Elvex – Documentation

Lionel Clément

Version 2.10 – February 2019

1   The program `elvex`

`Elvex` is an Natural Text Generator. It takes as an input a concept, a local lexicon, a compacted lexicon, and a grammar, and then outputs a text corresponding to the concept writing in Natural Language.

1.1   Installation

To install `Elvex`, simply download the software, compile it and install it :

```
tar xf elvex.xxx.tar.gz
cd elvex
./configure
make
sudo make install
```

Compilation and full installation of `Elvex`

The full compilation of `Elvex` requires the following programs on your machine :

— g++
— flex
— bison
— aclocal
— automake
— autoconf

The full command is :

```
aclocal
automake −fa
autoconf
./configure
make
sudo make install
```

1.2   Elvex command

The `elvex` command is :

```
elvex [options] [input]*
```

The options are :

-h|--help

Displays help and exits

-v|--version

Displays the version number and exits

-a|--reduceAll

Reduces all rules during the generation process. This mode allows fine
debugging and requires an understanding of the generation process to
be mastered.

-t|--trace

Displays the reduced rules during the generation process. This mode
allows one for a grammar debugging.

-r|--random

Displays only one randomly selected output.

-grammarFile <file>

File containing the grammar.

-lexiconFile <file>

File containing the lexicon.

-inputFile <file>

File containing the input.

-compactDirectory <directory>

Directory containing the files of the compacted lexicon.

```
-compactLexiconFile <filePrefix>
```
    Prefix of the compacted lexicon files `<directory>/<filePrefix>.tbl` and `<directory>/<filePrefix>.fsa`

```
-maxLength <integer>
```
    Defines the maximum length of the sequence to be produced.

```
-maxUsages <integer>
```
    Defines the maximum number of times the same rule can be used.

```
-maxCardinal <integer>
```
    Defines the maximum cardinality of the item sets.

```
-maxTime <integer>
```
    Defines the number of seconds before the program stops.

1.3   Command for grammar debugging

```
elvexdebug [options] [input]*
```

The `elvexdebug` command works the same way with same options but produces a `HTML` document on the standard output. You must use your favorite browser to view this document properly and review the computation.

The command `elvexdebug` is recommended to set the grammars in the most difficult cases.

1.4 `elvexbuildlexicon` for managing big lexicons

The command of `ElvexBuildLexicon` is
```
elvexbuildlexicon [global-option] <build|consult|test> <input>
```

Where the options are :

```
-h|--help
```
    Displays help and exits

```
-d <directory>
```
    Directory containing the files of the compacted lexicon.

```
-f <filePrefix>
```
    Prefix of the compacted lexicon files `<directory>/<filePrefix>.tbl` and `<directory>/<filePrefix>.fsa`.

Command to build a compacted lexicon

```
$buildlexiconmain -d <directory> -f <filePrefix> build <input>
```

Command to test a compacted lexicon

```
$buildlexiconmain -d <directory> -f <filePrefix> test
```

Command to consult a compacted lexicon

```
$buildlexiconmain -d <directory> -f <filePrefix> consult
```

The compacted lexicon is a set of lexical entries of the type

```
cat#lexeme form#FS
```

— `cat` is a lexical category
— `lexeme` is the lexeme given by the `PRED` feature
— `form` is the produced form (a string of UTF8 characters in version 2.10)
— `FS` is a feature structure

2 Grammar

An example will familiarize the reader with `Elvex` before we go any further. Let us take the case of the text generation of a nominal phrase in French.

Example

```
 1 @grammar
 2 /* **************************************************
 3  * Elvex grammar
 4  ************************************************** */
 5 NP → det N {
 6    ↓2 = ↑;
 7    ↓1 = ⇓2;
 8 }
 9
10 N → adj N {
11    [mod:<$Head :: $Tail>, $Rest];
12    [number:$inhNb] ⊂ ↑;
13    ↓1 = $Head ∪ [number:$inhNb, gender:$synthGd];
14    ↓2 = [$Rest, mod:$Tail];
15    [gender:$synthGd] ⊂ ⇓2;
16    ⇑ = ⇓2 ∪ [qual:yes];
17 }
18
19 N → n {
20    [mod:NIL];
21    ↓1 = ↑;
22    ⇑ = ⇓1;
23 }
```

2.1 Syntagmatic rules (*Context-Free Grammar*)

The first rule of grammar (line 5) defines `NP` as consisting of `det` and `N` from left to right.

`det` is a word; it is not defined in the grammar, but in the lexicon that we will see later.

Lines 10) and 19) follow the same logic and we have the context-free grammar :

```
NP → det N
N → adj N
N → n
```

The only sequence we can generate with this simple example is a noun, preceded by a determiner and possibly an adjective.

Syntagmatic rules are therefore made to describe the order of components of phrases, sentences and speeches.

Empty rules, optional terms

It is possible to write empty rules, i.e. components that have no realization in text production. It is also possible to put optional elements written into brackets signs [].

Example : The following rule gives all possible combinations for clitic pronouns placed before the verb.

```
CLITICS → [clr] [cld] [cla] [cld] [clg] [cll]
```

In French, the dative clitic (`cld`) is sometimes placed before, sometimes placed after the accusative clitic (`cla`) according to the person of the pronoun.

(1)   a.   *Jean*   ***me***   ***le***   *donne*
           Jean   **to_me**   **it**   gives
           Jean gives it to me

     b.   *Jean*   ***le***   ***lui***   *donne*
           Jean   **it**   **to_him**   gives
           Jean gives it to him.

According to this rule, the phrase `CLITICS` can also be empty, because all terms are optional.

Paradigms

Each element of a syntagmatic rule can be a list of terms separated by the character | instead of being a single term. In this case, the product phrase is constructed with one of the terms in the list.

Example : The following rule allows you to build a sentence without a subject, with a nominative clitic subject `cln` (*je*, *tu*, *il*, ...), or with a subject as a nominal group.

```
S → [ cln | NP] VP
```

A grammar written by using massively rules with alternatives and optional terms is equivalent to a very large grammar. This is a combinatorial problem. For example, the rule describing the order of clitic pronouns is equivalent to a set of 64 rules :

```
CLITICS →
CLITICS → clr
CLITICS → cld
...
CLITICS → clr cld
CLITICS → clr cl cla
...  ...
```

However, `Elvex` never builds these sets of rules and remains effective with very large grammars, even written in doing so. The use of alternatives and optional terms is therefore recommended to describe the order of words, without regard to the combinatorial problem mentioned above.

2.2   Operational semantics

Each rule contains a set of operations. These operations describe the functional properties of the language, which may or may not depend on the order of the words.

The basics

We will explain this semantics through the example given to the beginning of this section before giving details.

— The rules on lines 6 and 7 express the nature of the links that exist between `NP`, the name `N` and its determiner `det` ; the main one being that agreement in gender and number occurs between nouns and their specifier.

  — Rule 6) assigns the nominal phrase properties to the noun. The attributes of the name are noted $\downarrow 2$ (this is the second term of the right part of the rule), those of the nominal phrase $\uparrow$.
  — Rule 7) retrieves information from analyses of `N` to assign them to the determiner. The attributes of the determiner are noted $\downarrow 1$

(this is the first term of the right part of the rule), those from the name `N` Downarrow 2.

With these two rules, the noun will be chosen in the lexicon according to the concept to be expressed (attribute of the nominal phrase ↑). A female singular determiner will be chosen if the gender of the noun is feminine and the nominal phrase is singular.

This mechanism, which makes it possible to articulate lexical choices according to the concepts and syntactic properties according to lexical properties is at the heart of `Elvex` and implements two types of attributes : inherited attributes (noted $\uparrow$, $\downarrow_i$) and synthesized attributes (noted $\Uparrow$, $\Downarrow_i$).

Inherited attributes are calculated from left to right of the syntagmatic rules. Synthesized attributes are calculated from right to left. To put it simply, we can say that the inherited attributes convey information from the concept to the lexicon, and that the synthesized attributes convey information from the lexicon to higher layers of the syntactic analysis. Thus the rule 6) allows you to propagate the input to the lexicon to make a lexical choice, whereas conversely, rule 7) allows to retrieve the lexical information of the chosen name according to the concept expressed and to propagate this information (grammatical features, selection restrictions, phraseological units constraints, etc.) to the entire nominal phrase.

— The rules in lines 11 to 16 allow you to set the gender and number agreement of the noun and adjective and also to propagate the concepts that produce adjectives. We notice that in French, the number is a semantism expressed directly as a sign associated with the noun, whereas gender is a grammatical feature given by the lexicon. The first will be propagated from the concept to the noun, the second will be propagated from the lexicon to the layers regulating the agreement.

— Rule 11) is a *guard, i. e.* a condition to realize `N` → `adj N`. It expresses that the realization of an adjective is conditioned by the existence of a list `<$Head::$Tail>` as the value of an attribute `mod` in the inherited attribute *uparrow*.

`$Head` and `$Tail` are two variables that respectively define the head and the tail of a list, *i. e.* respectively the first item in the list and the rest of the list.

if the condition is verified, `$Head`, `$$Tail` and `$Rest` will be affected by values given by the subsumption [1] values of `[mod:<$Head::$Tail>, $Rest]` by ↑.

— Rule 12) assign to the variable `$inhNb` (*Inherited noun*) the value of the `number` feature of the inherited attribute *uparrow*. This allows you to retrieve the number given to the nominal phrase.

— Rule 13) assigns to the inherited attribute of the adjective the head of the list `$Head` unified [2] with a structure that gives the synthesized gender `$synthGd` (*synthesized gender*) and the inherited number `$inhNb`.

— Rule 14) assigns to the inherited attribute of the noun all values passed by inheritance (`$Rest`) and adds the feature `mod` with the value `$Tail`, which is the tail of the list that brings together all other modifiers of the noun (other adjective or a proposal for example).

— Rule 15) allows you to assign the variable `$synthGd` which is the gender synthesized from the noun.

— Finally rule 16) assigns to the synthesized attribute of the phrase the synthesized attribute of the noun and adds a feature `qual` which marks that this noun phrase is qualified (which makes it possible to restrict others modifications).

— The rules on lines 20 to 22 are not specific, except for the custody of rule 20) which implements the value `NIL`.

— Rule 20) is a guard which allows us to prevent the inherited attribute ↑ to contain any feature `mod`. `mod:NIL` means that the attribute `mod` does not exist in ↑.

— Rules 21) and 22) do not require any futher comment.

It becomes clear by examining this simple example that the order of operational rules is not the one given by grammar, but the one dictated by the availability of operands. Here, a possible order of application of the rules is 11), 12), 14), 14), 15), 13) and 16). In the version 2.10 of `Elvex` this order is calculated on the fly. In the subsequent versions a dependency graph will be built with guidelines for developing grammars on this particular aspect.

---

1. A feature structure `A` subsumes a feature structure `B` iif all the features of `B` are present in `A` with the same values or with values that subsume those of the values of `A`.

2. The unification of a feature structure `A` with a feature structure `B`, is the smallest feature structure `C` such that `A` subsumes `C` and `B` subsumes `C`.

The advice I can give to develop a grammar is to write the rules as the synthesized attributes are available.

We see that the syntagmatic rules are not produced linearly from left to right and from top to bottom, but in such a way that which is not determined by producing the first available elements.

Before going any further, let us give definitions on the ratings that allow us to understand the syntax of the rules.

Grammar syntax

A syntagmatic rule is written
$A \rightarrow B_1 B_2 \ldots B_K < operational\ semantics >$;
Where B_i is written :

— $[B_i]$ if the symbol B_i is optional.

— $C_1 | C_2 | \ldots | C_n$ if B_i is a choice between the C_i

B_i, C_i can be lexical categories or symbols of the grammar.

Operational Semantics

Notations Let the rule $A \rightarrow B_1 B_2 \ldots B_i \ldots B_k < operational\ semantics >$

— $\uparrow$: refers to the attribute inherited from A.

— $\downarrow_i$: refers to the attribute inherited from Bi.

— $\Uparrow$: refers to the synthesized attribute of A.

— $\Downarrow_i$: refers to the synthesized attribute of Bi.

The named variables are noted by an identifier preceded by « $ », the free variables are noted « _ », the constant values are noted by identifiers.

The feature structures are noted $[line_1, line_2, \ldots, line_k]$ where $line_i$ is

— is a free variable

— is a named variable

— is `FORM:"<form>"` a feature for a literal form

— is $attr_i : val_i$

    — with $attr_i$ the name of an attribute or `PRED` or `FORM`

    — and with val_i the value of an attribute that can be :

        — A named variable

        — A free variable

— A zero value (`NIL`)

— A set of atomic constants separated by |

— A literal constant (number)

— A feature structure

— A list

The lists are noted

— $< l_1, l_2, \ldots, l_k >$ with l_i as feature structures

— $< head :: tail >$ with *head* as a feature structure and *tail*, a list. *head* refers to the first item on the list, *tail* the rest of the list.

If one wishes to bring an operational semantics to the syntagmatic rule, one defines $< operational\ semantics >$ which is a list of rules in brackets.

Each rule is written :

— {<list of rules >} A list of rules.

— `attest <expr>`; Explicit guard where `<expr>` is an boolean expression that must be checked.

— `[attr, attr, ...,..., attr]`; Guard where the feature structure must subsume *uparrow*. The guard may contain the feature `PRED` which represents the concept to be generated. Thus, a concept is not necessarily linked to a lexical entry with `Elvex`, unlike usual text generation systems ; it can produce a syntactic constraint.

— `print <expr>`; Displays the expression at standard output (useful for debugging).

— `println <expr>`; Does the same and displays a new line.

— `<expr1> = <expr2>`;

An assignment of the expression $< expr1 >$ with the value of the expression $< expr2 >$. The expression $< expr1 >$ is a complex variable, or simple variable, the expression $< expr2 >$ is a constant or a complex variable or a simple variable such that there is a environment [3] of the syntagmatic rule to solve it.

The assignments of $\uparrow$ and $\Downarrow_i$ are not possible, as these attributes are implicitly assessed during the text generation and modification would result in inconsistencies.

The assignments are limited to these cases :

---

3. An environment is an application ϕ from E to F such that E is a set of variables and F a set of constants or variables. The resolution of a variable α in an environment is $\beta = \phi^k(\alpha)$ such that β is a constant. It should be noted that `Elvex` also handles complex variables. In this case $\phi^k(\alpha)$ is replacing variables in terms by their constants.

— Assign a list

  — <...> = <...> Assigns a list to another one.

  — <...> = $X Assigned by the value of the variable $X

— Assign an inherited attribute

  — ↓i =[...] Assigned by a constant or variable feature structure.

  — ↓i = ↑ Assigned by an inherited attribute.

  — ↓i = ... ∪ ... Assigned by the result of the unification of two expressions (feature structures or attributes).

  — ↓i = ⇓j Assigned by a synthesized attribute.

— Assign a synthesized attribute

  — ⇑ = $X

  — ⇑ = [...]

  — ⇑ = ↑

  — ⇑ = ... ∪ ...

  — ⇑ = ⇓j

— Assign a simple variable

  — $X = $Y

  — $X = a Assigned by a constant or a literal.

  — $X = <...>

  — $X = [...]

  — $X = ↑

  — $X = ... ∪ ...

  — $X = ⇓j

  — $X = <expr> by evaluating an arithmetical or logical expression.

— [...] ⊂ <expr>;
  A feature structure is subsumed

  — [...] ⊂ ↑ by an inherited attribute.

  — [...] ⊂ ⇓j by a synthesized attribute

  — [...] ⊂ $X by the value of a variable.

— `if (<boolExpr>) <rules>`

  Interprets operational semantics `<rules>` if and only if the expression `<boolExpr>` is verified.

— `if (<boolExpr>) <rules> else <rules>`

  Interprets operational semantics `<rules1>` if and only if `<boolExpr>` is verified, otherwise interpret `<rules2>`.

A logical expression `<booExpr>` is

— `<boolExpr>` $\vee$ `<boolExpr>` Or

— `<boolExpr>` $\wedge$ `<boolExpr>` And

— $\neg$ `<boolExpr>` Negation

— `<boolExpr>` $\Rightarrow$ `<boolExpr>` Material implication

— `<boolExpr>` $\Leftrightarrow$ `<boolExpr>` Biconditional

— `<expr>` $==$ `<expr>` Equality

— `<expr>` $\neq$ `<expr>` Difference

— `<expr>` Expression evaluated as Boolean : A feature structure is evaluated as true if it is not `NIL` or $\bot$ (result of the failure of unification between two feature structures).

The expressions are :

— Integer

— String

— `<expr>` $\cup$ `<expr>` Unification of two feature structures

— $\uparrow$

— $\Uparrow$

— $\downarrow_i$

— $\Downarrow_i$

— `[...]` Feature structure

— `NIL` Null value of a feature

— `a|b...` Constant

— `$xxx` Variable

— `<...>` List

Let's add algebraic expressions and numbers :

— `<expr>` $<$ `<expr>`

— `<expr>` $\le$ `<expr>`

— `<expr>` $>$ `<expr>`

— `<expr>` $\ge$ `<expr>`

— `<expr>` $+$ `<expr>`

— `<expr>` $-$ `<expr>`

— `<expr>` $*$ `<expr>`

— `<expr>` $\%$ `<expr>`

— `<expr>` $/$ `<expr>`

— $-$ `<expr>`

— Double-precision floating-point format number

Variable evaluation

Variables are evaluated locally at a syntagmatic rule. This defines the scope of a variable.

Here are different methods for solving variables :

— Direct assignment

```
1  S → S {
2     [ vform : NIL ] ;
3     [ tense : $tense ,  aspect : $aspect ,  mode : $mode ]  ⊂ ↑ ;
4     if  ( ! $tense )
5        $tense=present ;
6  ...
7  }
```

Line 5 indicates that the value of the variable `$tense` is `present` by default (in the absence of another value previously given). The value of `$tense` is eventually given by the inherited attribute ↑ online 3.

Note that the order of evaluation of the rules is indifferent and that it is not necessary to precede rule (3) in the order that they are listed.

— Guard

```
S → NP VP {
   [ subject : $inheritedSubject ,  $Rest ] ;
...
}
```

The value of the variable `$inheritedSubject` is given by the *uparrow* subject feature. The value of the variable `$Rest` is the feature structure that remains when you remove the feature `subject`.

— Subsumption

```
S → NP VP {
. . .
   [ subject : $synthesizedSubject ]  ⊂  ⇓2;
. . .
}
```

The value of the variable `$synthesizedSubject` is given by the `subject` feature of ⇓ 2.

— Binding a variable within a syntagmatic rule

```
S → NP VP {
   [ subject : $inheritedSubject ,  $Rest ];
   [ subject : $synthesizedSubject ]  ⊂  ⇓2;
   ↓1  =  [ subject : $inheritedSubject ]
                    ∪  [ subject : $synthesizedSubject ];
. . .
}
```

The variables `$synthesizedSubject` and `$inheritedSubject` are both used within this rule to be transmitted to the `subject` feature of the inherited attribute ↓ 1.

— Linking a variable within a lexical entry

```
a           v [PRED: brouillard ,  subjC : [FORM:" il "] ,
                objC : [PRED: brouillard ,  det : yes ,
                def : no ,  part : yes ,  neg : $Neg ] ,
                locCl : [PRED: _pro ,  number : sg ,
                            person : three ] ,
                vform : tensed ,  vtense : present ,
                mode : indicative ,
                subj : [ person : three ,  number : sg ] ,
                neg : $Neg ,  fct : none ]
```

The variable `$Neg` is linked in this lexical entry to the expression "it is foggy" so that its use as negative propagates a negation of the noun :

   (2)   a. *Il y a du brouillard* (it is foggy)

    b. *Il n'y a pas de brouillard* (There is no fog)

    c. **Il y a de brouillard* ()

    d. **Il n'y a pas du brouillard*

2.3 Lexicon

The lexicon is a list of entries separated by semicolons.

Each entry defines the sequence to be produced, followed by a disjunction (noted by the character |) of lexical categories optionally followed by a feature structure.

The sequence to be produced may be :

— A string of characters between "", in which case the sequence will be exactly this one. Note that the sequence may be empty.

— A string of characters between "" containing variables. The sequence produced will be this one with the variables substituted by their respective values.

— `FORM` : the sequence produced will literally be the one given by the feature `FORM`. `Elvex` version 2.10 only provides for constant, subsequent versions will also have complex variables.

The category is an identifier, namely the final symbol of grammar.

The feature structure allows to represent all the constraints imposed by the lexicon.

A few examples :

— `"?" interrogativeDot;`

— `"tag-$Line" tag[line:$Line];`

— `FORM title;`

— `"elles-mêmes" itself [gender:fm, number:pl, person:three, itself:yes];`

— `serai aux_être [aux:être, voice:active, finite:yes,`
`    mode:indicative,  vtense:future_anterieur, vform:tensed,`
`  subj:[person:one, number:sg]];`

```
—  // Lexical input "temperature" producing
   "La température est de xxx degrés" (the temperature is xxx degrees Celsius"
   est v[PRED:température,
   subjC:[PRED:température, number:sg, det:yes, def:yes],
   pobjC:[PRED:degré, number:pl, pcas:de, det:yes,
   detNum:[PRED:num, value:$Deg]],
   vform:tensed, vtense:present, mode:indicative,
   subj:[person:three, number:sg], fct:none, value:$Deg]

   "la température est douce" (the temperature is mild)
   |v[PRED:température, subjC:[PRED:température, number:sg,
   det:yes, def:yes], modC:[PRED:doux, number:sg, gender:fm],
   vform:tensed, vtense:present, mode:indicative,
   subj:[person:three, number:sg], fct:few]

   "la température est élevée" (the temperature is high)
   |v[PRED:température, subjC:[PRED:température, number:sg,
   det:yes, def:yes], modC:[PRED:élevé, number:sg, gender:fm],
   vform:tensed, vtense:present, mode:indicative,
   subj:[person:three, number:sg], fct:much];

—  // Lexical entry of "temperature".
   // producing "Il fait chaud" (It's hot)
   fait v[PRED:température, subjC:[FORM:"il"],
   modC:[PRED:chaud, number:sg, gender:ms], vform:tensed,
   vtense:present, mode:indicative,
   subj:[person:three, number:sg], neg:$Neg, fct:high]

   //"Il fait très chaud" (It's very hot)
   |v[PRED:température, subjC:[FORM:"il"], modC:[PRED:chaud,
   number:sg, gender:ms, mod:<[PRED:très]>], vform:tensed,
   vtense:present, mode:indicative, subj:[person:three,
   number:sg], neg:$Neg, fct:very_high]

   //"Il fait très froid" (It's very cold)
   |v[PRED:température, subjC:[FORM:"il"], modC:[PRED:froid,
   number:sg, gender:ms, mod:<[PRED:très]>], vform:tensed,
   vtense:present, mode:indicative, subj:[person:three,
   number:sg], neg:$Neg, fct:very_low]
```

```
//"Il fait froid" (It's cold)
|v[PRED:température, subjC:[FORM:"il"], modC:[PRED:froid,
number:sg, gender:ms], vform:tensed, vtense:present,
mode:indicative, subj:[person:three, number:sg],
neg:$Neg, fct:low];

// Lexical entry of "rain" producing
// "Il y a une pluie fine" (A fine rain fell)
a v[PRED:pleuvoir, subjC:[FORM:"il"],
objC:[PRED:pluie, det:yes, number:sg, qual:yes,
def:no, mod:<[PRED:fin, pos:post]>],
locCl:[PRED:_pro, number:sg, person:three],
vform:tensed, vtense:present, mode:indicative,
subj:[person:three, number:sg], neg:$Neg, fct:few]

// "il y a un peu de pluie" (It is raining a bit)
|v[PRED:pleuvoir, subjC:[FORM:"il"],
objC:[PRED:pluie, det:yes, number:sg,
detForm:[FORM:"un peu de"]],
locCl:[PRED:_pro, number:sg, person:three],
vform:tensed, vtense:present, mode:indicative,
subj:[person:three, number:sg], neg:$Neg, fct:not_much]

// "il y a beaucoup de pluie" (There is plenty of rainfall)
|v[PRED:pleuvoir, subjC:[FORM:"il"],
objC:[PRED:pluie, det:yes, number:sg,
detForm:[FORM:"beaucoup de"]],
locCl:[PRED:_pro, number:sg, person:three],
vform:tensed, vtense:present, mode:indicative,
subj:[person:three, number:sg], neg:$Neg, fct:much]

// "il y a de fortes précipitations" (It is raining so much)
|v[PRED:pleuvoir, subjC:[FORM:"il"],
objC:[PRED:précipitation,
        det:yes, number:pl, qual:yes, def:no, mod:<[PRED:fort,
        pos:pre]>], locCl:[PRED:_pro, number:sg, person:three],
        vform:tensed, vtense:present, mode:indicative,
        subj:[person:three, number:sg], neg:$Neg, fct:a_lot];
```

2.4 Input

The input is summarized by the type of phrase or text you want to generate, followed by a feature structure containing the concept and the illocutionary structure to be expressed.

`PRED` allows you to define the named concept to be generated.

examples

— `sentence [PRED:raining, neg:yes]`

Simple entry for the concept of *raining* producing

(3) a. *Il ne pleut pas* (It is not raining)

b. *Il n'y a pas de pluie* (There is no rain)

— `sentence`

```
  [PRED:to_ask,
   i:[PRED:menuisier, id:3, number:sg, gender:ms, def:yes],
   iii:[PRED:apprenti, id:6, def:yes, gender:ms, number:sg],
   ii:[PRED:scier,
       i:[idref:6],
       ii:[PRED:poutre, number:sg, def:yes]
       ]
  ]
```

Complex concept where the features `id` and `idref` make it possible to constrain co-references.

This entry produces

(4) *Le menuisier demande à l'apprenti   de scier la poutre*
     Carpenter   asks   the apprentice to saw   the beam
     The carpenter asks the apprentice to saw the beam

— `sentence`

```
  [PRED:raining,
   tense:future,
   modSType:time,
   modS:<[FORM:"lundi 18 mars 2019", type:time]>
  ]
```

Concept producing an adverbial on the basis of the constant "lundi 18 mars 2019" (Monday, the 18th of March 2019). This entry produces :

(5) *Il va pleuvoir le lundi 18 mars 2013.* (It will rain on Monday, March 18, 2013)

— sentence
```
[PRED:cause,
 i: [PRED:raining]]
 ii: [PRED:se_couvrir, i:[PRED:you], modality:devoir]
 ]
```
Input for the following generated sentences :

(6) a. *Il pleut, tu dois te couvrir* (It is raining, you have to dress warmly)

  b. *Il pleut, par conséquent tu dois te couvrir* (It is raining, therefore you have to dress warmly)

  c. *Tu dois te couvrir s'il pleut* (You have to dress warmly if it is raining)

  d. *Tu dois te couvrir car il pleut* (You have to dress warmly because it is raining)

  e. *S'il pleuvait, tu devrais te couvrir* (If it rained, you should dress warmly)

  f. *S'il pleut, tu devras te couvrir* (If it is raining, you should dress warmly)

  g. *Il pleuvrait, tu devrais te couvrir* (It would rain, you should dress warmly)

— sentence
```
[PRED:initiative,
 i:[PRED:Jean],
 objC:[lexFct:Magn, mod:<[PRED:audacieux]>],
 lexFct:Oper1
 ]
```
Input containing features to fix lexical functions. The production is

(7) *Jean prend une initiative très audacieuse.* (Jean takes a very bold initiative)

— sentence
```
[PRED:initiative,
 i:[PRED:Jean],
 objC:[lexFct:Magn],
 lexFct:Oper1
 ]
```
Input for

(8)   *Jean prend une belle initiative.* (Jean takes a fine initiative)

```
—  sentence
       [PRED:to_break,
        i:[PRED:Jean],
        ii:[PRED:carafe, number:sg, def:yes,
            mod:<[PRED:little],[PRED:white]>],
        diathesis:passive, tense:past, neg:yes,
        modV:<[PRED:roughly]>
       ]
```

Input containing a negative passive feature. The production is

(9)   *La petite carafe blanche n'avait pas été cassée brutalement par Jean.* (The small white carafe had not been broken brutally by Jean.)

3

3.1   Design pattern