Elvex – Documentation

Lionel Clément

Version 2.13.0 – (to be updated)

1 Grammar by sample

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

```
NP → det N {
    ↓2 = ↑;
    ↓1 = ⇓2;
    ⇑ = ⇓2;
}

N → adj N {
    [mod:<$Head::$Tail>, $Rest];
    [number:$inhNb] ⊂ ↑;
    ↓1 = [number:$inhNb, gender:$synthGd, $Head];
    ↓2 = [mod:$Tail, $Rest];
    [gender:$synthGd] ⊂ ⇓2;
    ⇑ = ⇓2 ∪ [qual:yes];
}

N → n {
    [mod:NIL];
    ↓1 = ↑;
    ⇑ = ⇓1;
}
```

1.1 Syntagmatic description (*Context-Free Grammar rules*)

The first rule of grammar (line 1) defines `NP` as consisting of `det` and `N` from left to right.

`det` is a word ; it is defined in the lexicon that we will see later.

Lines 7) and 16) 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 one or several adjectives.

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

1.2 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 lines 2) and 3) 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.
  - Line 2) assigns the given nominal phrase properties to the noun. The attribute of the name is noted $\downarrow 2$ (this is the second term of the right part of the rule), and nominal phrase's is noted $\uparrow$.
  - Line 3) retrieves information from analyses of `N` to assign them to the determiner. The attribute of the determiner is noted $\downarrow 1$ (this is the first term of the right part of the rule), the attribute from the name `N` is noted $\Downarrow 2$.
  - Line 4) assigns the analyse of `N` to the synthesized attribute `NP`.

  With the two first lines, the noun will be chosen in the lexicon according to the concept to be expressed (attribute of the nominal phrase

$\uparrow$). 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 inherited attributes allow you to propagate the input to the lexicon to make a lexical choice, whereas conversely, synthesized attributes allow 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 lines 8) to 13) 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.

  — Line 8) is a *guard*, *i. e.* a condition to realize `N` $\rightarrow$ `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 $\uparrow$.

  — Line 9) assign to the variable `$inhNb` (*Inherited noun*) the value

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`.

of the `number` feature of the inherited attribute *uparrow*. This allows you to retrieve the number given to the nominal phrase.

  - — Line 10) 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 `$inhNb`.
  - — Line 11) assigns to the inherited attribute of the noun all values passed by inheritance (`$Rest`) and adds the feature `mod` with the value `$Tail`, which is the tail of the list that brings together all other modifiers of the noun (other adjective or a proposal for example).
  - — Line 12) allows you to assign the variable `$synthGd` which is the gender synthesized from the noun.
  - — Finally line 13) 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 17) to 19) are not specific, except for the custody of rule 17) which implements the value `NIL`.
  - — Line 17) is a guard which allows us to prevent the inherited attribute $\uparrow$ to contain any feature `mod`. `mod:NIL` means that the attribute `mod` does not exist in $\uparrow$.
  - — Lines 18) and 19) 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 8), 9), 11), 12), 10), and 13). 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.

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.

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`.

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**(dative) **it**(accusative) gives
Jean gives it to me

b. *Jean* ***le*** ***lui*** *donne*
Jean **it**(accusative) **to_him**(dative) gives
Jean gives it to him.

According to this rule, the phrase `CLITICS` can also be empty, due to the fact that each term is 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. 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 cld cla
... ...
```

However, Elvex never builds these set 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.

2 Elvex notations

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.

2.1 Operational Semantics

Notations Let the rule $A \ \rightarrow B_1 B_2 \ldots B_i \ldots B_k$

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

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

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

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

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

The feature structures are noted $[f_1, f_2, \ldots, f_k]$ where f_i is a feature as

— a linked variable feature

— an anonymous variable feature

— `HEAD:<identifier of a lexeme>` a feature for a predicate

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

— a pair $attr_i : val_i$

  — with $attr_i$ an named attribute

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

  — A named variable

  — An anonymous variable

  — A nil value (`NIL`)

  — An atomic constant

  — A set of atomic constants separated by `|`

  — A literal constant (number)

  — A feature structure

  — A list of feature structures

The lists are noted

— $< f_1, f_2, \ldots, f_k >$ with f_i as feature structures or lists

— $<$ *head* :: *tail* $>$ with *head* as a feature structure or a list 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.

— `[f_1, feat_2, \dots, f_k];` Guard where the feature structure must subsume ↑. The guard may contain the feature `HEAD` 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. And a syntactic constraints may be not joined to a specific concept.

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

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

— `<expr_1> = <expr_2>;`

An assignment of the expression $< expr_1 >$ with the value of the expression $< expr_2 >$. The expression $< expr_1 >$ is a complex variable, or simple variable, the expression $< expr_2 >$ 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 :

- Assign a list
  - `<...> = <...>` Assigns a list to another one.
  - `<...> = $X` Assigned by the value of the variable `$X`
- Assign an inherited attribute
  - `↓i = $X` Assigned by the value of the variable `$X`
  - `↓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 = [...]`

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.

  - $X = $\uparrow$
  - $X = ... $\cup$ ...
  - $X = $\Downarrow$j
  - $X = <expr> by evaluating an arithmetical or logical expression
  - $X = <search> by evaluating a lexical entry (see below)

- Subsume a feature structure

  A feature structure is subsumed

  - [...] $\subset$ $\uparrow$ by an inherited attribute.
  - [...] $\subset$ $\Downarrow$j by a synthesized attribute
  - [...] $\subset$ $X by the value of a variable.
  - [...] $\subset$ <search> by a lexical entry

- if (<boolExpr>) <rules>

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

- if (<boolExpr>) <rules_1> else <rules_2>

  Interprets operational semantics <rules_1> if and only if <boolExpr> is verified, otherwise interpret <rules_2>.

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 $\perp$ (result of the failure of unification between two feature structures).

The expressions are :

- Integer evaluated as true if not zero
- Double evaluated as true if not zero

- String evalued as true if not empty
- <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 evaluated as true
- $X Variable
- <...> List evaluated as true if not null (<>)

Let's add algebraic expressions and numbers :

- <expr> $<$ <expr>
- <expr> $\leq$ <expr>
- <expr> $>$ <expr>
- <expr> $\geq$ <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[HEAD:brouillard, subjC:[FORM:"il"],
                 objC:[HEAD:brouillard, det:yes,
                 def:no, part:yes, neg:$Neg],
                 locCl:[HEAD:_pro, number:sg,
                         person:three],
                 vform:tensed, vtense:present,
                 mode:indicative,
                 subj:[person:three, number:sg],
                 neg:$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*

2.2 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[HEAD:température,
subjC:[HEAD:température, number:sg, det:yes, def:yes],
pobjC:[HEAD:degré, number:pl, pcas:de, det:yes,
detNum:[HEAD: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[HEAD:température, subjC:[HEAD:température, number:sg,
det:yes, def:yes], modC:[HEAD: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[HEAD:température, subjC:[HEAD:température, number:sg,
det:yes, def:yes], modC:[HEAD: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[HEAD:température, subjC:[FORM:"il"],
modC:[HEAD: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[HEAD:température, subjC:[FORM:"il"], modC:[HEAD:chaud,
number:sg, gender:ms, mod:<[HEAD: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[HEAD:température, subjC:[FORM:"il"], modC:[HEAD:froid,
number:sg, gender:ms, mod:<[HEAD:très]>], vform:tensed,
vtense:present, mode:indicative, subj:[person:three,
number:sg], neg:$Neg, fct:very_low]

//"Il fait froid" (It's cold)
|v[HEAD:température, subjC:[FORM:"il"], modC:[HEAD: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[HEAD:pleuvoir, subjC:[FORM:"il"],
objC:[HEAD:pluie, det:yes, number:sg, qual:yes,
def:no, mod:<[HEAD:fin, pos:post]>],
locCl:[HEAD:_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[HEAD:pleuvoir, subjC:[FORM:"il"],
objC:[HEAD:pluie, det:yes, number:sg,
detForm:[FORM:"un peu de"]],
locCl:[HEAD:_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[HEAD:pleuvoir, subjC:[FORM:"il"],
objC:[HEAD:pluie, det:yes, number:sg,
detForm:[FORM:"beaucoup de"]],
locCl:[HEAD:_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[HEAD:pleuvoir, subjC:[FORM:"il"],
objC:[HEAD:précipitation,
        det:yes, number:pl, qual:yes, def:no, mod:<[HEAD:fort,
        pos:pre]>], locCl:[HEAD:_pro, number:sg, person:three],
        vform:tensed, vtense:present, mode:indicative,
        subj:[person:three, number:sg], neg:$Neg, fct:a_lot];
```

2.3 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.

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

examples

— `sentence [HEAD: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`

```
   [HEAD:to_ask,
    i:[HEAD:menuisier, id:3, number:sg, gender:ms, def:yes],
    iii:[HEAD:apprenti, id:6, def:yes, gender:ms, number:sg],
    ii:[HEAD:scier,
        i:[idref:6],
        ii:[HEAD: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*
The carpenter asks the apprentice to saw the beam
The carpenter asks the apprentice to saw the beam

— `sentence`
```
  [HEAD: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 2019.* (It will rain on Monday, March 18, 2019)

— `sentence`
```
  [HEAD:cause,
   i: [HEAD:raining]]
   ii: [HEAD:se_couvrir, i:[HEAD: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`
```
[HEAD:initiative,
 i:[HEAD:Jean],
 objC:[lexFct:Magn, mod:<[HEAD: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`
```
[HEAD:initiative,
 i:[HEAD:Jean],
 objC:[lexFct:Magn],
 lexFct:Oper1
]
```
Input for

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

— `sentence`
```
[HEAD:to_break,
 i:[HEAD:Jean],
 ii:[HEAD:carafe, number:sg, def:yes,
    mod:<[HEAD:little],[HEAD:white]>],
 diathesis:passive, tense:past, neg:yes,
 modV:<[HEAD: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 Design pattern (to be completed)

A design pattern applies in a general case which can be summarized by some constraints. It is not associated with a specific event, but to a recurring organization of the grammar for which a single and coherent answer must be provided.

A - Local propagation of synthesized attributes

Context : A feature `A` that must constrain a result `R`, where `R` depends only on the lexicon or grammar.

Examples of applications :

- Production of a constrained noun in an idiomatic phrase (*breaking the **ice***)
- Conjugation to the passive voice

The general rule is written as follows

```
X → Y {
  [A:$v, $rest];
  [ARealization:R] ⊂ ⇓1;
  ↓1 = [R:$v, $rest];
}
```

Example 1 : The sentence *We basically got taken to the cleaners on that deal.* introduces the fixed noun *cleaners* which comes from the verb *take one to the cleaner*. Here the agent of the passive sentence is not represented.

```
VN → V [PP] PP {
  [obj:NIL, obl:$Obl];
  [subj:$subj] ⊂ ↑;
  ↓1 = ↑;
  ↓3 = ↑$Obl;
  [oblC:$oblSynt] ⊂ ⇓1;
  if (#2) {
    attest $oblSynt;
    ↓2 = $oblSynt;
  }
  else {
    attest ¬$oblSynt;
  }
}

taken verb[HEAD:take_one_to_the_cleaners,
        constI:[HEAD:cleaner, number:pl, def:yes]];
```

Example 2 : The agent of a clause is carried out by the grammatical function subject for conjugation to the active voice, and by the oblique function in *by* for a passive voice conjugation.

```
// agent as subject
Sentence → Sentence {
  [agent:$agent, $rest];
  [agentRealization:subject] ⊂ ⇓1;
  ↓1 = [subject:$agent, $rest];
  ⇑ = ⇓1;
}

// agent as oblique
Sentence → Sentence {
  [agent:$agent, $rest];
  [agentRealization:oblBy] ⊂ ⇓1;
  ↓1 = [byObj:$agent, $rest];
  ⇑ = ⇓1;
}

// patient as object
Sentence → Sentence {
  [patient:$patient, agent:NIL, $rest];
  [patientRealization:object] ⊂ ⇓1;
  ↓1 = [object:$patient, $rest];
  ⇑ = ⇓1;
}

// patient as subject
Sentence → Sentence {
  [patient:$patient, agent:NIL, $rest];
  [patientRealization:subject] ⊂ ⇓1;
  ↓1 = [subject:$patient, $rest];
  ⇑ = ⇓1;
}

// passive voice
Sentence → NP VERB PP {
  [subject:$subject, parObl:$parObl, $rest];
  ↓1 = $subject;
```

```
  ↓2 = $rest ∪ [voice:passive];
  ↓3 = $parObl;
  ⇑ = [agentRealization:oblBy,
        patientRealization:subject];
}

// active voice
Sentence → NP VERB NP {
  [subject:$subject, object:$object, $rest];
  ↓1 = $subject;
  ↓2 = $rest ∪ [voice:active];
  ↓3 = $object;
  ⇑ = [agentRealization:subject,
        patientRealization:object];
}
```

B - Transformation of the attribute inherited by a syntactic head

Context : An inherited attribute `A` that must constrain a realization `R` by a new predicate in generation.

Examples of applications :

- Use of a verbal tense auxiliary or a modal auxiliary
- Compound conjugation in French

The general rule is written as follows

```
X → Y {
  [A:K];
  [HEAD:$HEAD, A:k, $Rest] ⊂ ↑;
  ↓1 = [HEAD:L, arg:[HEAD:$HEAD, $Rest]];
}
```

Example : In French, the aspect is reflected in conjugated forms, but also in compound verb constructions (≪ être en train de + inf ≫ (to be in the process of doing sth).

```
sentence → sentence {
  [aspect:progressive];
  [HEAD:$HEAD, aspect:progressive, i:$I, $Rest] ⊂ ↑;
  ↓1 = [HEAD:être_en_train_de,
        i:[HEAD:$HEAD, i:$I, $Rest]];
```

```
  ⇑ = ⇓1;
}
```

4 The `elvex` command

`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.

Compilation and full installation of `Elvex`

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

- g++
- flex
- bison
- cmake
- libxml2-dev

The full command is :

```
cmake .
make
sudo make install
```

4.0.1 try `Elvex`

```
. ./try-me.sh
```

4.1 `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.

`-o|--one`

Displays only the first output.

`-rulesFile <file>`

File containing the rules.

`-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.

`-maxItems <integer>`

Defines the maximum cardinality of the item sets.

`-maxTime <integer>`

Defines the number of seconds before the program stops.

4.2 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.

Where the specific options are :

--traceInit
: Trace the initial items

--traceStage
: Trace the items sets for each generation stage

--traceClose
: Trace the items sets closure

--traceShift
: Trace the items for each lexical entry

--traceReduce
: Trace the items for each reduction

--traceAction
: Trace the operational semantics results

4.3 elvexbuildlexicon for managing big lexicons

The command of ElvexBuildLexicon is

```
elvexbuildlexicon [global-option] <build|consult|list> <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.

-patternFile <file>

The pattern file which consists of lines `lexeme pos lemma features`

-morphoFile <file>

The morphological file which consists of lines `form pos lemma features`

The lexicon is the mapping of forms where the features of pattern and morpho are unified for each couple of (pos, lemma).

Command to build a compacted lexicon

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

Command to test a compacted lexicon

```
$elvexbuildlexicon -d <directory> -f <filePrefix> list
```

Command to consult a compacted lexicon

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

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

```
pos#lexeme form#FS|form#FS...
```

— `pos` is a part of speach

— `lexeme` is the lexeme given by the `HEAD` feature

— `form` is the produced form (a string of UTF8 characters in version 2.10)

— `FS` is a feature structure