt at once when we try to build a translator. I give an example of a problem I faced in building the English and Tok Pisin

modules for the NSM translator: one "canonical sentence" of NSM is "I feel something bad". To express this in Tok Pisin, we have to say *bel bilong mi nogut*, literally "my stomach is bad". The Tok Pisin parser produces a "superficial" LF of the kind `BAD(STOMACH(ME))`, which is then turned into a "deep" LF like `FEEL_BAD(ME)`.

The Tok Pisin generator works in reverse: suppose it is fed with the output of the parsing of the English NSM sentence "I feel something bad," which will be something like `FEEL_BAD(ME)`. Before the derivation process starts, this LF will have to be "translated" into `BAD(STOMACH(ME))`.

Summarising, the Tok Pisin equivalent of "I feel something bad" is processed along the schema shown in figure 3 (to be read bottom-up for parsing and top-down for generation).[18]

Figure 3: Performance model of *bel bilong mi nogut*



If we look at the derivation of this sentence from a competence point of view, we could represent it like in figure 4.

We can think of such PF- and LF-rules as not, strictly speaking, part of syntax, but belonging already to the interface levels.

Putting all together, we obtain the following model of competence shown in figure 1.7.

to which corresponds the performance model shown in figure 1.7 (again, a top-down reading represents generation and a bottom-up reading represents parsing).

In the next section, we will look again at the derivation process, this time from the *performance* angle: in particular, we will see how the generator, given a specified LF, works out the corresponding LF. The parser, which will not described here, goes the opposite way.

[18] Cf. again **?**, pag. 226: "Much as there are phonological operations that apply exclusively for interface internal reasons there are numerous and powerful semantic processes that cannot and should not be reduced to syntax".

Figure 4: Competence model of *bel bilong mi nogut*

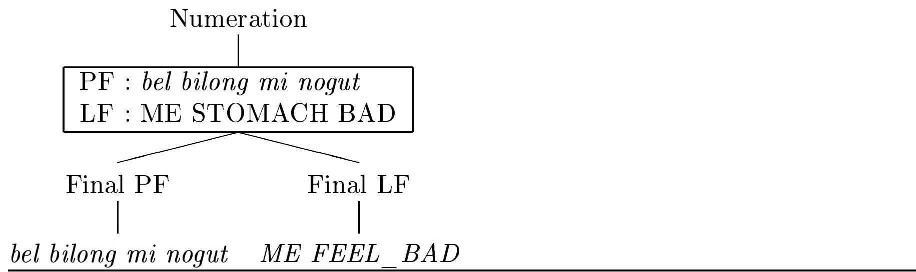


Figure 5: A model of Minimalist Grammar

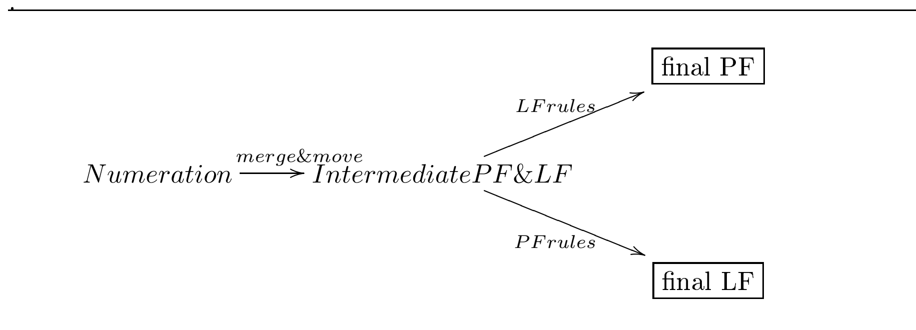


Figure 6: Generation and Parsing

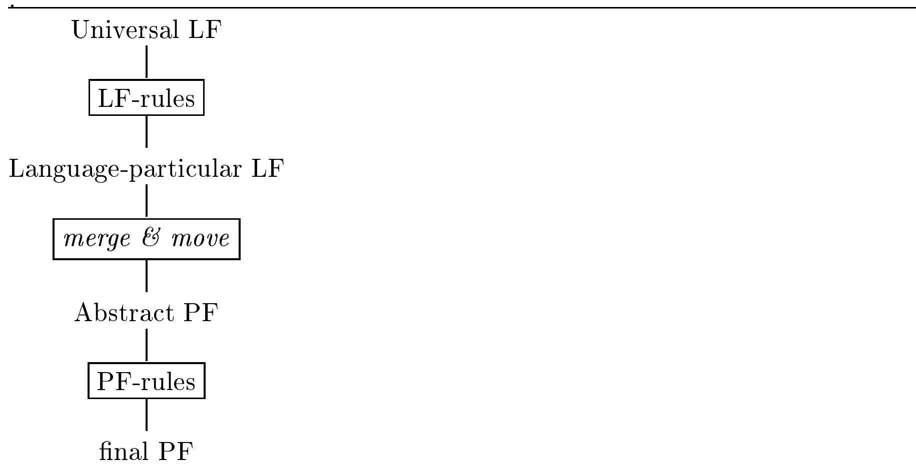


Figure 7: A Sample Lexicon

```
lexicon = {
    'LAVINIA' :  ['lavinia', 'n', '-k'],
    'TITUS' :  ['titus', 'n', '-k'],
    'PRAISE' :  ['praise', '=n', 'vt', '-v'],
    'PRED'   :  ['', '=vt', '+k', '=n', 'pred'],
    'INFL'   :  ['s', '=pred', '+v', '+k', 'i']
    }
```

2 Generation

A Minimalist Grammar consists mainly of a lexicon, represented as a Python dictionary. For our example, we shall take the very restricted lexicon of figure 2.

2.1 Running the generator

By running the generator in verbose mode, we get a trace of the generation process.

After having loaded the dictionary, and selected it as L2, we feed the generator with the input LF:

```
TITUS LAVINIA PRAISE PRED INFL DECL
```

2.2 The Generation Process

The generator works in a bottom-up shift-reduce fashion. While there are items on the list:

1. **scan**:
   (a) remove the first LF item from the list;
   (b) lookup its features in the dictionary, using LF as key, obtaining a minimal tree;
   (c) push the obtained minimal tere on the stack.
2. **merge, move**: Once a new item has been shifted on the top of the stack, succesive *merge* and *move* are attempted until no more operation is possible:
   (a) While the top item's first feature is $=f_i$ and the item immediately under it has f_i as its first feature:
      i. pop the two items from the stack;
      ii. apply *merge*
      iii. push the new formed item on the stack;

(b) Then, try to repeatedly apply *move* to the top item.

(c) And finally, if the top item's first feature is $=f_i$, repeat from point 2.a above.

(d) If there are LF items left, return to 1.

(e) When the LF string is exhausted, the top of the stack should contain the generated string, with its base feature.

2.3 Tracing the Algorithm

2.3.1 Scan

The first two operations will always be a *scan*, because a *merge* requires at least two items on the stack.

```
---------------------------
1 SCAN
 -- pf:  titus
 -- lf:  TITUS
 -- f:  ['::', 'n', '-k']
 -- stack :  []
---------------------------
2 SCAN
 -- pf:  lavinia
 -- lf:  LAVINIA
 -- f:  ['::', 'n', '-k']
 -- stack :  TITUS [n  ...];
---------------------------
```

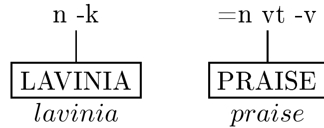
The tracing shows the stack before the scanned item is pushed. So, at the first scan, the stack is empty; when *LAVINIA* is scanned, the stack contains the minimal tree for $TITUS$.

No merge is possible, so another *scan* follows:

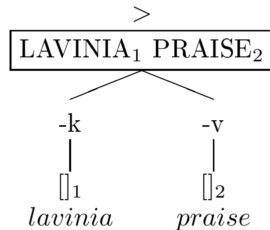
```
---------------------------
3 SCAN
 -- pf:  praise
 -- lf:  PRAISE
 -- f:  ['::', '=n', 'vt', '-v']
 -- stack :  TITUS [n  ...];  LAVINIA [n  ...];
---------------------------
```

2.3.2 Merge

As $PRAISE$ is scanned, a *merge* is possible: $PRAISE$ has $=n$ as its first features, and the element just "under" it on the stack has n as its categorial features.



Merging the two, we get the following tree:



When the generator performs a *merge*,

1. the categorial and selector features which have triggered the *merge* operation are deleted;

2. the LF of the selector, which is the head of the construction, is suffixed to the string representing the LF of the selected item (complement or specifier);[19]

3. As for the PF, two cases must be distinguished:

   (a) If the selected item does not have licensee features, it will not move further. So, *its PF can be concatenated to the PF of the head.*[20]

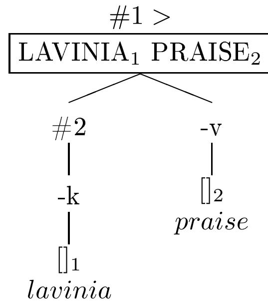
   (b) If, however, the selected item does have licensee features, *its PF cannot still be concatenated to the PF of the head, because the selected item will move when its licensee featuer will be attracted by a corresponding licensor feature.* This licensor could come into the structure at any later point.[21]

In the sentence we are generating, we cannot concatenate the PFs, because the item $LAVINIA$ has a $-k$ feature to be licensed. The tree is, as it were, "split" by the $-k$ feature:

[19] As LF is represented in reverse-Polish notation, suffixing the LF of the head means that the LF of the head is a predicate which takes the LF of the selected item as its argument.
As for the difference between complement and specifier, cfr Harkema (2001).

[20] Complements and specifiers

[21] This is *merge 3* of Harkema (2001, p. 86) and Stabler (1997).



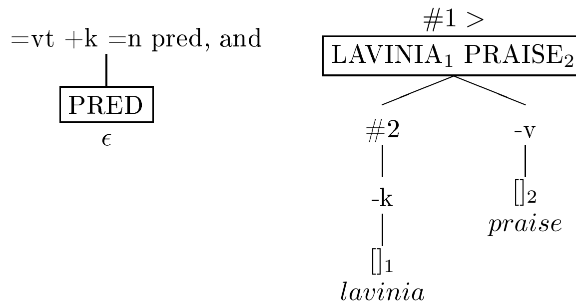
The tracing facility of the program represents it as follows:

```
---------------------------
4 MERGE
 -- lf:  LAVINIA PRAISE
#1  -- pf:  'praise'
    -- f:  [':', 'vt', '-v']
----------
#2  -- pf:  'lavinia'
    -- f:  [':', '-k']
---------------------------
```

At this point, the generator cannot apply *move*, because the top feature on the stack is a categorial one (*vt*). So, it goes on to scan the next item:

```
5 SCAN
 -- pf:
 -- lf:  PRED
 -- f:  ['::', '=vt', '+k', '=n', 'pred']
 -- stack :  TITUS [n  ...];  LAVINIA PRAISE [vt  ...];
---------------------------
```

The two topmost items on the stack are now



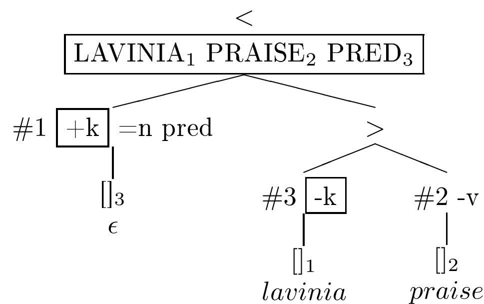
These items can be merged and, again, the PFs cannot be concatenated yet, because of the $-k$ features:

```
6 MERGE
 -- lf:  LAVINIA PRAISE PRED
#1  -- pf:  ''
    -- f:  [':', '+k', '=n', 'pred']
----------
#2  -- pf:  'praise'
    -- f:  [':', '-v']
----------
#3  -- pf:  'lavinia'
    -- f:  [':', '-k']

---------------------------
```

2.3.3 Move

The above trace represents the tree:



After having performed *merge*, the generator tries to apply *move*, and this time succeeds. The item $-k$ *lavinia* is fronted and its licensee feature is deleted. The phonetic form *lavinia* can be concatenated to the *pred* item triggering movement; however, as that item is phonetically null, no PF effect is seen:

```
7 MOVE
 -- lf:  LAVINIA PRAISE PRED
#1  -- pf:  'lavinia '
    -- f:  [':', '=n', 'pred']
----------
#2  -- pf:  'praise'
    -- f:  [':', '-v']

---------------------------
```

After deleting $+k$ feature from the head, the new topmost feature is another $=n$ selector. The next item on the stack is $n\,-k$ *titus*, therefore a new *merge* is possible. Again, the $-k$ feature in *titus* prevents the PFs from concatenating:

```
8 MERGE
 -- lf:  TITUS LAVINIA PRAISE PRED
#1  -- pf:  'lavinia '
    -- f:  [':', 'pred']
----------
#2  -- pf:  'titus'
    -- f:  [':', '-k']
----------
#3  -- pf:  'praise'
    -- f:  [':', '-v']

---------------------------
```

At this point, the topmost feature on the stack is the selector *pred*. The next move wil therefore be a *scan*:

```
9 SCAN
 -- pf:  s
 -- lf:  INFL
 -- f:  ['::', '=pred', '+v', '+k', 'i', '-f']
 -- stack :  TITUS LAVINIA PRAISE PRED [pred  ...];
---------------------------
```

The topmost feature $=pred$ triggers *merge*:

```
10 MERGE
 -- lf:  TITUS LAVINIA PRAISE PRED INFL
#1  -- pf:  's lavinia '
    -- f:  [':', '+v', '+k', 'i', '-f']
----------
#2  -- pf:  'titus'
    -- f:  [':', '-k']
----------
#3  -- pf:  'praise'
    -- f:  [':', '-v']

---------------------------
```

In the next step, the verb moves to (Spec,INFL) to check its $-v$ feature; then the subject noun *titus* moves to (Spec,INFL) too, to check case.

In the first instance of *move*, the PF of the verb *praise* is concatenated to the left of the INFL head $-s$, because it does not have any more licensees, so it will not move further:

```
11 MOVE
 -- lf:  TITUS LAVINIA PRAISE PRED INFL
#1  -- pf:  'praise s lavinia '
```

```
    -- f:  [':', '+k', 'i', '-f']
----------
#2  -- pf:  'titus'
    -- f:  [':', '-k']

---------------------------
```

The same happens to *titus* in the second instance of *move*:

```
12 MOVE
 -- lf:  TITUS LAVINIA PRAISE PRED INFL
#1  -- pf:  'titus praise s lavinia '
    -- f:  [':', 'i', '-f']

---------------------------
```

The last item is now pushed onto the stack:

```
13 SCAN
 -- pf:  .
 -- lf:  DECL
 -- f:  ['::', '=i', '+f', 'c']
 -- stack :  TITUS LAVINIA PRAISE PRED INFL [i  ...];
---------------------------
```

There remain two final *merge* and *move*:

```
14 MERGE
 -- lf:  TITUS LAVINIA PRAISE PRED INFL DECL
#1  -- pf:  '.'
    -- f:  [':', '+f', 'c']
----------
#2  -- pf:  'titus praise s lavinia '
    -- f:  [':', '-f']

---------------------------
15 MOVE
 -- lf:  TITUS LAVINIA PRAISE PRED INFL DECL
#1  -- pf:  'titus praise s lavinia  .'
    -- f:  [':', 'c']
```

The generator stops, as no more items remain to be scanned. The stack contains only one element, of category c. Note how the generator, in building the PF, has also rebuilt the original LF.

Figure 8: A Sample Spanish Lexicon

```
lexicon =  {
    'PERSON' :  ['persona-$num$', 'n:1', '-gen:F', '-num:$num$'],

    'GOOD' :  ['buen-$gen$-$num$',
               '=n:1', '+gen:$gen$', '+num:$num$',
               'n:2', '-gen:$gen$', '-num:$num$'],

    'TWO' :  ['dos',
             '=n:2', '+gen:$gen$', '+num:P', '>',
              'n:3', '-gen:$gen$', '-num:P'],

    'THIS' :  ['est-$gen$-$num$',
       '=n:5', '+gen:$gen$', '+num:$num$', '>',
               'n:7', '-gen:$gen$', '-num:$num$']

    }
```

3 Feature Values and Instantiation

Now we will see how feature checking instantiates morphological agreement. We start from the Spanish lexicon in figure 3.[22]

Some peculiarities:

- The symbol `'>'` after a licensor, which effects concatenation of the moved element to the right of the PF of the licensor, instead of the default left-attachment);
- more importantly, the same variables used in licensors and licensees are present in the morpho-phonetic features.

For example, the lexical entry *PERSON* is

```
'PERSON' :  ['persona-$num$', 'n:1', '-gen:F', '-num:$num$'],
```

with the two licensees *–gen* and *–num*. In Spanish, *persona* has feminine gender, so the value of the licensee *–gen* is *F*. As the value of the *num*(ber) feature, however, will depend on the context, it is represented here as a variable ($*num*$). When the number feature will be checked, this variable will get a value and will pass that value to the morpho-phonetic representation `persona-$num$`.

[22]To simplify matters, the adjective GOOD here takes directly the head noun as its complement. In a real grammar of Spanish, GOOD will have a base feature `a`, and will be selected as a specifier by a functional head like EvalP or SubjectCommentP, which will be part of the extended projection of the noun head (Scott, 2002; Cinque, 2009).

3.1 Persistent Features

The lexical entry for *GOOD* is:

```
'GOOD' :  ['buen-$gen$-$num$',
   '=n:1',  '+gen:$gen$', '+num:$num$',
           'n:2',   '-gen:$gen$', '-num:$num$'],
```

"Good" is an adjective[23], so both its gender and number will have to be instantiated by agreement with the head noun. This entry exemplifies also the solution I propose to "persistent features", i.e. features which do not delete after checking, because they will have to "pass on" their values further.

A subject noun phrase, for example, concords in number (and sometimes also in gender) with the predicate (est**a** person**a** **es** buen**a** vs. est**os** niñ**os** **son** buen**os**). But if these agreement features get deleted when they are checked between the head noun and the attribute, they will be no more available.

There is, however, a simple solution: we can simply put the same feature twice in the lexical entries for adjectives, both as a licensor and a licensee. In the above lexical entry for *GOOD* there are both $+gen$, $+num$, and $-gen$, $-num$, and, crucially, *they share the same variable.* As the adjective will check its $+gen$:\$$gen$\$ features against the $-gen$:F of a feminile noun, the variable \$$gen$\$ will become F in all of its instances, and the item *GOOD* will become

```
'GOOD' :  ['buen-F-$num$', '+num:$num$',
               'n:2', '-gen:F', '-num:$num$'],
```

We will see this at work in the following trace. We want to generate the Spanish noun phrase *esta persona*, represented as $THIS(PERSON)$ or, in reverse-Polish notation,

```
PERSON THIS
```

The generator will scan the first item and push onto the stack the resulting tree:

```
---------------------------
1 SCAN
 -- pf:  persona-$num.1$
 -- lf:  PERSON
 -- f:  ['::', 'n:1', '-gen:F', '-num:$num.1$']
 -- stack :  []

---------------------------
```

[23]Here its category is represented as *n:2*. A more realistic grammar will surely have the category a.

3.2 Variable Renaming

Note how the variable num has been renamed as num_1 (`$num.1$`). When an item is pushed onto the stack, all its variables are renamed to avoid conflicts with other variables with the same name. For example, in generating the Spanish *esta persona hace muchas cosas*, we do not want the subject NP to share the same num variable with the object.[24].

After the first item, the second is scanned:

```
2 SCAN
 -- pf:  est-$gen.2$-$num.3$
 -- lf:  THIS
 -- f:  ['::', '=n:5', '+gen:$gen.2$', '+num:$num.3$', '>',
        'n:7', '-gen:$gen.2$', '-num:$num.3$']
 -- stack :  PERSON [n:1  ...]

---------------------------
```

Note how the gender and number variable are renamed: We do not want the variable of this item to end up accidentally called like the one in other items, but we do want that the three instances of gen present in one and the same item get the same name, and so the three instances of num, or the "feature-passing" mechanism described above will not work. And that is exactly what we have: once *GOOD* is scanned, all its three num variables become num_3, which is different from the num_1 of the previous item.[25]

Returning to the sentence we are generating, now *merge* becomes possible:

```
3 MERGE
 -- lf:  PERSON THIS
#1  -- pf:  'est-$gen.2$-$num.3$'
    -- f:  [':', '+gen:$gen.2$', '+num:$num.3$', '>',
  'n:7', '-gen:$gen.2$', '-num:$num.3$']
----------
#2  -- pf:  'persona-$num.1$'
    -- f:  [':', '-gen:F', '-num:$num.1$']

---------------------------
```

As usual, the licensees in the complement prevent its phonetic features to be concatenated with those of the head.

Now, the `-gen:F` licensee will check against the `+gen:$gen.2$` of the head. After checking, the two features `-gen:F` and `+gen:$gen.2$` disappear, but the variable gen_2 will become F everywhere. So, after move, we have:

[24] This renaming operation is an analogous of α-conversion in λ-calculus.

[25] In this particular case, the two items will end up sharing the same value for these num_i variables, but the generator procedure cannot know it at this stage! It is only after *merge* and *move* will have applied that the two variable will eventually coincide.

```
4 MOVE
 -- lf:  PERSON THIS
#1  -- pf:  'est-F-$num.3$'
    -- f:  [':', '+num:$num.3$', '>',
         'n:7', '-gen:F', '-num:$num.3$']
----------
#2  -- pf:  'persona-$num.1$'
    -- f:  [':', '-num:$num.1$']

---------------------------
```

The $-gen$ licensee of the adjective has got the value F, which will be available for later checking, when for example the noun phrase will be selected by a predicate. In this way, we have preserved the value of the feature, without introducing any new machinery (e.g. the distinction between features which delete after checking and those which do not).

At this point, `+num:$num.3$` and check `-num:$num.1$` will effect another move. Since both values are variable, the two variables will become one:

```
5 MOVE
 -- lf:  PERSON THIS
#1  -- pf:  'est-F-$num.1$ persona-$num.1$'
    -- f:  [':', 'n:7', '-gen:F', '-num:$num.1$']
```

The generation process ends, producing the string

```
est-F-$num.1$ persona-$num.1$
```

3.3 Default Values

The above string has an unstantiated variable $\$num_1\$$. Using a standard minimalist terminology, this would cause the derivation to crash at PF.

To avoid this, uninstantiated variables get a *default value* before passing through PF-rules. Default values are declared in the grammar.

For example, for the Spanish mini-grammar we are considering, we could declare 'masculine' and 'singular' as the default values of, respectively, 'gender' and 'number'. The declaration looks like this in the Spanish file:

```
default = {'gen' : 'M', 'num' : 'S'}
```

By applying the default values, the string

```
est-F-$num.1$ persona-$num.1$
```

becomes:

```
est-F-S persona-S
```

Morpho-phonetic rules then transform this into:

```
esta persona
```

3.4 Another example of generation

Let us look at another example, generating the phrase *estas dos personas.* The initial LF is

```
PERSON TWO THIS
```

and the process begins, as usual, with two *scan* operations:

```
---------------------------
1 SCAN
 -- pf:  persona-$num.4$
 -- lf:  PERSON
 -- f:  ['::', 'n:1', '-gen:F', '-num:$num.4$']
 -- stack :  []

---------------------------
2 SCAN
 -- pf:  dos
 -- lf:  TWO
 -- f:  ['::', '=n:2', '+gen:$gen.5$', '+num:P', '>',
        'n:3', '-gen:$gen.5$', '-num:P']
 -- stack :  PERSON [n:1  ...]

---------------------------
```

The item “two” triggers plural agreement in the head noun (by its licensor `+num:P`). This renders the whole noun phrase plural (thanks to the licensee `-num:P`).

The numeral and the head noun are first merged:

```
3 MERGE
 -- lf:  PERSON TWO
#1  -- pf:  'dos'
    -- f:  [':', '+gen:$gen.5$', '+num:P', '>',
     'n:3', '-gen:$gen.5$', '-num:P']
----------
#2  -- pf:  'persona-$num.4$'
    -- f:  [':', '-gen:F', '-num:$num.4$']

---------------------------
```

then, gender and number features are checked by two successive *move* operations:

```
4 MOVE
 -- lf:  PERSON TWO
#1  -- pf:  'dos'
    -- f:  [':', '+num:P', '>', 'n:3', '-gen:F', '-num:P']
----------
#2  -- pf:  'persona-$num.4$'
    -- f:  [':', '-num:$num.4$']

---------------------------
5 MOVE
 -- lf:  PERSON TWO
#1  -- pf:  'dos persona-P'
    -- f:  [':', 'n:3', '-gen:F', '-num:P']

---------------------------
```

After this step, when the determiner is first scanned, then merged, the noun phrase under construction is already feminine and plural:

```
6 SCAN
 -- pf:  est-$gen.6$-$num.7$
 -- lf:  THIS
 -- f:  ['::', '=n:5', '+gen:$gen.6$', '+num:$num.7$', '>',
              'n:7', '-gen:$gen.6$', '-num:$num.7$']
 -- stack :  PERSON TWO [n:3  ...]

---------------------------
7 MERGE
 -- lf:  PERSON TWO THIS
#1  -- pf:  'est-$gen.6$-$num.7$'
    -- f:  [':', '+gen:$gen.6$', '+num:$num.7$', '>',
                 'n:7', '-gen:$gen.6$', '-num:$num.7$']
----------
#2  -- pf:  'dos persona-P'
    -- f:  [':', '-gen:F', '-num:P']

---------------------------
```

Now it's the turn of the determiner to check gender and number:

```
8 MOVE
 -- lf:  PERSON TWO THIS
#1  -- pf:  'est-F-$num.7$'
    -- f:  [':', '+num:$num.7$', '>',
```

```
          'n:7', '-gen:F', '-num:$num.7$']
----------
#2  -- pf:  'dos persona-P'
    -- f:  [':', '-num:P']

--------------------------
9 MOVE
 -- lf:  PERSON TWO THIS
#1  -- pf:  'est-F-P dos persona-P'
    -- f:  [':', 'n:7', '-gen:F', '-num:P']
```

And we are done. Spell-out will derive the phonetic form:

```
 ========= SPELL-OUT =========
 ==>   est-F-P dos personas
 ----
 ==>   estas dos personas
 ----
```

4 Conclusions

The generator described in the pesent article, together with the parser, has been already "put to use" in automatic translation between English and Tok Pisin NSMs, with encouraging resultates. Much remains to do, of course.

- A first "minimalist" question imposes itself: there are no constraints on feature composition of lexical item. We could envisage such improbable a lexical items as:

  `=v =v =v =v =n =n =n a PF LF`

  just as in early generative grammars it was possible to write base rules as:

  `V --> NP + A.`

  To make such rules impossible, X-bar theory was developed, as it is well known.

  In Zamblera (Forthcoming) I have attempted to advance some suggestions about how lexical representation could be constrained.

- Another important question concerns agreement: As noted in the literature (Cf. e.g. Chesi, 2004, 2007; Sigurðsson, 2006), *merge* must involve features agreement; especially Chesi's idea of merge as unification is very appealing from both a theoretical and a computational point of view.

  This idea can be readily implemented in this model if a select feature is immediately followed by all the licensees which represent the relevant agreeing features.

For example, suppose head `a` selects head `b` as its complement[26] only if the two heads agree in features f_1 and f_2. The following lexical entries capture exactly this situation:[27]

$$b \ -f_1 \ -f_2 \ \ldots \tag{11}$$

and

$$=b \ +f_1 +f_2 > \ldots a \ldots \tag{12}$$

If `a` selects `b` as its specifier, we will simply omit the $>$ symbol from the above representation, and *move* will adjoin the moved item to the left, yielding Spec $\prec$ Head linearization.

By the way, it would be desirable to have a uniform *move*, which only left-adjoins the goal to the probe[28]. In this case, we could keep the "right adjoining" move only to simulate *merge*-triggered agreement in a head-complement configuration. A head which selects both complement and specifier would thus have the following representation:

$$\begin{aligned} =&Comp \ +agr_i +agr_j > \ldots \\ &=Spec \ +agr_x +agr_y \ldots \\ &\qquad\qquad Base \ldots \end{aligned} \tag{13}$$

Finally, some plans/wishes for the future:

- Empirically adequate grammar modules should be written for a variety of languages, which can cover at least the NSM subset;
- A bottom-to-top parser builds tipically a lot of unnecessary structure, to be later discarded. A worthwhile enterprise would be developing of a top-down parsing and generation algorithm, like that in Harkema (2001) or Chesi (2004). The notion of *phase* could then be very useful; the present implementation, based on the "pre-*phases*" model of Stabler 1997, does not uses this concept;
- The grammar coverage could be extended to some larger subset than NSM;

[26] In Stabler's model, as well as in my implementation, there is a simple linearization rule: complements are spelled out on the right of their heads, and specifiers on the left.

[27] Recall that the $>$ symbol after a licensor effects the "right-adjoining" of the moved item. So, in this case, the resulting order is that of a simple *merge*.

[28] As Stabler (2011b) has shown, allowing *merge* and *move* to specify the adjoining direction does not increase the generative power of the grammar anyway.

- An interesting "variation on the minimalist theme" is represented by Brody's mirror theory,[29], which has been formalized by Kobele (2002). Kobele's formalization of Brody's "mirror theory" is very similar to Stabler's model. The Python program that I have developed contains a module which works derivations using Mirror Theory, though I have not tested it yet. Developing a Mirror Throretic Grammar for English NSM would be an interesting exercise;

- A question which is interesting also from a theoretical point of view: could there be variable-sharing among LF and syntactic features, as already PF has? Though I have not used this feature in the English NSM grammar, the program could already allow it "for free".

Appendix: The program

The program has, so far, only a command-line interface. A grammar for the parser is loaded by issing the command `l1 <filename>`, while the command `l2 <filename>` loads a grammar for the translator. After a grammar is loaded, the command `t <sentence>` translates a sentence from l1 to l2 (if no l2 is selected, the sentence is simply parsed and the LF is output. If no l1 is selected, the input must be a LF formula, and the output will be the l2 sentence generated from that formula. For example, given the lexicon-grammar defined above in the derivatrion of the sentence "Titus praises Lavinia" (section **??**) contained in the file "svo.py":

```
>> l1 svo
>> t titus praise s lavinia .
TITUS LAVINIA PRAISE PRED INFL DECL
```

is an example of parsing, while

```
>> l2 svo
>> t TITUS LAVINIA PRAISE PRED INFL DECL
titus praise s lavinia  .
```

is a sample generation.

The command `test` is useful in developing new grammar. After the command, the user can input sentences, which are first parsed. The LF thus obtained is fed back into the generator, and, if all goes well, the input sentence should be produiced again.

```
>>>
>> l1 svo
>> v
```

[29] Brody (2000).

```
>> test
?- titus praise s lavinia .
parse:  TITUS LAVINIA PRAISE PRED INFL DECL
generation: titus praise s lavinia  .
```

References

Belletti, Adriana (ed). 2004. *Structures and Beyond*. The Cartography of Syntactic Structures, vol. 3. Oxford University Press.

Boeckx, C. 2010. *Oxford Handbook of Linguistic Minimalism*. Oxford University Press.

Boeckx, Cedric (ed). 2006. *Agreement Systems*. Amsterdam, Philadelphia: John Benjamins.

Brody, Michael. 2000. Mirror Theory: Syntactic Representation in Perfect Syntax. *Linguistic Inquiry*, **31**, 29–56.

Chesi, Cristiano. 2004. *Phases and Cartography in Linguistic Computation. Toward a cognitively motivated computational model of linguistic competence.* Ph.D. thesis, University of Siena.

Chesi, Cristiano. 2007. An Introduction to Phase-based Minimalist Grammars: why *move* is Top-Down from Left-to-Right. *STiL Studies in Linguistics*, **1**.

Chomsky, Noam. 1957. *Syntactic Structures*. The Hague: Mouton.

Chomsky, Noam. 1965. *Aspects of the Theory of Syntax*. Cambridge, MA: The MIT Press.

Chomsky, Noam. 2001. *Derivation by Phase*. *In:*Kenstowicz (2001). Pages 1–52.

Chomsky, Noam. 2005. *On Phases*.

Cinque, Guglielmo. 1999. *Adverbs and functional heads: a cross-linguistic perspective*. Oxford University Press.

Cinque, Guglielmo (ed). 2002. *Functional structure in DP and IP*. The Cartography of Syntactic Structures, vol. 1. Oxford University Press.

Cinque, Guglielmo (ed). 2006. *Restructuring and functional heads*. The Cartography of Syntactic Structures, vol. 4. Oxford University Press.

Cinque, Guglielmo. 2009. *The Syntax of Adjectives. A Comparative study*. The MIT Press.

Haegeman, Liliane (ed). 1997. *Elements of Grammar*. Dordrecht: Kluwer Publications.

Harkema, Hendrik. 2001. *Parsing Minimalist Languages.* Ph.D. thesis, University of Califrnia, Los Angeles.

Hornstein, Norbert, Nunes, Jairo, & Grohmann, Kleanthes K. 2005. *Understanding Minimalism.* Cambridge Textbooks in Linguistics. Cambridge University Press.

Kenstowicz, M. (ed). 2001. *Ken Hale: A life in language.* Cambridge MA: MIT Press.

Kobele, Gregory. 2002. Formalizing Mirror Theory. *Grammars*, **5**(3), 177–221.

Retoré, C. (ed). 1997. *Logical Aspects of Computational Linguistics.* Springer.

Rizzi, Luigi. 1997. *The Fine Structure of the Left Periphery.*

Rizzi, Luigi (ed). 2004. *The Structure of CP and IP.* The Cartography of Syntactic Structures, vol. 2. Oxford University Press.

Scott, Gary-John. 2002. *Stacked Adjectival Modification and the Structure of Nominal Phrases.* Vol. 1 of Cinque (2002). Pages 91–120.

Sigurðsson, Halldór. 2006.

Stabler, Edward. 1997. *Derivational Minimalism. In:*Retoré (1997). Pages 68–95.

Stabler, Edward. 2011a. *Computational Minimalism. Acquiring & Parsing Languages With Movement.* John Wiley & Sons Inc.

Stabler, Edward. 2011b. *Computational perspectives on minimalism.* Pages 616–641.

Steedman, Mark. 2000. *The Syntactic Process.* MIT Press.

Wojdak, Rachel. 2005. *The Linearization of Affixes: Evidence from Nuu-chah-nulth.* Ph.D. thesis, The University of British Columbia.

Zamblera, Francesco. Forthcoming. NSM meets Minimalism. A Preliminary Minimalist Grammar of the English Semantic Metalanguage.