Elvex – Documentation

Lionel Clément

version 2.10 – février 2019

1 Le programme `elvex`

`Elvex` est un générateur automatique de texte. Il prend en entrée un concept, un lexique local, un lexique compacté, une grammaire, et donne en sortie un texte correspondant au concept.

Compilation et installation complète d'Elvex

La compilation complète d'`Elvex` réclame les éléments suivants sur sa machine :

- g++
- flex
- bison
- aclocal
- automake
- autoconf

La commande complète est :

```
aclocal
automake -fa
autoconf
./configure
make clean
make
sudo make install
```

1.1 Commande `elvex`

La commande d'`Elvex` est

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

Où les options sont :

`-h|--help`
: Affiche une aide et sort

`-v|--version`
: Affiche le numéro de version et sort

`-a|--reduceAll`
: Réduit toutes les règles lors du processus de génération. Ce mode permet un débogage fin et demande de comprendre l'algorithme de génération pour être maitrisé.

`-t|--trace`
: Affiche les règles réduites lors de la génération. Ce mode permet un débogage de la grammaire.

`-r|--random`
: N'affiche qu'une seule sortie choisie au hasard.

`-grammarFile <file>`
: Fichier contenant la grammaire.

`-lexiconFile <file>`
: Fichier contenant le lexique.

`-inputFile <file>`
: Fichier contenant l'input.

`-compactDirectory <directory>`
: Répertoire contenant les fichiers du lexique compacté.

`-compactLexiconFile <filePrefix>`
: Préfixe des fichiers du lexique compacté `<directory>/<filePrefix>.tbl` et `<directory>/<filePrefix>.fsa`

`-maxLength <integer>`
: Définit la longueur maximale de la séquence à produire.

`-maxUsages <integer>`
: Définit le nombre maximal d'utilisation de la même règle.

`-maxCardinal <integer>`
: Définit la cardinalité maximale des ensembles d'items.

`-maxTime <integer>`
: Définit le nombre de secondes avant l'arrêt du programme.

1.2 Commande `elvexdebug` pour débogage des grammaires

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

La commande `elvexdebug` fonctionne de la même manière avec les mêmes options mais produit un document `HTML` en sortie standard. Ce document, à lire à l'aide d'un navigateur, permet de suivre les calculs d'`Elvex`.

La commande `elvexdebug` est recommandable pour mettre au point les grammaires dans les cas les plus difficiles.

1.3 Commande `elvexbuildlexicon` pour la gestion des gros lexiques

```
elvexbuildlexicon [global-option] <build|consult|test> <input>
```

Où les options sont :

`-h|--help`
: Affiche une aide et sort

`-d <directory>`
: Répertoire contenant les fichiers du lexique compacté.

`-f <filePrefix>`
: Préfixe des fichiers du lexique compacté `<directory>/<filePrefix>.tbl` et `<directory>/<filePrefix>.fsa`.

Commande pour construire un lexique

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

Commande pour tester un lexique

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

Commande pour consulter un lexique

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

Le lexique compacté est un ensemble d'entrées lexicales de type

```
cat#lexème forme#FS
```

— `cat` est une catégorie lexicale

— `lexème` est le lexème que l'on trouve comme valeur du trait `PRED`

— `forme` est la forme produite (une chaine de caractères UTF8 dans la version 2.10)

— `FS` est la structure de traits de l'entrée lexicale

2 Grammaire

Un exemple permettra de familiariser le lecteur avec `Elvex` avant d'aller plus loin. Nous prendrons le cas de la génération d'un groupe nominal en français.

Exemple

```
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>, $Rest];
12    [number:$NInh] ⊂ ↑;
13    ↓1 = $Head ∪ [number:$NInh, gender:$GSynth];
14    ↓2 = [$Rest, mod:$Tail];
15    [gender:$GSynth] ⊂ ⇓2;
16    ⇑ = ⇓2 ∪ [qual:yes];
17 }
18
19 N → n {
20    [mod:NIL];
21    ↓1 = ↑;
22    ⇑ = ⇓1;
23 }
```

2.1 Règles syntagmatiques (*Context-Free Grammar*)

La première règle de la grammaire (ligne 5) définit `NP` comme constitué de `det` et `N` dans cet ordre.

`det` est un mot ; il n'est pas définit dans la grammaire, mais dans le lexique que nous verrons plus loin.

Les lignes 10) et 19) suivent cette même logique et nous avons la grammaire hors contexte suivante :

```
NP → det N
N → adj N
N → n
```

La seule séquence que nous pouvons générer avec ce simple exemple est un nom, précédé d'un déterminant et éventuellement d'un adjectif.

Les règles syntagmatiques sont donc faites pour décrire l'ordre des constituants des syntagmes, des phrases et des discours.

Règles vides, termes optionnels

Il est possible d'écrire des règles vides, c'est-à-dire des constituants qui n'ont aucune réalisation en production de texte. Il est aussi possible de mettre des éléments optionnels. Ils sont alors écrit entre crochets `[]`.

Exemple : La règle suivante donne toutes les combinaisons possibles pour les pronoms clitiques placés avant le verbe.

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

Le clitique datif (`cld`) est tantôt placé avant, tantôt placé après le clitique accusatif (`cla`) selon la personne. (*Jean* ***me le*** *donne*, *Jean* ***le lui*** *donne.*).

Selon cette règle, le syntagme `CLITICS` peut être vide.

Paradigmes

Chaque élément d'une règle syntagmatique peut être une liste de termes séparés par le signe `|` au lieu d'être un terme unique. Dans ce cas, le syntagme produit est construit avec l'un des termes de la liste.

Exemple : La règle suivante permet de construire une phrase sans sujet, avec un sujet clitique nominatif `cln` (*je*, *tu*, *il*, ...), ou avec un sujet sous forme d'un groupe nominal.

```
S → [cln | NP] VP
```

Une grammaire écrite en utilisant massivement les règles avec des alternatives et des termes optionnels revient à une grammaire très grosse qui pose des problèmes d'explosion combinatoire. Par exemple la règle décrivant l'ordre des pronoms clitiques est équivalente à un ensemble de 64 règles :

```
CLITICS →
CLITICS → clr
CLITICS → cld
```

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

Cependant `Elvex` ne construit jamais ces ensembles de règles et reste efficace avec de très grosses grammaires ainsi écrites. L'usage des alternatives et des termes optionnels est donc préconisé pour décrire l'ordre des mots, sans se soucier de l'explosion combinatoire évoquée plus haut.

2.2 Opérations de sémantique opérationnelle

Chaque règle contient un ensemble d'opérations en sémantique opérationnelle. Ces opérations décrivent les propriétés fonctionnelles de la langue qui dépendent ou non de l'ordre des mots.

Les bases

Nous allons expliquer cette sémantique à travers l'exemple donné au début de cette section avant de donner des détails.

— Les règles des lignes 6 et 7 expriment la nature des liens qui existent entre `NP`, le nom `N` et son déterminant `det` ; le principal étant qu'ils s'accordent en genre et en nombre.

  — La règle 6) attribue au nom les propriétés du groupe nominal. Les attributs du nom sont notés $\downarrow 2$ (c'est le second terme de la partie droite de la règle), ceux du groupe nominal $\uparrow$.

  — La règle 7) récupère les informations issues des analyses de `N` pour les attribuer au déterminant. Les attributs du déterminant sont notés $\downarrow 1$ (c'est le premier terme de la partie droite de la règle), ceux issues du nom `N` $\Downarrow 2$.

  Avec ces deux règles, le nom sera choisi dans le lexique en fonction du concept à exprimer (attribut du groupe nominal $\uparrow$). Un déterminant singulier féminin sera choisi si le nom est de genre féminin et que le groupe nominal est singulier.

  Ce mécanisme, qui permet d'articuler les choix lexicaux en fonction des concepts et les propriétés syntaxiques en fonction des propriétés lexicales, est au coeur d'`Elvex` et met en œuvre deux types d'attributs : les attributs hérité (notés $\uparrow$, $\downarrow_i$) et les attributs synthétisé (notés $\Uparrow$, $\Downarrow_i$).

Les attributs hérités sont calculés de gauche à droite des règles syntagmatiques, les attributs synthétisé sont calculés de droite à gauche. Pour faire simple, nous pouvons dire que les attributs hérités véhiculent des informations données depuis l'*input* vers le lexique, et que les attributs synthétisés véhiculent des informations venant du lexique vers des couches plus hautes de l'analyse. Ainsi la règle 6) permet de propager l'*input* vers le lexique pour réaliser un choix lexical, alors qu'à l'inverse, la règle 7) permet de récupérer les informations lexicales du nom choisi en fonction du concept exprimé et de propager ces informations (le genre, les restrictions de sélection, les contraintes d'une expression phraséologique, etc) vers le groupe nominal entier.

— Les règles des lignes 11 à 16 permettent de régler l'accord en genre et nombre du nom et de l'adjectif et aussi de propager les concepts qui produisent les adjectifs. On remarque qu'en français, le nombre est un sémantème exprimé directement sous la forme d'un signe associé au substantif, alors que le genre est une marque morphologique du lexique. Le premier sera propagé depuis le concept d'un nom, le second sera propagé depuis le lexique vers les couches réglant l'accord.

  — La règle 11) est une *garde*, c'est-à-dire une condition de réalisation de la production `N` → `adj N`. Elle exprime le fait que la réalisation d'un adjectif est conditionnée par l'existence d'une liste `<$Head::$Tail>` comme valeur d'un attribut `mod` dans l'attribut hérité ↑.

    `$Head` et `$Tail` sont deux variables qui définissent respectivement la tête et la queue d'une liste, c'est-à-dire respectivement le premier élément de la liste et le reste de cette liste.

    Si la condition de cette garde est réalisée, alors `$Head`, `$Tail` et `$Rest` seront affectées par des valeurs données par la subsomption [1] de `[mod:<$Head::$Tail>, $Rest]` par ↑.

  — La règle 12) affecte la variable `$NInh` (*Inherited noun*) avec la valeur du trait `number` de l'attribut hérité ↑. Ceci permet de récupérer le nombre donné au groupe nominal.

  — La règle 13) affecte à l'attribut hérité de l'adjectif la tête de la liste `$Head` unifiée [2] avec une structure qui donne le genre synthétisé `$GSynth` (*synthesized gender*) et le nombre hérité `$NInh`.

1. Une structure de traits `A` subsume une structure de traits `B` si tous les traits de `B` sont présents dans `A` avec des valeurs équivalentes ou avec des valeurs qui subsument celles des valeurs de `A`.

2. L'unification d'une structure de traits `A` avec une structure de traits `B`, est la plus

- La règle 14) affecte à l'attribut hérité du nom l'ensemble des valeurs passées par héritage (`$Rest`) et ajoute le trait `mod` avec comme valeur `$Tail`, c'est-à-dire la queue de la liste qui rassemble l'ensemble des autres modifieurs du nom (autre adjectif ou une proposition relative par exemple).
- La règle 15) permet d'affecter la variable `$GSynth` qui est le genre synthétisé à partir du nom.
- Enfin la règle 16) transmet comme attribut synthétisé l'attribut synthétisé du nom et ajoute l'information que cet attribut est qualifié (ce qui permet de restreindre par exemple d'autres modifications avec un adjectif léger).

— Les règles des lignes 20 à 22 ne présentent rien de particulier, à part la garde de la règle 20) qui met en œuvre la valeur `NIL`.

- La règle 20) est une garde telle que l'attribut hérité ↑ ne contient aucun trait `mod`. `mod:NIL` signifie que l'attribut `mod` n'existe pas dans ↑.
- Les règles 21) et 22) ne demandent aucun commentaire supplémentaire.

Il devient clair en examinant cet exemple simple que l'ordre des règles opérationnelles n'est pas celui donné par la grammaire, mais celui dicté par les disponibilités des opérandes. Ici, un ordre possible d'application des règles est 11), 12), 14), 15), 13) et 16). Dans la version 2.10 d'`Elvex` cet ordre est calculé à la volée. Dans les versions ultérieures un graphe de dépendance sera construit avec des indications permettant de mettre au point les grammaires sur ce point précis.

Le conseil que je peux donner pour mettre au point une grammaire est d'écrire les règles au fur et à mesure que les attributs synthétisés sont disponibles.

On voit que les règles syntagmatiques ne sont pas produites linéairement de gauche à droite et de haut en bas, mais de façon a priori indéterminée en produisant les éléments disponibles en premier.

Avant d'aller plus loin, donnons des définitions sur les notations qui nous permettent de comprendre la syntaxe des règles.

---

petite structure de traits `C` telle que `A` subsume `C` et `B` subsume `C`. Pour un éclairsissement de ces notions de subsomption et d'unification, voir par exemple l'introduction du livre *Les grammaires d'unification*, Anne Abeillé, Lavoisier 2007, Coll. langues et syntaxe, Paris

Syntaxe de la grammaire

Une règle syntagmatique s'écrit
$A \rightarrow B_1 B_2 \ldots B_K < operational\ semantics >$;
Où B_i s'écrit :

— $[B_i]$ si le terme B_i est optionnel.

— $C_1|C_2|\ldots|C_n$ si le terme B_i est l'alternative entre les C_i

B_i, C_i peuvent être des catégories lexicales ou des termes de la grammaire.

Sémantique opérationnelle

Notations Soit la règle $A \rightarrow B_1 B_2 \ldots B_i \ldots B_k < operational\ semantics >$

— $\uparrow$: désigne l'attribut hérité de A.

— $\downarrow_i$: désigne l'attribut hérité de Bi.

— $\Uparrow$: désigne l'attribut synthétisé de A.

— $\Downarrow_i$: désigne l'attribut synthétisé de Bi.

Les variables nommées sont notées par un identifiant précédé de $\ll \$ \gg$, les variables libres sont notées $\ll \_ \gg$, les valeurs constantes sont notées par des identifiants.

Les structures de traits sont notées $[trait_1, trait_2, \ldots, trait_k]$ où $trait_i$ est

— soit une variable libre

— soit une variable nommée

— soit `FORM:"<form>"` un trait pour une forme littérale

— soit $attr_i : val_i$

  — avec $attr_i$ le nom d'un attribut ou `PRED` ou `FORM`

  — et avec val_i la valeur d'un attribut qui peut être :

    — Une variable nommée

    — Une variable libre

    — Une valeur nulle (`NIL`)

    — Un ensemble de constantes atomiques séparées par `|`

    — Une constante littérale (nombre)

    — Une structure de traits

— Une liste

Les listes sont notées

— $< l_1, l_2, \ldots, l_k >$ avec l_i structure de traits

— $< head :: tail >$ avec *head* structure de traits et *tail*, liste ou variable nommée ou non
head désigne le premier élément de la liste, *tail* le reste de la liste.

Si l'on souhaite apporter une sémantique opérationnelle à la règle syntagmatique, on définit $<$ *operational semantics* $>$ qui est une liste de règles entre accolades.

Chaque règle s'écrit :

— {<liste de règles >} Une liste de règles.

— `attest <expr>;` Garde explicite où `<expr>` est une expression booléenne qui doit être vérifiée.

— `[attr, attr, ..., attr];` Garde où la structure de traits doit subsumer $\uparrow$. La garde peut contenir le trait `PRED` qui représente le concept à générer. Ainsi, un concept n'est pas nécessairement lié à une entrée lexicale avec `Elvex`, contrairement aux systèmes de génération de texte habituels ; il peut produire une contrainte syntaxique.

— `print <expr>;` Affiche l'expression en sortie standard (utile pour le débogage).

— `println <expr>;` Fait la même chose en affichant une nouvelle ligne.

— `<expr1> = <expr2>;`
Une affectation de l'expression $< expr1 >$ avec la valeur de l'expression $< expr2 >$. L'expression $< expr1 >$ est une variable complexe ou simple, l'expression $< expr2 >$ est une constante ou une variable complexe ou simple telle qu'il existe un environnement [3] de la règle syntagmatique pour le résoudre.
Les affectations de $\uparrow$ et $\Downarrow_i$ ne sont pas possibles, car ces attributs sont implicitement évalués lors de la génération de texte et leur modification provoquerait des incohérences.
Les affectations se résument à ces cas :

3. Un environnement est une application ϕ de E dans F telle que E est un ensemble de variables et F un ensemble de constantes ou de variables. La résolution d'une variable α dans par un environnement est $\beta = \phi^k(\alpha)$ tel que β est une constante. Notons qu'`Elvex` manipule aussi les variables complexes. Dans ce cas $\phi^k(\alpha)$ revient à remplacer les variables dans les termes par leurs constantes.

— Affecter une liste
  — `<...> = <...>` Affecte une liste à une autre.
  — `<...> = $X` Affecte à une liste la valeur de la variable `$X`
— Affecter un attribut hérité
  — ↓`i = [...]` Affecte une structure de traits constante ou variable.
  — ↓`i =` ↑ Affecte un attribut hérité.
  — ↓`i =` ... ∪ ... Affecte le résultat de l'unification de deux expressions (structures de traits ou attributs).
  — ↓`i =` ⇓`j` Affecte un attribut synthétisé.
— Affecter un attribut synthétisé
  — ⇑ `= $X`
  — ⇑ `= [...]`
  — ⇑ `=` ↑
  — ⇑ `=` ... ∪ ...
  — ⇑ `=` ⇓`j`
— Affecter une variable simple
  — `$X = $Y`
  — `$X = a` Affecte par une constante ou un littéral.
  — `$X = <...>`
  — `$X = [...]`
  — `$X =` ↑
  — `$X =` ... ∪ ...
  — `$X =` ⇓`j`
  — `$X = <expr>` par l'évaluation d'une expression arithmétique ou logique.
— `[...]` ⊂ `<expr>;`
  Subsume une structure de traits
  — `[...]` ⊂ ↑ par un attribut hérité
  — `[...]` ⊂ ⇓`j` par un attribut synthétisé
  — `[...]` ⊂ `$X` par la valeur d'une variable

— `if (<boolExpr>) <rules>`
  Interprète la sémantique opérationnelle `<rules>` si et seulement si l'expression `<boolExpr>` est vérifiée.

— `if (<boolExpr>) <rules1> else <rules2>`
  Interprète la sémantique opérationnelle `<rules1>` si et seulement si `<boolExpr>` est vérifiée, sinon interprète `<rules2>`.

Une expression logique `<booExpr>` est

— `<boolExpr>` $\vee$ `<boolExpr>` Ou logique

— `<boolExpr>` $\wedge$ `<boolExpr>` Et logique

— $\neg$ `<boolExpr>` Négation logique

— `<boolExpr>` $\Rightarrow$ `<boolExpr>` Implication logique

— `<boolExpr>` $\Leftrightarrow$ `<boolExpr>` Équivalence logique

— `<expr>` == `<expr>` Égalité entre deux expressions

— `<expr>` $\neq$ `<expr>` Différence entre deux expressions

— `<expr>` Expression évaluée comme booléen : Une structure de traits est évaluée comme vraie si elle n'est pas `NIL` ou $\perp$ (résultat de l'échec de l'unification entre deux structures de traits).

Les expressions sont :

— Entier

— Chaine de caractères

— `<expr>` $\cup$ `<expr>` Unification de deux structures de traits

— $\uparrow$

— $\Uparrow$

— $\downarrow_i$

— $\Downarrow_i$

— `[...]` Structure de traits

— `NIL` Valeur nulle d'un trait

— `a|b...` Constante

— `$xxx` Variable

— `<...>` Liste

Ajoutons les expressions algébriques et les nombres :

— `<expr>` $<$ `<expr>`
— `<expr>` $\leq$ `<expr>`
— `<expr>` $>$ `<expr>`
— `<expr>` $\geq$ `<expr>`
— `<expr>` $+$ `<expr>`
— `<expr>` $-$ `<expr>`
— `<expr>` $*$ `<expr>`
— `<expr>` % `<expr>`
— `<expr>` / `<expr>`
— $-$ `<expr>`
— Double flottant

Évaluation des variables

Les variables sont évaluées localement à une règle syntagmatique. Cela définit la portée d'une variable.

Voici différentes méthodes permettant de résoudre les variables :

— Affectation directe

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

La ligne 5 indique que la valeur de la variable `$tense` est `present` à défaut d'une autre valeur donnée préalablement. La valeur de `$tense` est éventuellement donnée par l'attribut hérité ↑ en ligne 3.

Noter que l'ordre d'évaluation des règles est indifférent et qu'il n'est pas nécessaire de faire précéder la règle 3) dans l'ordre d'écriture des règles.

— Garde

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

La valeur de la variable `$subjectInherited` est donnée par le trait `subject` de ↑. La valeur de la variable `$Rest` est la structure de traits qui reste quand on retire le trait `subject`.

— Subsomption

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

La valeur de la variable `$subjectSynthesized` est donnée par le trait `subject` de ⇓ 2.

— Liage d'une variable au sein d'une règle syntagmatique

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

Les variables `$subjectSynthesized` et `$subjectInherited` sont toutes deux exploitées au sein de cette règle pour être transmises au trait `subject` de l'attribut hérité ↓ 1.

— Liage d'une variable au sein d'une entrée lexicale

```
a       v[PRED:fog, 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]
```

La variable `$Neg` est liée dans cette entrée lexicale de l'expression "il y a du brouillard" de sorte que son emploi au négatif propage un négatif du substantif :

(1) a. *Il y a du brouillard*
b. *Il n'y a pas de brouillard*
c. **Il y a de brouillard*
d. **Il n'y a pas du brouillard*

2.3 Lexique

Le lexique est une liste d'entrées séparées par des point-virgules.

Chaque entrée définit la séquence à produire, suivie d'une disjonction (notée par le caractère `|`) de catégories lexicales optionnellement suivies d'une structure de traits.

La séquence à produire peut être :

— Une chaine de caractères entre `""`, auquel cas la séquence produite sera exactement celle-ci. À noter que la séquence peut être vide.

— Une chaine de caractères entre `""` contenant des variables. La séquence produite sera celle-ci avec les variables substituées pas leur valeurs respectives.

— `FORM` : la séquence produite sera littéralement celle donnée par le trait `FORM`. La version 2.10 ne prévoit que des `FORM` constantes, les versions ultérieures auront aussi des variables complexes.

La catégorie est un identifiant, symbole terminal de la grammaire.

La structure de traits permet de représenter l'ensemble des contraintes imposées par le lexique.

Quelques exemples :

— `"?" 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]];
```

```
— // Entrée lexicale "temperature" produisant
  "la température est de xxx degrés" avec [cel:xxx]
  est v[PRED:temperature,
  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"
  |v[PRED:temperature, 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"
  |v[PRED:temperature, 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];

— // Entrée lexicale de "temperature"
  // produisant "Il fait chaud"
  fait v[PRED:temperature, 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"
  |v[PRED:temperature, 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"
  |v[PRED:temperature, 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"
|v[PRED:temperature, 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];

// entrée lexicale de "pleuvoir" produisant
// "il y a une pluie fine"
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"
|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"
|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"
|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

L'input se résume au nom du syntagme que l'on souhaite générer, suivi d'une structure de traits.

`PRED` permet de définir le concept à générer. Concept correspondant à un lexème ou à une construction syntaxique particulière.

exemples

— `sentence [PRED:pleuvoir, neg:yes]`

Simple entrée pour le concept de *pleuvoir* produisant

(2) a. *Il ne pleut pas*
b. *Il n'y a pas de pluie*

—
```
sentence [PRED:demander,
    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]
 ]
    ]
```

Concept complexe où les traits `id` et `idref` permettent de contraindre les coréférences. Cette entrée produit

(3) *Le menuisier demande à l'apprenti de scier la poutre*

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

Concept produisant une adverbiale sur la base de la constante "lundi 18 mars 2019". Cette entrée produit

(4) *Il va pleuvoir le lundi 18 mars 2013.*

—
```
sentence [PRED:cause,
    i: [PRED:pleuvoir]]
    ii: [PRED:se_couvrir, i:[PRED:you], modality:devoir]
]
```

Entrée pour

(5) a. *Il pleut, tu dois te couvrir*
b. *Il pleut, par conséquent tu dois te couvrir*

c. *Tu dois te couvrir s'il pleut*

d. *Tu dois te couvrir car il pleut*

e. *S'il pleuvait, tu devrais te couvrir*

f. *S'il pleut, tu devras te couvrir*

g. *Il pleuvrait, tu devrais te couvrir*

— `sentence [PRED:initiative, i:[PRED:Jean], objC:[lexFct:Magn, mod:<[PRED:audacieux]>], lexFct:Oper1]`
Input contenant des traits pour fixer des fonctions lexicales. La production est

(6) *Jean prend une initiative très audacieuse.*

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

(7) *Jean prend une belle initiative.*

— `sentence [PRED:casser, i:[PRED:Jean], ii:[PRED:carafe, number:sg, def:yes, mod:<[PRED:petit],[PRED:blanc]>], diathesis:passive, tense:past, neg:yes, modV:<[PRED:brutalement]>]`
Input contenant un trait passif négatif. La production est

(8) *La petite carafe blanche n'avait pas été cassée brutalement par Jean.*

3 Écriture des grammaires

3.1 Schémas de conception

Un shéma de conception s'applique dans un cas général qui peut être résumé par quelques contraintes. Il n'est pas associé à un phénomène linguistique en particulier, mais à une organisation récurrente de la grammaire pour laquelle une réponse unique et cohérente doit être apportée.

A - Propagation locale des attributs synthétisés

Un trait `A` qui doit contraindre une réalisation `R` en génération telle de `R` ne dépende que du lexique ou de la grammaire.

Exemples d'applications :

— Production d'un groupe nominal contraint dans une expression phraséologique

— Conjugaison à la voix passive

La règle générale s'écrit

```
1 X → Y {
2    [A:$v, $rest];
3    [ARealization:R] ⊂ ⇓1;
4    ↓1 = [R:$v, $rest];
5 }
```

Exemple 1 : La production des compléments contraints dans une expression phraséologique. L'expression *casser du sucre sur le dos de quelqu'un* introduit la réalisation de deux compléments non actualisés *du sucre* et *sur le dos de qqu'un*, et le second complément (*sur le dos*) contient un génitif qui est actualisé dans l'énoncé par le patient de la clause.

(9) *Ce rapport casse du sucre sur le dos* ***de la commission****.*

(10) *La commission, c'est toujours sur* ***son*** *dos qu'on casse du sucre.*

(11) *J'ai failli* ***te*** *casser du sucre sur le dos.*

Il faut donc rendre compte de la réalisation des compléments constants de l'expression et du patient sous la forme d'un génétif dans une telle expression.

```
1  VN → V [NP] [PP] {
2     [obj:NIL, obl:NIL];
3     [subj:$subj] ⊂ ↑;
4     ↓1 = ↑;
5     [objC:$objSynt, oblC:$oblSynt] ⊂ ⇓1;
6     if (#2) {
7        attest $objSynt;
8        ↓2 = $objSynt;
9     }
10    else {
11       attest ¬$objSynt;
12    }
13    if (#3) {
```

```
14      attest $oblSynt;
15      ↓3 = $oblSynt;
16    }
17    else {
18      attest ¬$oblSynt;
19    }
20  }
21
22  casser verb[PRED:casser_du_sucre,
23           constI:[PRED:sucre, part:yes],
24           constII:[PRED:sur, i:[PRED:dos,
25                    number:sg, def:yes,
26                    genitive:$patient]],
27           patient:$patient];
```

Exemple 2 : L'agent d'une clause se réalise par la fonction grammaticale sujet pour une conjugaison à la voix active, et par la fonction oblique en « par » pour une conjugaison à la voix passive.

```
1  // Réalisation d'un agent comme sujet
2  Sentence → Sentence {
3    [agent:$agent, $rest];
4    [agentRealization:subject] ⊂ ⇓1;
5    ↓1 = [subject:$agent, $rest];
6    ⇑ = ⇓1;
7  }
8
9  // Réalisation d'un agent comme objet prépositionnel
10 Sentence → Sentence {
11   [agent:$agent, $rest];
12   [agentRealization:oblBy] ⊂ ⇓1;
13   ↓1 = [byObj:$agent, $rest];
14   ⇑ = ⇓1;
15 }
16
17 // Réalisation d'un patient comme objet
18 Sentence → Sentence {
19   [patient:$patient, agent:NIL, $rest];
20   [patientRealization:object] ⊂ ⇓1;
21   ↓1 = [object:$patient, $rest];
22   ⇑ = ⇓1;
```

```
23 }
24
25 // Réalisation d'un patient comme sujet
26 Sentence → Sentence {
27   [patient:$patient, agent:NIL, $rest];
28   [patientRealization:subject] ⊂ ⇓1;
29   ↓1 = [subject:$patient, $rest];
30   ⇑ = ⇓1;
31 }
32
33 // Construction Passive
34 Sentence → NP VERB PP {
35   [subject:$subject, parObl:$parObl, $rest];
36   ↓1 = $subject;
37   ↓2 = $rest ∪ [voice:passive];
38   ↓3 = $parObl;
39   ⇑ = [agentRealization:oblBy,
40         patientRealization:subject];
41 }
42
43 // Construction Active
44 Sentence → NP VERB NP {
45   [subject:$subject, object:$object, $rest];
46   ↓1 = $subject;
47   ↓2 = $rest ∪ [voice:active];
48   ↓3 = $object;
49   ⇑ = [agentRealization:subject,
50         patientRealization:object];
51 }
```

B - Transformation de l'attribut hérité par une tête syntaxique

Un trait hérité `A` qui doit contraindre une réalisation `R` par un nouveau prédicat en génération.

Exemples d'applications :

— Utilisation d'un auxiliaire de temps verbal ou de modalité

— Conjugaison composée

La règle générale s'écrit

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

Exemple : L'aspect se traduit par des formes conjuguées, mais aussi par des constructions composées (≪ être en train de + inf ≫, ≪ aller + participe présent ≫, etc.). La catégorie grammaticale `aspect`

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