The English NSM generator grammar.

Francesco Zamblera

August 13, 2010

In Zamblera (2010) I presented the NSM-PROLOG notation for representing NSM texts in a language-independent manner.

In this paper, the English grammar is briefly described which allows the program `nsm-gen-pl` to generate English NSM sentences from NSM-PROLOG formulas explains the grammar notation, working out the English NSM grammar.

The grammar here described is superseded by the new PROLOG program `nsm-dalia-pl`, which can work both as generator and parser.

---

1 Introduction

I have developed the NSM-PROLOG notation[1] together with a program (`nsm-gen-pl`) which can read that notation and generate English NSM sentences, given a grammar.

The grammar is based on simple pattern-matching algorithm, which generates a sentence top-down from a full formula down into its constituents, until single predicates are reached, which are then replaced with the appropriate allolex.

The following section describes the outook of the source code of this "NSM-PROLOG-to-English" grammar[2].

2 The structure of a grammar

The grammar consists of various sections:

1. pattern-matching rules;
2. paradigms;
3. morphophonemic rules;
4. lexicon

[1] See Zamblera (2010).

[2] You can download the program from the author's website; the full source code of the English grammar is in the file `./nsm-pl-gen/grammars/eng_nsm.pl`.

3 The lexicon

The lexicon section of the gramamr file contains the NSM primes with their equivalents in the language whose grammar we are writing.

The lexicon is simply a list of entries of the form

(1) $$prime - "equivalent".$$

that is, the NSM prime is written first, followed by a minus sign, followed by the exponent of that prime between double quotation marks, and followed by a full-stop.

For example:

```
think - "think".
know - "know".
```

This notation will be enough for all the primes which have no allolexic variation. A prime with more than one exponent will be represented like this:

(2) $$prime(allolex) - "equivalent".$$

that is, the nsm prime is represented by a PROLOG unary predicate instead of an atom like before.

The argument of that predicate can be everything the gramamr writer chooses to differentiate the allolexes, with one restriction: the main allolex should be represented by a "0" (zero) argument, that is

(3) $$prime(0)$$

I have chosen numbers to identify the other allolexes: in this English-NSM grammar we have, for example:

```
someone(0) - "person".
someone(1) - "someone".
someone(2) - "anyone".
someone(3) - "everyone".
something(0) - "thing".
something(1) - "something".
sometihng(2) - "anything"
...
```

The lexicon contains not only the primes, but also the language-specific morphosyntactic items which are inserted in the frames by the grammar rules, so, fo example, the English lexicon will contain entries for the preposition "with", "to", the suffixes $-s$, $-ed$, and so on. Some examples:

```
with - "with".
to - "to".
of - "of".
about - "about".
in - "in".
...
```

```
% TMA

fut - "will".
ind_pres_3 - "s_form=".
past - "d_form=".
ptc - "ing_form=".
...
```

Grammatical elements such as English prepositions "with", "to", are stored in the lexicon like this:

```
with - "with".
to - "to".
about - "about".
...
```

These markers are not primes, but they are introduced by rules in certain frames. For example, the following rule realized the TOPIC argument of THINK with an "about"-phrase:

```
p(think,[TOP]) >> [think,about,TOP].
```

4 The grammar

The grammar is an (ordered) sequence rules pairing an NSM-PROLOG formula with its language-specific realization. In their simplest form, rules have the following format (we have already seen one in the previous section):

(4) $$Formula >> Frame$$

For example, the following construction specifies the English realization of the construction *think that + sentence*:

```
p(think,[prop(S)]) >> [think,that,S].
```

The construction to the left of the >> sign matches the phrase-level formula $p(think, [prop(S)])$, which specifies the sentential valence of the prime *think*. The template on the right inserts the PROLOG predicates `think` and `that`, which will be later looked up in the lexicon, and substituted with the actual English words:

```
think - "think".
that - "that".
```

When, during the generation process, a formula is met which matches the left-hand side of the construction, the formula is substituted for the template specified on the right hand. So, $p(think, [prop(S)])$ is replaced by $[think, that, S]$, where S is a sentence-level formula, to which the process will be applied recursively.

To see an example of how all this works, let's consider a fragment of a (very simplified) grammar and lexicon for English substantive phrases, which can generate phrases like "this very good person":

```
sp(Det,Q,Same,Eval,Head) >> [Det,Q,Same,Eval,Head].
very(Eval) >> [very,Eval].

this - "this".
very - "very".
good - "good".
someone - "person".
```

Suppose to feed the generator with the formula

```
sp(this,e,e,very(good),someone).
```

Suth a formula is put into a PROLOG list, and passed to the generator. The generator scans the list and, for each item in the list, tries to build another list with the translation of the expantion of that item. In out case, the list scanned by the generator has only one item, out initial formula, so it looks like this:

```
[sp(this,e,e,very(good),someone)]
```

The generator extracts the *sp* formula and tries first to find a lexical entry which matches the formula. Not finding any, as a second step, it tries to match the formula with the left-hand part of a construction. This time, it succeeds, and so, inserts into the new list the right hand part of the construction. As our formula was the only item, the original list is exhausted.

At this point, starting from our formula, the generator has built the following template:

```
[this,e,e,very(good),someone]
```

Now the generator applies the process recursively to the new list: each item in order is extrated, and matched against a lexical entry or a formula. When a match is found, it is added to the new list under construction. So, first of all, `this` is found in the lexicon, and the two `e`'s are simply skipped. The new list is now `["this"]`. When the generator extracts `very(good)`, however, no match in the lexicon is found.

So, the generator calls a "clone" of itself in order to expand the formula `very(good)`. That is to say, another instance of the generator is run with initial list `[very(good)]`. As before, the only item is extracted, and the rule

```
very(A) >> [very,A]
```

is selected. During the matching process, the variable A is instantiated to `good`. So, the clone of the generator returns to his "father" the following template:

```
["very","good"]
```

The generator adds this to the list it's building, which now looks like this:

```
["this", "very", "good"]
```

Now the last item is selected, *someone*, which matches the lexical entry

```
someone - "person".
```

The entry is added to the list, which is now

```
["this", "very", "good", "person"].
```

This new list is scanned again, for the last time, to check again for untranslated primes or unexpanded formulas. As this list consists only of phonemes, the generator ends its task.

Now let's see a more complex example: the construction for the prime *do*, with its valency:

```
p(do,[OBJ,PAT,INSTR,COMIT]) >> [do,OBJ,to/PAT,with/INSTR,with/COMIT].
```

the notation `to/PAT` means that the item `to` is inserted only if the content of the variable PAT is not the empy item `e` (this is in order to avoid inserting "dangling" prepositions in the frames).

Some further examples of rules now follow:

Valence rules for prime THINK

```
p(know,[way(S)]) >> [know,how,S].
p(know,[What,Top]) >> [know,What,about/Top].
p(know,[prop(sg(A,B,C)::SG)]) >> [know,colon,sg(A,B,C)::SG].
p(know,[prop(S)]) >> [know,that,S].
p(know,[wh(S)]) >> [know,S].
p(know,[What]) >> [know,What].
```

In the third rule above, the notation

```
sg(A,B,C)::SG
```

represents a *sentence group*. I have recently popted for a simplification of the representation of sentence groups, which are now simply PROLOG lists. The third rule above would become:

```
p(know,SG]) >> [know,colon,SG] // ['P' is_list(SG)].
```

where `// [p is_list(SG)]` is a *condition*.

4.1 Rules with Conditions

The previous formula exemplifies the most general form which a grammatical construction can take:

(5) $$Formula >> Frame//Conditions$$

where *Conditions* is a list of predicates. This form is used to set the allolexes and the morphological variants of the primes. Each item in the *Contitions* list must be either a PROLOG built-in predicate preceded b the operator $'P'$[3], or (much more often) a predicate written by the author of the grammar himself,

[3]The previous rule called the PROLOG predicate `is_list/1`, which checks whether its argument is a list.

and stored in the "paradigm" section of the grammar. All this is much more understandable by looking at an actual example from the English grammar.

We will now examine the full rule for the substantive phrase, which looks like this:

```
sp(Det,Q,Other,Eval,N)
      >> [Det1,Q,Other,Eval1,AGR,N,Eval2]
      // [
           np_select_eval(Eval,Other,Eval1,Eval2),
           np_num(Q,N,AGR),
           np_det(Q,AGR,Other,Det,Det2),
           poss_det(N,Det2,Det1)].
```

Note how the formula and the frame use different variables (e.g. `EVAL` in the formula and `EVAL1` and `EVAL2` in the frame, for the attribute slot). The different variables are likned in the paradigm

```
np_select_eval(Eval,Other,Eval1,Eval2),
```

which is defined as follows:

```
np_select_eval(like(X),same,e,like(X)).
np_select_eval(like(X),other,e,like(X)).
np_select_eval(like(X),e,e,like(X)).
np_select_eval(X,same(AS),X,as(AS)).
np_select_eval(X,same,X,e).
np_select_eval(X,other,X,e).
np_select_eval(X,e,X,e).
```

The PROLOG engin scans the clauses of the paradigm in the order given, and stops as soon as it finds a match. The match will instantiate the variables, providing the desired allolex. For example, suppose that the engine is generating from the formula

```
sp(e,two,e,like(me),someone)
```

When the formula is matched against the rule

```
sp(Det,Q,Other,Eval,N)
```

the variables *Det*, *Q*, *Other*, *Eval* and *N* are assigned the values of the slots in the formula, so $Det = e$, $Q = two$, $Other = e$, $Eval = like(me)$ and $N = someone$. Then the paradigm `np_subject_eval` is called, the variables *Other* and *Eval* have values, the other are uninstantiated. So the paradigm wil be called like this

```
np_select_eval(like(me),e,Eval1,Eval2),
```

The first two clauses of the paradigms will be attempted unsuccessfully, because their *Other* slot is already filled by another value than `e`, so cannot match. The third clause, however, can match:

Calling procedure:	like(me)	e	Eval1 (variable)	Eval2 (variable)
Matching clause:	like(X)	e	e	like(X)

Variables are assigned the values they match: so, after the match, $X = me$, $Eval = e$ and $Eval2 = like(me)$. In this way, the frame generated will be (after the other paradigms have done their work):

```
[e,two,e,e,more_than_one(0),person,like(me)]
```

As you can see, the value contained in $Eval$ is assigned to $Eval2$ if we have a postnominal modifier (as LIKE THIS, LIKE ME). Prenominal adjectives are assigned to $Eval1$. For example, when the rule is matched by the following formula:

```
sp(e,two,e,good,someone)
```

the paradigm is invoked with the values:

```
np_select_eval(good),e,Eval1,Eval2),
```

and the matching clause of the paradigm, this time, will be the last one:

```
np_select_eval(X,e,X,e).
```

the variable X is assigned `good`, so the other X in the third slot inherits this `good` and assigns it to $Eval1$, while, this time, $Eval2$ receives `e`:

Calling procedure:	good	e	Eval1 (← `good`)	Eval2 (← `e`)
Matching clause:	X (← `good`)	e	X (← `good`)	e

and the generated frame will be

```
[e,two,e,good,more_than_one(0),person,e]
```

5 Transformations

A transformation is a rule which replaces a frame with another. Transformations have the form:

(6) $$Frame_1 \Longrightarrow Frame_2 // Conditions.$$

for example:

```
[Det,Q,SameOrOther,Eval,PLUR,N,Like] =>
[Q2,Det1,Same,Q1,Other,Eval,PLUR1,N1,Like]
          // [h_allolex(Det,Q,N,Det1,N1),
              other_allolex(SameOrOther,Det1,Same,Other),
              check_q(Q,Det1,N1,Q1,Q2),
              check_plur(PLUR,Det1,PLUR1)].
```

The pattern matching and the calls to paradigms work exactly like in the previous case. For example, the paradigm `check_q` assigns the quantifier `all` its place in front of the whole noun phrase (represented in the second frame by the variable $Q2$).

6 Morphophonemics

The last type of rules are the morphological adjustments necessary to spell out the right inflected form. These rules need not be discussed here, because they will probably be dropped from future versions of the program, (if there will be any).

7 Conclusion

The grammar format was developed together with the NSM PROLOG notation. However, in the meantime, I have developed a new program which allows both parsing and generation, and thus, it has superseded the present generator-only program. The grammar format described here has just, if any, a historical interest, as a first attempt to develop NLP applications for NSM.

References

Zamblera, Francesco. 2010. The NSM-PROLOG notation. Draft.