Rapport LO17
Analyse morphologique et syntaxique

ASCHENBRENNER Céline MARECHAL Anaig

26 septembre 2016

Table des matières

Résumé

Après avoir procédé à l'indexation de notre corpus, l'étape suivante est d'associer les termes de ce corpus à des requêtes de recherche. Pour cela nous avons tout d'abord effectué une analyse morphologique. Il s'agit d'un programme en Java qui utilise différents algorithmes pour associer chaque mot d'une requête à un ou plusieurs termes pertinents issus de notre liste de termes contenus dans le corpus de recherche. Cela nous permet d'obtenir le ou les lemmes les plus probables de chaque mot de la requête. Cet analyseur doit pouvoir prendre en compte les éventuelles fautes de frappe et fautes d'orthographe de l'utilisateur.
Ensuite nous avons effectué une analyse syntaxique. Nous sommes parties d'un recueil de requêtes possibles et nous avons analysé leur forme. Il s'agissait ici d'en extraire leur syntaxe pour pouvoir réaliser la grammaire adéquate avec AntlrWorks.
Une fois la grammaire construite et les requêtes optimisée après un post-traitement, nous avons pu obtenir des résultats en requêtant la base de données.
Une fois les trois étapes analyse morphologique, analyse syntaxique et SQL effectuées et les projets liés, on obtient des résultats de la base de données directement après une requête en langage naturelle, en console.
A chaque fois nous avons cherché à optimiser notre travail, via différentes méthodes et solutions que nous présentons dans ce rapport.

1 Analyse morphologique

1.1 Le projet Java

Pour commencer nous avons constitué notre propre fichier de lemmes pour tester les différents algorithmes. Pour que cela soit pertinent, il fallait y insérer des mots assez proches pour pouvoir analyser véritablement si l'algorithme prenait le plus pertinent.
Notre programme est constitué de plusieurs classes Java. La première, *Lexique.java*, fait le lien entre le lexique et un mot reçu en argument. Elle a donc différentes méthodes permettant de parcourir le lexique selon différents algorithmes. Elle a comme attribut une hashMap appelée lexique, qui contient les mots et les lemmes de notre fichier de lemmes. Cette map est initialisée et remplie à partir d'un dictionnaire donné dans le constructeur grâce à une méthode inspirée de celle donnée en TD.

Nous avons décidé de programmer étape par étape, nous avons donc testé chaque algorithme séparément avant de les réunir en un seul processus.
La classe *Requete.java* a pour objet de demander à l'utilisateur d'entrer une requête. Ensuite, pour chacun des termes de la requête elle appelle la méthode *getLemme()* de *Lexique.java*, qui renvoie le ou les meilleurs lemmes candidats pour le terme entré en paramètres.
getLemme() de Lexique.java va effectuer sur le terme les différentes informations demandées :

- Elle le met en lettres minuscules
- Elle teste si il existe dans le lexique, et si oui retourne son lemme.
- Dans le cas contraire, elle appelle une autre méthode qui cherche des lemmes candidats grâce à l'algorithme des préfixes. Si celui-ci en trouve elle les retourne.
- Sinon, elle appelle une dernière méthode qui utilise le test de Levenshtein pour trouver des lemmes candidats. Si celui-ci en trouve, elle les retourne.
- Enfin, si aucun lemme candidat n'a pu être identifié, elle l'explicite par un message.

Algorithm 1 getLemme()

```
mot ← toLower(mot)
if mot ∈ lexique then
  return lexique.getlemme
else if getLemmeByPrefixe() != null then
  return getLemmeByPrefixe()
else if getLemmeByLevenshtein() != null then
  return getLemmeByLevenshtein()
else
  print "Aucun lemme candidat trouvé"
end if
```

Pour chaque algorithme, on va créer une liste qui contiendra les lemmes au poids le plus fort, c'est-à-dire les lemmes les plus pertinents. Pour cela, à chaque mot du dictionnaire comparé avec le mot recherché, on ajoute dans une liste le premier mot, on enregistre son poids puis on compare son poids avec le mot suivant. Si le nouveau mot a un poids plus important, on vide la liste, on ajoute ce mot et on enregistre son poids. Si le nouveau mot a un poids égal, on l'ajoute à la liste.

1.2 Algorithme de préfixe

L'algorithme préfixe va comparer le terme entré en paramètre avec chacun des termes du lexique. La première étape consiste à étudier la taille des mots. En effet, l'algorithme utilise des valeurs *seuilMin* et *seuilMax*, que nous avons choisi de façon arbitraire pour commencer.
Si le mot de la requête est trop petit (inférieur à *seuilMin* que nous avons fixé à 3) alors l'algorithme par préfixe ne pourra pas lui trouver de lemme candidat, la fonction s'arrête là.
Sinon pour chaque mot du lexique, on va regarder si il est suffisamment grand (supérieur à *seuilMin*) et si sa longueur n'est pas trop différente de celle du mot recherché (la longueur doit être inférieure à *seuilMax*). Si une de ces conditions n'est pas respectée, la proximité entre les deux termes est fixée à 0. Sinon il s'agit d'un pourcentage qui correspond au nombre de lettres communes situées côte à côte à partir du début, divisé par le nombre total de lettres du mot le plus grand. La proximité est donc d'autant plus grande que les mots ont un début commun, c'est à dire un préfixe identique.
On récupère ensuite la liste des lemmes des mots qui ont la plus grande proximité avec le terme entré en paramètre.

Algorithm 2 Préfixes

```
while le lexique contient des mots candidats do
  if (|mot| < seuilMin OR |motCandidat| < seuilMin) then
    proximite ← 0
  else if ratio(nb lettres communes) > seuilMax then
    proximite ← 0
  else
    i ← 0
    while i < min(|mot|, |motCandidat|) AND mot[i] == motCandidat[i] do
      i ← i + 1
      proximite ← 100 × i / max(|mot|, |motCandidat|)
      if proximite > proximiteMax then
        liste ← proximite
        proximiteMax ← proximite
      else if proximite == proximiteMax then
        liste ← liste + proximite
      end if
    end while
  end if
end while
```

Longueur L	$L \leq 4$	$4 < L \leq 8$	$8 < L \leq 12$	$12 < L$
$seuilMax$	2	0.5L	0.4L	0.3L

Lors de la rédaction de cet algorithme la seule difficulté technique que nous avons rencontré était la formule de proximité qui ne donnait pas de résultats cohérents. Cela était du à un problème d'arrondis, que nous avons pu fixer rapidement à l'aide d'un float. Nous avons ensuite choisi d'optimiser cet algorithme en ne fixant plus arbitrairement le *seuilMax*. En effet, puisqu'il s'agit d'un seuil de différence de longueur entre deux mots, il nous semble cohérent qu'il soit fixé en fonction de la longueur du mot entré en paramètres.

1.3 Algorithme de Levenshtein

Si aucun lemme candidat n'est retenu par l'algorithme des préfixes, l'algorithme utilisé est alors *Levenshtein*.
Au début de la méthode, on initialise une matrice faisant la taille des deux mots à comparer, plus un caractère (qui correspondra à la cellule de départ de la comparaison, initialisée à 0).
On remplit ensuite la matrice : la première ligne et la première colonne ne représenteront forcément que des suppressions et des insertions. On incrémente donc de 1 la première colonne en descendant chaque cellule et la première ligne en parcourant les cellules vers la droite. Pour le reste de la matrice, grâce à une double boucle, on ajoute 1 en cas de suppression ou insertion, donc en allant vers la droite ou le bas. Pour la substitution, représentée en diagonale, si la lettre comparée est la même pour les deux mots, le poids ajouté est de 0, 1 sinon.
Après un test sur quelques mots, notre méthode semble fonctionner en nous renvoyant des résultats corrects. Cependant, on pourrait lui reprocher de nous renvoyer des listes de lemmes candidats, donc au même poids, mais dont l'on voit pourtant qu'un mot en particulier aurait pu se démarquer.
C'est le cas ici lorsque nous tapons "ilste". L'algorithme nous propose les deux mots du dictionnaires "liste" et "piste" comme candidats à part égal. Cependant il paraîtrait normal que "liste" soit désigné comme étant le plus proche car cela pourrait correspondre à une faute de frappe en inversant les caractères “i” et “l”. La solution serait donc de diminuer ici le coût de "liste" en faisant en sorte que l'algorithme repère qu'il y a inversion de lettres.
Pour cela, pour chaque caractère du mot comparé, nous avons regardé le caractère de l'autre mot, directement précédent ou suivant. Si les deux caractères sont égaux, on attribue alors un poids de 0 à la substitution. De cette manière, c'est bien le mot "liste" qui est ici le seul proposé quand un utilisateur tape "ilste".
Une autre résolution à ce problème aurait été de vérifier si les deux mots comparés sont des anagrammes. On aurait alors effectué une permutation circulaire sur les mots pour le vérifier. Cependant, nous n'avons pas choisi d'implémenter cette méthode car nous pensons que les résultats pourraient devenir aberrant si l'on ne prend pas assez compte de la place des lettres, et de plus le coût de l'algorithme est plus lourd. Voici la façon dont nous procéderions :

- On récupère toutes les lettres de chaque mot dans deux listes ;
- On trie ces listes dans l'ordre alphabétique (complexité : $O(n^2)$ avec un tri par insertion) ;
- On effectue une comparaison des deux listes et on pénalise de 0.1 point par lettre qui diffère. ;

L'anagramme parfait reste donc à 0. (complexité : O(2n))

Nous avons ensuite effectué un nouveau test en recherchant le mot "leste". Les mots candidats retournés sont "liste", "peste" et "reste". Or, si l'on tient compte de la proximité des lettres sur un clavier, un des mots auraient pu se distinguer. Nous avons donc créé une nouvelle classe *LetterIndex*, qui, lors de l'appel de son constructeur, initialise une matrice de taille 26 sur 26 qui, pour chaque combinaison de 2 lettres parmi les 26, indique 1 si elles sont côte à côte sur un clavier azerty, 0 sinon. La méthode *isNext(char x, char y)* va donc prendre en argument deux caractères, et grâce à leur position dans la table ASCII qui détermine leur index dans la matrice de proximité va renvoyer une valeur booléenne, true si les lettres sont proches sur un clavier, false sinon. Lors de l'algorithme de Levenshtein, si la méthode *isNext* renvoie vrai pour deux caractères, le poids de la substitution est allégé à seulement 0.1 (nous avons choisi un poids très faible mais qui se démarque tout de même de 2 lettres égales ou inversées). Ainsi, en effectuant à nouveau un test sur le mot leste, le lemme renvoyé est désormais "peste".
En outre, nous avons pensé que par faute de temps, l'habitude d'écrire sur un clavier étranger ou à cause d'une faute d'orthographe par exemple, certaines personnes ne prenaient pas la peine d'écrire les accents sur certains mots. Ainsi, nous avons ajouté au fichier texte les mots "cinéma" et "cinima". Quand l'utilisateur tape "cinema", les deux mots sont candidats. Or il nous semblerait plus pertinent

de renvoyer le mot "cinéma". Nous avons donc ajouté à la méthode *isNext* une vérification qui appelle une méthode *checkAccent*. Dans cette méthode, on vérifie pour chaque lettre les lettres accentuées qui lui sont associées (par exemple pour la lettre e : è, é, ê et ë) et on renvoie true si les caractères comparés peuvent être associés. On diminue alors un peu le coût de la substitution et, de cette manière, le mot candidat de "cinema" est seulement "cinéma".

Algorithm 3 Levenshtein

```
dist ← double[|mot| +1][|motCandidat| + 1]
dist[0][0] ← 0
for (i=1, i<|mot|, i++) do
  dist[i][0] ← dist[i-1][0] + 1
end for
for (i=1, i<|motCandidat|, i++) do
  dist[0][j] ← dist[0][j-1] + 1
end for
for (i=0, i<|mot|, i++) do
  for (j=0, j<|motCandidat|, j++) do
    substitution ← 1
    if (mot[i]==motCandidat[j] OR j < |motCandidat| - 1 AND mot[i] == motCandidat[j+1] AND
    i < |mot| - 1 AND mot[i+1] == motCandidat[j]) then
      substitution ← 0
    else if (procheSurLeClavier(mot[i] , motCandidat[j])) then
      substitution ← 0.1
    end if
    d1 ← dist[i][j] + substitution
    d2 ← dist[i][j+1] + 1
    d3 ← dist[i+1][j] + 1
    dist[i+1][j+1] ← min(d1, d2, d3)
  end for
end for
```

Choix du seuil Comme pour l'algorithme de lemmatisation par préfixes, on fait ensuite une liste des meilleurs candidats retournés par l'algorithme de Levenshtein. Les meilleurs candidats sont ici sont avec la plus faible proximité : on va donc sélectionner ceux avec une proximité inférieure à une distance maximale.
Nous avons tout d'abord choisi une distance seuil de 5, qui nous fourni de bons résultats. En revanche, nous avons remarqué après quelques tests que les mots longs comme "environnement" étaient moins bien lemmatisés que des mots plus courts comme "chimie". Comme pour l'algorithme des préfixes, nous avons donc décidé de moduler la valeur seuil en fonction de la taille du mot, ici 0.80 $\times$ la longueur du mot. Cette fois ci un peu trop flexible pour les mots longs, nous avons modéré l'influence du facteur longueur du mot en le remplaçant par longueur du mot - 1.

En résumé, notre projet Java Correcteur est tout d'abord lancé avec la classe *Requete* qui contient la méthode main. Cette méthode, permet à l'utilisateur de saisir une requête en entrée. Grâce à l'objet StringTokeniser, les mots de cette requête sont lus un à un et sont envoyés à la classe *Lexique* qui se charge de trouver le lemme correspondant au mot. Si le mot appartient au dictionnaire il renvoie son lemme, autrement le mot passe par l'algorithme de préfixe, puis de Levenshtein qui renvoient les meilleurs candidats.

2 Analyse Syntaxique

2.1 Grammaire Antlr

2.1.1 Introduction

Pour la suite du projet, nous avons travaillé sur la plateforme AntlrWorks. Antlr signifie ANother Tool for Language Recognition. C'est un compilateur permettant de construire des grammaires sous forme d'arbres à partir d'un framework. L'analyseur peut ensuite être généré en Java (dans notre cas, mais c'est également possible en C par exemple) et permet d'analyser un texte afin de reconnaître sa grammaire.
Le principe de notre travail sur cette plateforme consiste à recevoir une requête normalisée, dont les mots sont externes à la requête SQL, sous la forme par exemple vouloir article contenir <x> et <y>. On veut ensuite transformer cette requête en requête SQL : ainsi, pour notre exemple, on aurait SELECT article FROM titretext WHERE mot=<x> AND mot=<y>.

2.1.2 Prise en main

La première étape a été la prise en main de la plateforme. Pour cela, nous avons utilisé une grammaire basique qui nous a été fournie dans un fichier .g, formant une phrase du type groupe nominal suivi d'un groupe verbal.
Nous avons tout d'abord parcouru globalement le fichier afin de comprendre comment manipuler le framework. On observe en premier lieu une liste de règles lexicales. Elles vont être utilisées par **l'analyseur lexical**, aussi nommé lexer, qui va découper un flux de caractères en mots, aussi appelés tokens. Par exemple VERBE : *'mange' | 'mangent'* ; signifie qu'un verbe peut être représenté soit par *mange*, soit par *mangent*. On peut utiliser des expressions régulières pour composer ces règles.

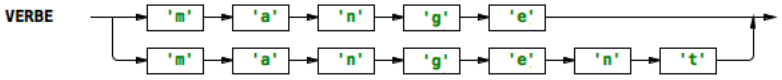

FIGURE 1 – Grammaire de VERBE

On trouve d'autre part **l'analyseur syntaxique**, ou parser, qui va donner une valeur à ces tokens issus du lexer dans le contexte grammatical. Par exemple la règle syntaxique du groupe verbal est que gv est composé d'un verbe (lexique) suivi éventuellement d'un groupe nominal (syntaxe, elle même à partir de lexique).

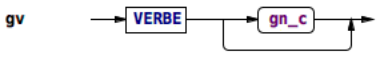

FIGURE 2 – Grammaire de gv

Par la suite, nous avons testé cette grammaire avec quelques exemples de phrases.
On peut observer de manière visuelle l'arbre généré par notre requête dans Antlr. Ici nous avons testé : *Pierre mange des bananes jaunes et des pommes vertes.*

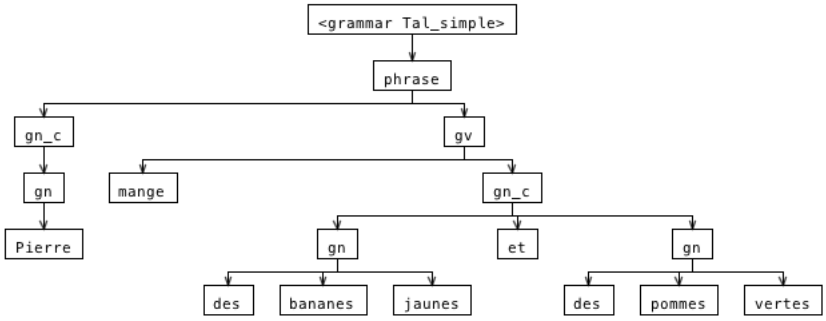

FIGURE 3 – Diagramme de la syntaxe de la grammaire Tal_simple

Cette phrase est syntaxiquement correcte, elle contient bien le lexique défini dans les règles et elle a du sens. Nous avons ensuite testé la phrase : *Des Marie vertes mangent des bananes et des Pierre.* Malgré son non-sens, cette phrase est conforme à la syntaxe du langage défini par la grammaire. C'est ici la faiblesse du travail par grammaire de type 2 qui permet de juger une phrase grammaticalement correcte mais pas sémantiquement. Cette question ne sera pas traitée dans le cadre du projet de LO17, on supposera donc ici que les utilisateurs feront des requêtes sémantiquement cohérentes.
Enfin, nous avons testé la grammaire sur Eclipse. Pour cela nous avons généré du code Java à partir de AntlWorks. Une classe est créée pour le lexer et une autre pour le parser, ainsi qu'un fichier contenant la liste des tokens. En plus de cela, dans notre projet de grammaire, on compte une classe pour la génération de l'arbre, ainsi que la classe Requete qui prend une saisie en console et renvoie donc la requête correspondante renvoyée par la grammaire.

2.1.3 Première grammaire complète

L'objectif suivant consiste à créer notre propre grammaire pour les requêtes sur le corpus. Pour cela, nous avons dans un premier temps imaginé les différentes requêtes en langage naturel possibles. Afin de se lancer dans la construction de la grammaire, nous avons sélectionné des paquets de structure simple, que l'on regroupe selon leur forme. Par exemple, « Je veux les articles qui parlent de... », « Je veux les articles concernant ... écrit le .... », etc. La structure nous paraissant la plus simple pour commencer est du type : "Je veux les articles qui parlent de microbiologie". Nous souhaitons obtenir une requête SQL qui serait alors "SELECT DISTINCT fichier FROM titretext WHERE mot='microbiologie'".
Pour cela, nous avons défini que la syntaxe "requête" est composé du lexique select (ici vouloir), suivi d'un article (article), puis d'un mot (parler), puis d'une variable qui est une chaine de caractère quelconque.

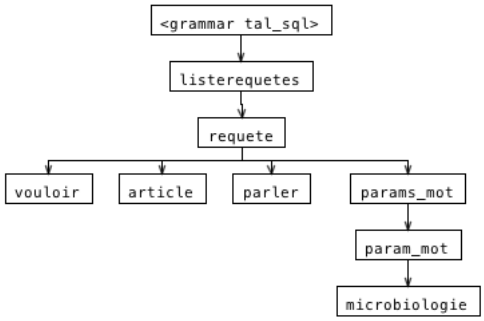

FIGURE 4 – Diagramme de la syntaxe de notre première grammaire

Après avoir généré la grammaire et l'avoir intégré dans notre projet Eclipse, nous pouvons saisir une requête simple en console, "vouloir article parler microbiologie" et obtenir la requête SQL.
Après cette étape, nous pouvons désormais nous pencher sur d'autres structures de phrases, plus compliquées cette fois, pour lesquelles nous pouvons implémenter une grammaire.
Mais avant tout, nous allons tenter de lier notre analyseur syntaxique avec notre analyseur morphologique précédemment mis en place afin de pouvoir formuler notre requête simple en langage naturel.

2.2 Formulation en langage naturel

N.B : Pour plus de clarté, nous emploierons dès à présent le terme de *"partie structure de la requête"* et de *"partie recherche de la requête"* ou variable pour désigner respectivement ce qui sert à formuler sa requête ("je recherche les auteurs qui ont écrit…", "donne moi les articles parlant de…") et ce qui concerne directement le corpus de texte ("microbiologie et laboratoire"....).

2.2.1 La lemmatisation

L'idéal serait maintenant de pouvoir formuler la recherche en langage naturel. Nous allons alors utiliser la saisie de l'analyseur morphologique : celui-ci, comme nous l'avons vu précédemment, va regarder le token. S'il le connaît, il le remplace par son lemme. S'il ne le connaît pas, il le remplace par un mot du lexique proche du token et s'il ne le connaît toujours pas, il le renvoie tel quel. Pour l'instant, nous utilisons le lexique de lemmatisation du corpus.

Nous avons donc intégré le projet de grammaire dans le projet correcteur. La requête en langage naturel saisie est alors envoyée directement au projet de grammaire, en argument dans une méthode qui fonctionne comme le main de saisie du projet grammaire.
Or, de cette façon, notre requête est mal traduite. Les tokens de la partie structure de la requête comme « voudrais », « article », « affiche », « qui contienne », « à propos » etc., n'existent pas dans notre fichier de lemmes. Il va donc falloir que nous complétions notre lemmatisation. Nous pouvons d'une part ajouter les mots manquant à notre lexique déjà existant, d'autre part créer un fichier lexique spécifique à la partie structure de la requête (voire plusieurs lexiques). Nous avons opté dans un premier temps pour la deuxième solution, en créant un nouveau lexique spécifique dont la lemmatisation va correspondre aux règles lexicales de la grammaire. Par exemple « voudrais », « veux », « désire », etc. auront tous le même lemme « vouloir » qui correspond à la syntaxe du SELECT.

Puisque nous utilisons deux lexiques différents, il faut se demander s'ils ne vont pas interférer l'un sur l'autre et nous faire obtenir des incohérences. Nous avons décidé d'appliquer tout d'abord la lemmatisation de structure, puis celle de recherche, car le lexique de structure est plus petit et risque moins de lemmatiser des variables que l'inverse. Si un mot est identifié dans le lexique de struture alors il ne sera pas recherché dans le lexique de recherche.Mais il reste tout de même un risque que des mots jouent à la fois le rôle de structure et celui de variable, et soient donc lemmatisé de la mauvaise façon. Par exemple, « Je veux les documents qui parlent *d'articles* de pêche ».
Dans un premier temps, nous avons simplement émis l'hypothèse simplificatrice qu'il n'y aura pas de mot de structure dans la partie document. Par la suite, une fois notre projet plus avancé, nous avons réfléchi à une façon de séparer notre requête entre structure et recherche. Nous avons notamment noté un problème avec le mot « recherche » que nous avons choisi de lemmatiser en vouloir. La requête en langage naturel « Je recherche les articles concernant la recherche fondamentale » nous donne la requête normalisée : « vouloir article parler *vouloir* fondamental », ce qui change totalement le sens voulu. Comment donc distinguer les deux sémantiques de ce token selon son emplacement dans la

requête ?
Nous sommes parties du constat que, généralement, après avoir employé le token de structure dont le lemme est « vouloir », l'utilisateur précise l'objet de sa recherche, qui, selon la conception de notre grammaire à l'heure actuelle, peut être un article, un bulletin, un auteur ou une rubrique. Notre solution a donc été de créer un troisième lexique de lemmatisation concernant le lemme vouloir, qui n'est appelé que si les termes cités précédemment n'ont pas encore été lemmatisés. Chaque token de la requête passe donc par au minimum 1 lexique et au maximum 2 (normalement tant que le lexique vouloir est actif, il n'y a pas de variables et tous les tokens sont lemmatisés avant d'atteindre le lexique recherche).

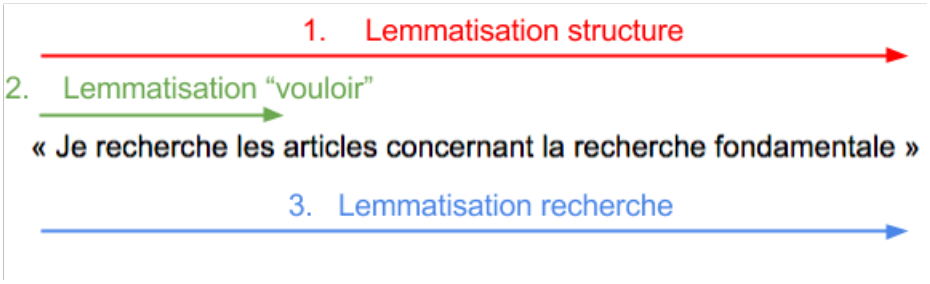

FIGURE 5 – Schema du fonctionnement de nos lexiques

Cette amélioration est la seule effectuée concernant cette problématique de tokens structure/recherche car c'est le seul problème qui s'est posé à nous. Cependant, nous aurions pu aller plus loin dans notre découpe de la requête. Nous avons notamment pensé à "article", qui dans tous nos exemples de requêtes n'apparaît qu'une fois dans la structure : nous aurions pu empêcher l'interprétation du mot article lors de sa deuxième occurence par exemple.

2.2.2 La stop-list

En plus des lexiques de lemmatisation, nous utilisons une stop-list dans l'analyseur morphologique, comme nous l'avons fait pour le corpus, qui va permettre de nous affranchir des mots qui n'apportent pas d'informations essentielles à notre requête. Nous avons décidé d'appliquer la stop-list de requête à toute la requête, car en effet, si elle s'applique à la partie "requête pure" elle pourra certainement s'appliquer au reste. On a donc créé une classe StopList dans notre analyseur morphologique. Celle-ci lit le fichier stop-list qui contient les mots à retirer et les enregistre dans une liste. Une méthode removeStopWords prend la requête en argument et parse un à un les mots. Pour chacun des mots, elle vérifie si celui-ci est contenu dans la liste. S'il ne fait pas partie de la stop-list, le mot est concaténé à la requête, sinon il est oublié.

2.2.3 Ajustement de l'algorithme de Levenshtein

Avec cette lemmatisation structure/recherche, on se retrouve avec les mots de recherche remplacés très facilement par des mots de structure lors de la première lemmatisation par Levenshtein. Ainsi : *"Je veux les zarticls qui parlent de robots et d'animaix"* donne *"vouloir vouloir article parler mot référence 05 "*. Si la valeur du seuil que nous avions choisi lors de la conception de l'analyseur morphologique nous paraissait afficher de bons résultats, il n'en est plus de même une fois intégré avec l'analyseur syntaxique. En effet, ce seuil est trop élevé pour la partie structure de la requête : on aimerait que l'algorithme de Levenshtein soit plus restrictif pour le lexique de structure.
Nous avons donc ajusté le seuil en le modifiant selon le lexique. Si la lemmatisation s'effectue sur la partie recherche de la requête, nous utilisons un seuil élevé de 0.80 $\times$ (la longueur du mot -1). Pour la

partie structure et pour le lexique “vouloir”, nous baissons ce seuil à seulement 0.25 × (la longueur du mot -1). Avec de tels seuils, on obtient bien pour notre phrase d'exemple : *“vouloir article parler robot et animal”*.

Il faut noter cependant que l'on perd en flexibilité au niveau des fautes de frappe dans la structure. Avant le changement de seuil, *“zaarrticls”* donnait *“article”*. Avec le nouveau seuil en revanche il n'est plus retrouvé. Néanmoins, on émet ici l'hypothèse que la requête ne contiendra pas trop de fautes de frappes au niveau de la structure. De plus, nous gardons tout de même une flexibilité tout à fait correcte grâce à ce seuil, et nous n'avons en parallèle plus affaire à des lemmatisations abusives des variables en lemme de structure. Nous avons donc atteint selon nous un bon équilibre.

2.2.4 Intégration SQL

La base de données utilisée par le projet est une base PostgreSQL. Elle contient 6 tables : titre-text, titre, texte, date, rubrique et numéro. L'API JDBC nous permet de communiquer avec la base de données depuis notre projet. Nous avons ensuite utilisé le programme interrogPostegresql.java qui nous était fourni, que nous avons placé dans un nouveau projet. Comment fonctionne la classe interrogPostegresql ?
Tout d'abord, à l'aide d'un objet InputStreamReader, le programme lit une requête entrée par l'utilisateur en console. La classe effectue ensuite la connexion avec la base de données à partir des paramètres donnés, c'est-à-dire l'url de la base de données, le nom de l'utilisateur et le mot de passe. Puis la requête est exécutée et les résultats sont affichés.
Cependant, dans le programme fourni, l'affichage des résultats est fixe : il est prévu que la première colonne contienne les articles et l'autre les bulletins. Or dans notre projet, en fonction de la requête, l'objet recherché doit varier. Nous avons donc modifié la partie affichage. Pour chaque ligne de résultat, on affiche autant d'objets que le nombre de colonnes.
Une fois ces modifications effectuées, on peut entrer une requête sql dans l'input et obtenir une résultat depuis la base. Ce nouveau projet SQL a alors été intégré dans notre projet de grammaire. Ce dernier, une fois la requête normalisée transformée sous forme SQL, envoie directement la requête en input de notre nouveau projet. Néanmoins pour certaines grammaires, la requête est mal formée, par exemple une jointure incomplète. Comme nous ne pouvons pas règler ce problème directement avec Antlr, nous avons opté pour le faire a posteriori, en ajustant les requêtes SQL incomplètes ou mal formées dans Eclipse. Nous avons donc implémenté une méthode de post-traitement par laquelle passe la requête après son analyse syntaxique. Nous développerons au cours de ce rapport la façon dont nous avons utilisé ce post-traitement pour chaque structure de requête.

2.3 Optimisation de la grammaire

2.3.1 Requêtes complexes

Les requêtes complexes comportent plusieurs paramètres de recherche et/ou de structure. Les éléments évoqués dans la suite du rapport peuvent être donc couplés qu'importe le nombre ou l'ordre des éléments. Pour les paramètres de recherche tout d'abord, nous avons fait une boucle les englobant ainsi que MOT. Ainsi, tous les mots tels que "publiés en", "parus dans", etc. entremêlés dans le texte sont pris en compte par notre grammaire. Nous avons aussi ajouté les mots ET et OU dans cette boucle, ce qui permet de lier les éléments avec "OR" et "AND" dans la requête SQL.
Le cas des paramètres de structure a été plus technique à traiter. En effet, des requêtes existent avec seulement des paramètres de recherche et aucun de structure. Nous ne pouvions donc pas faire une boucle similaire, surtout que cela créait des conflits pour certains paramètres tel que BULLETIN qui est à la fois un paramètre de recherche et de structure. Nous avons donc décidé de d'abord positionner

les paramètres de structure qui sont toujours facultatifs. Ensuite nous avons mis une boucle sur ET et à nouveau les paramètres structures elle aussi facultative. Ainsi il n'y a plus d'ambiguité sur les paramètres, et toutes nos requêtes complexes fonctionnent.

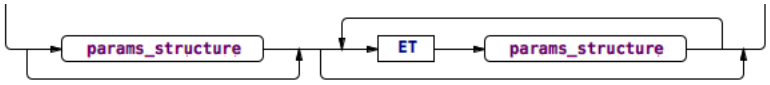

FIGURE 6 – Grammaire permettant de prendre en compte plusieurs paramètres de structure

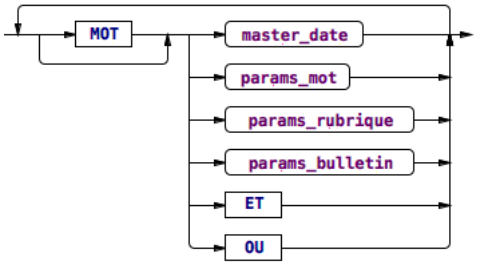

FIGURE 7 – Grammaire permettant de prendre en compte plusieurs paramètres de recherche

2.3.2 Date

Antlr Après avoir obtenu une première grammaire simple et l'avoir liée avec le correcteur orthographique et l'API Java pour interroger la base de données nous avons choisi d'étudier et de développer en profondeur les questions contenant une date. En effet les dates sont à la fois complexes puisqu'elles concernent une table à part, et en même elles comportent un nombre limité de syntaxes à traiter. Cela nous permet aussi d'étudier la jointure entre 2 tables. Ainsi, les autres types de recherche seront plus faciles à traiter car ils seront soient plus simples, soient sur le même modèle. Nous avons identifié quatre types de structure pour la date :

- date
- ENTRE date et date
- AVANT date
- APRES date

Nous avons donc identifié par un token dans Antlr chacun de ces trois mots pour permettre la reconnaissance du type de date recherché. Ensuite, pour identifier la date nous avons crée les tokens CHIFFRE qui correspond au jour ou au mois, et ANNEE qui correspond à l'année. Pour uniformiser les formats de date, nous avons transformé dans le lexique de structure chaque mois écrit en toutes lettres en son nombre associé. Par exemple juillet devient 07. Un autre défi réside dans les multiples formats de date et dans la reconnaissance du jour et du mois. Nous avons identifié plusieurs cas, il peut donc y avoir :

- Une année
- Un mois puis une année
- Un jour puis un mois puis une année

Nous avons donc estimé qu'il n'y avait un jour que si un mois était présent, ce qui nous a permis de reconnaître les deux. Enfin, nous nous sommes aperçues qu'un mot-clé était souvent présent dans la requête concernant les dates comme dans les phrases suivantes : Je veux les articles *parus* entre 2012 et 2014. Donne-moi les bulletins qui *datent* du 3 avril 2013. Quels articles ont été *écrits* avant août 2011. Nous avons alors fait le choix arbitraire de les regrouper sous le token DATER. Son usage a évolué

au cours du temps. Concernant les dates il est obligatoire pour la structure avec ENTRE et facultatif pour les autres. Plus tard nous nous sommes rendues compte que ces mots étaient aussi utilisés dans le cadre de l'auteur entre autre, et que son nom n'était donc plus adapté, nous avons donc alors fusionné avec MOT La grammaire adoptée pour une date simple est donc la suivante :

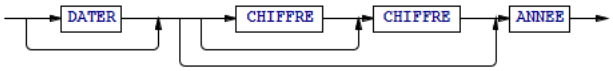

FIGURE 8 – Grammaire de date

Pour les dates avant et après seuls les mots clés correspondants ont été ajoutés. Pour la requête concernant ENTRE, nous avons considéré que les dates étaient symétriques, c'est-à-dire que si la première date était composée uniquement d'une année cela serait aussi le cas pour la seconde date.

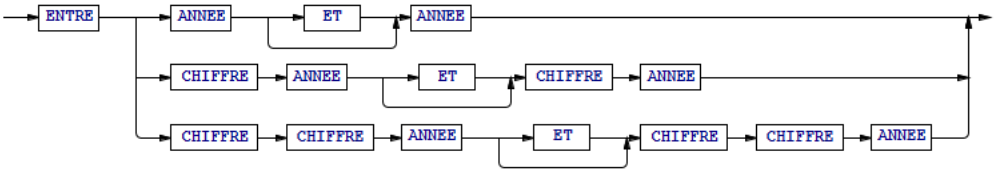

FIGURE 9 – Grammaire de ENTRE date

Concernant la requête SQL, nous l'avons construite de façon logique. Ainsi si une date doit être inférieure à une autre alors son année doit être strictement inférieure OU son année égale ET son mois strictement inférieur OU son année égale ET son mois égal ET son jour strictement inférieur. Pour chacun des cas énoncés précédemment nous avons donc construit la requête SQL suivant ce principe. Enfin, lorsqu'il nous nous sommes rendues comptes que les attributs de recherche s'accumulaient, nous avons réuni les différentes dates sous un paramètre nommé MASTER_DATE dans le but d'éclaircir et de rendre plus compréhensible le diagramme de syntaxe pour REQUETE.

Post-traitement Nous avons ensuite dû traiter les requêtes obtenues en post-traitement. La date est le premier attribut que nous avons traité, et il aurait pu surement être traité de manière élégante mais pas forcément plus efficace. Tout d'abord, nous allons vous présenter les remplacements en post-traitement qui sont valables pour tous les types de requête, mais qui ont été implémentés au fur et à mesure. La première action de post-traitement est d'identifier la table utilisée. Pour le moment il s'agit soit de *titretext* (si on recherche dans cette table ou dans la table titre qui elle sera traitée ultérieurement en post-traitement) soit d'une chaine de caractère vide. Les autres actions générales sont situées à la fin du post-traitement, il s'agit de supprimer un AND s'il est situé directement après le where ou s'il est présent en doublon, ou d'ajouter un AND s'il n'y en a pas entre deux groupes de parenthèses. Concernant ce dernier point nous avons du faire attention aux espaces entre les parenthèses qui ne sont pas toujours les même selon l'arbre dans Antlr. Ainsi cela permettait de régler certains problèmes d'ordre. Toutes nos requêtes marchent qu'importe l'ordre des attributs de la recherche. Concernant la date, nous avons traité trois cas. Si la table identifiée en début de post-traitement était vide, alors la requête porte uniquement sur la date, aucune jointure n'est nécessaire. Ensuite selon si la requête portait sur le fichier ou le bulletin la jointure avec la table titretext s'opérait sur l'un ou l'autre.

2.3.3 Combien

Un grand nombre de questions du corpus portant sur la quantité, nous les avons naturellement traité. En SQL pour obtenir la quantité d'un résultat il suffit de rajouter "count()" autour, ce que

nous avons fait au niveau de la grammaire avec l'attribut NOMBRE. Dans notre lexique de strucure, combien, compter, et nombre y ont été associé. Dans la grammaire nous avons choisi de créer une branche différente pour NOMBRE et les paramètres de structure afin de pouvoir facilement fermer la parenthèse ensuite. Cela aurait aussi pu être fait en post-traitement, mais nous avons jugé préférable de traiter tous les cas qui pouvaient facilement l'être dans la grammaire. Enfin NOMBRE peut être présent plusieurs fois dans la grammaire, et cela n'ajoutera que une fois "count(" à l'arbre pour prendre en compte les cas tels que "Compter le nombre...".

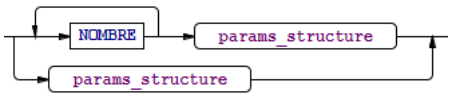

FIGURE 10 – Grammaire traitant le cas de nombre

2.3.4 Titre

Nous avons fait le choix de traiter uniquement le titre en post-traitement. Il aurait certes pu être inséré dans la grammaire, mais il était alors difficile de distinguer quels mots concernaient le titre par rapport aux autres éléments de recherche. Le mot titre est donc considéré comme un mot comme les autres par la grammaire, et le traite comme tel. Dans le post-traitement si le mot titre est présent, alors on supprime le morceau de SQL le concernant, et on remplace la table sur laquelle la recherche est effectuée, c'est-à-dire titretext par la table titre. Ils contiennent les même attributs il n'y a donc pas de modifications supplémentaires à effectuer. Nous avons conscience que cette solution, bien que traitant les cas les plus fréquents, reste limitée. Elle aurait pû être completée lors d'une version ultérieure, afin de prendre en compte la recherche sur un mot du titre et sur un mot du corpus par exemple. Cependant ce genre de questions n'étant pas présente dans le corpus donné nous avons préféré traiter d'autres cas.

2.3.5 Bulletin

Bulletin, en plus d'être un paramètre de structure est aussi un paramètre de recherche. Cependant à la différence des mots, un fichier est associé à un seul bulletin. Ainsi la requête "Je veux les articles des bulletins numéro 283 et 289" doit être transformé en

```
SELECT DISTINCT FICHIER FROM titretext WHERE (numero='283') OR (numero='289')
```

Cela doit aussi être pris en compte pour les autres paramètres de recherche. De plus nous avons pris en compte que les mots "bulletins" et "numéro" pouvaient apparaître plusieurs fois dans la requête et en différents endroits.

2.3.6 Auteur

Nous avons traité les recherches portant sur l'auteur comme celles portant sur l'article ou le bulletin. Cependant des termes concernant les articles ou des mots pouvaient aussi être présents dans n'importe quel sens et ont du être intégrés dans la grammaire. Ainsi les phrases contenant "les auteurs ayant publiés des articles sur" sont comprises dans notre grammaire.

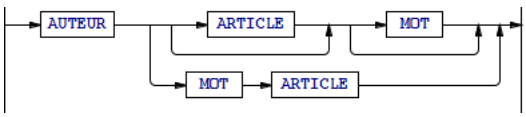

FIGURE 11 – Grammaire traitant le cas de l'auteur

La jointure avec la table email a été effectuée en post-traitement. Par ailleurs si une de ces requetes porte sur le numéro on précise alors sa table afin d'éviter toute ambiguité.

2.3.7 Quel, qui

Les questions commençant par "Quel" ou "Qui" ont pu facilement être ajoutés dans la grammaire. Nous les avons enlevé de la stop-list et associés aux termes VOULOIR et AUTEUR dans le lexique. Cependant, ces mots étaient dans la stop liste pour une raison, ils peuvent être présent à différents endroits sans toujours être pertinents. Par exemple "Je veux les articles *qui* parlent de. . . ." n'a aucun rapport avec l'auteur. Ces mots ont donc été ajoutés au lexique de vouloir, ils sont rattachés à leurs termes qu'avant certains mots de structure. Ensuite ils sont considérés comme des mots de la stop-list.

2.3.8 Rubrique

Les requêtes correspondantes à une recherche de ou par rubrique que nous avons regroupées sont de la forme « Je recherche les rubriques concernant. . . », « Quels sont les articles de la rubrique <x>. . . », « Quels sont les bulletins de la rubrique <x> parlant de. . . . ».
On remarque naturellement à chaque fois l'apparition du token « rubrique », éventuellement suivi par le nom de la rubrique concernée.
Nous avons donc créé le token « rubrique » sur Antlr. En apparaissant simplement après un select lors d'une recherche *de* rubrique, on va créer une requête de la forme « select rubrique from titretext where. . . ».
Pour une recherche *par* rubrique, nous avons développé une nouvelle règle syntaxique dans laquelle une rubrique est suivie par un « param_r », autant de fois que nécessaire si liée par des « ou » et « et ». Le « param_r » représente la variable qui sera selon nos prévisions le nom de la rubrique.

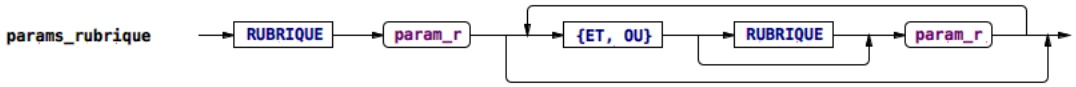

FIGURE 12 – Grammaire de params_rubrique

Lors de l'exécution sur Eclipse, on note néanmoins un problème pour les rubriques au nom composé : nous avons choisi de n'afficher qu'une variable pour décrire la rubrique. Une recherche sur « Du côté des pôles » donne donc « . . . WHERE rubrique = « pôle » ». En affichant plusieurs variables, les mots sont séparés et de plus certains ont été supprimés par la stop list : donc « . . . WHERE rubrique= « côté » rubrique = « pôle » ». Or pour effectuer la recherche en base de données, il nous faut nom exact des rubriques.
Nous avons répondu à cette problématique grâce à un post-traitement Java sur la requête qui va remplacer certains mots-clés par le nom de rubrique correspondant. Nous avons pris soin de mettre les mots-clés dans le lexique afin d'être sures qu'ils soient tous correctement reconnus et lemmatisés. Ainsi, la commande

```
Requete = requete.replace("( rubrique = 'pôle'  )", "( rubrique = 'du côté des pôles'  )") ;
```

va remplacer la recherche de la rubrique « pôle » par la recherche de la rubrique « du côté des pôles ». Cela permet donc d'obtenir un nom complet, de garder les mots supprimés par la stop-list, mais également de répondre au problème des différentes orthographes. En effet, pour une rubrique apparaissant sous différentes formes, on va rechercher en base sur une jointure de toutes ces différentes formes :

```
requete = requete.replace("( rubrique = 'innovation'  )",
"((rubrique = 'actualités innovation') OR
(rubrique = 'actualité-innovation') OR
(rubrique = 'actualité innovation') OR
(rubrique = 'actualité innovations'))");
```

Finalement, grâce à cette méthode, nous avons bien mis en place la recherche de et par rubrique, avec des requêtes fonctionnant comme : « Donne moi les rubriques ou l'on parle du nucléaire » et « Quels sont les bulletins de la rubrique du côté des pôles ou au cœur des régions concernant la géothermie ? ».

Le reproche que nous pouvons cependant faire à une telle méthode est qu'elle n'est pas vraiment scalable. Avec un nombre de rubrique très important, il aurait été plus compliqué de traiter au cas par cas comme nous l'avons fait. Il aurait alors fallu trouver un moyen pour que les variables correspondant aux noms de rubriques ne passent pas par le traitement par la stop list, soient éventuellement traitées par une lemmatisation spécifique et que tous les tokens soient regroupés comme n'en formant qu'un pour la requête SQL.

2.3.9 Variable simple

Une autre requête imaginable est une requête sous la forme d'une variable simple, par exemple : "Industrialisation". Pour cela, nous avons mis toute la partie de la requête SELECT, ARTICLE/BULLETIN/AUTEUR/RUBRIQUE, MOT en optionnel, ce qui permet d'écrire directement une variable. Avant traitement, on obtient juste une requête SQL de la forme : (mot = 'industrialis') . Dans notre méthode de post-traitement, nous avons donc précisé que lorsqu'une requête ne contient pas de SELECT, on ajoute "SELECT DISTINCT article, numero FROM titretext WHERE". Cela se base sur l'hypothèse que quand l'utilisateur tape une variable simple, celle-ci correspond à une recherche d'un mot dans le titre et le texte. De plus, nous affichons à la fois les articles et les bulletins afin d'englober plusieurs possibilités de recherche.

Une autre possibilité est que la variable recherchée soit une date. Par exemple l'utilisateur tape simplement "février 2012" pour avoir tous les articles et bulletins datant de février 2012. Nous avons donc changé l'ajout dans le post-traitement par "SELECT DISTINCT titretext.article, titretext.numero FROM titretext, date WHERE". Cette modification ne change rien au résultat de la recherche dans le premier cas et permet pour une date d'avoir une jointure correcte après post-traitement (comme nous l'avons vu dans la partie date). Pour l'exemple "février 2012", on obtient :

```
before treatment :  ( and date. =  and   (      ( mois = '02' and annee = '2012'  ) ) )
post treatment  : SELECT distinct titretext.fichier, titretext.numero FROM titretext, date
WHERE  ( date.fichier = titretext.fichier  and (( mois = '02' and annee = '2012'  ) ) )
```

Une autre solution aurait été de traiter cela directement au niveau de la grammaire, bien que cela aurait été alors été plus complexe.

3 Conclusion et pistes d'amélioration

Grâce à la grammaire que nous avons constituée sur AntlrWorks et au post-traitement effectué sur Eclipse, notre application permet déjà d'après nous de répondre à de nombreuses requêtes en langage naturel que pourrait effectuer un utilisateur, sous de nombreuses formes. De plus, la saisie de la recherche laisse une liberté de frappe grâce à notre correcteur orthographique, qui prend en compte les fautes d'orthographe dans une certaine mesure, les inversions de lettre et les fautes de frappe dûes aux touches côté à côte. Les seuils des lemmatisations, qui dépendent des lexiques utilisés ainsi que de la taille des mots, permettent de plus de ne pas corriger abusivement les mots de la requête.

Parmi les améliorations possibles, nous pensons qu'une découpe plus précise de la requête entre structure et variables aurait pu être appréciable. Elle aurait en effet permis d'utiliser les différents lexiques de manière bien séparée. Pour cela, on aurait pu s'appuyer sur le nombre d'occurences de certains mots ainsi que de leur positionnement dans la requête. Dans l'état actuel des choses, nous avons eu une réflexion à ce sujet qui nous a permis d'éviter les confusions principales.
De plus, notre projet de recherche ne nous permet pas d'effectuer de requêtes sur des termes consécutifs, par exemple "recherche fondamentale". On pourrait donc imaginer développer le projet en y utilisant un index bimots ou bien un index de position, qui permettraient de telles recherches.

Tout au long de ce projet nous nous sommes efforcées de conserver une grammaire robuste, capable de traiter de nombreux cas et à même de supporter des améliorations. En effet un des enjeux de ce projet était de construire une grammaire pas à pas, rajoutant des possibilités à chaque fois sans détruire le travail précédemment effectué. Notre grammaire, bien qu'imparfaite et loin d'être exhaustive, devrait être à même de supporter de futures améliorations sans perdre de sa qualité pour les cas déjà traités. Ainsi nous pouvons accumuler plusieurs parametres de recherche, peu importe leur ordre, nous aurons toujours un résultat cohérent. Une grammaire bien construite permet aussi de faire une utilisation raisonnable du post-traitement.

A Liste d'exemples de requêtes acceptées par notre système

— Je veux les articles parus en juin 2011.

— Donnez-moi des articles parus au 21 juin 2011.

— Listez-moi les articles écrits entre 03/2011 et 03/2012

— Listez-moi tous les articles écrits en 2012.

— je veux les articles parlant de laboratoire parus en 2011

— je veux les articles parus en 2011 parlant de laboratoire

— je veux les articles parlant de laboratoire de la rubrique focus

— je veux les articles de la rubrique focus parlant de laboratoire

— je veux les articles de la rubrique focus parus en 2011

— Je veux les articles sur "environnement" ou "innovation" sans "gouvernement".

— je veux les articles dont les bulletins sont 289 ou 290 parlant de laboratoire

— Je veux les articles qui a le numéro 291 dans la rubrique "Du côté des pôles".

— Je veux les articles sur "Philosophie" parus avant 2014.

— Combien d'auteurs ont publié l'article en janvier 2011 dans la rubrique Focus ?

— je veux chercher les articles dont le numéro est 288 et le titre contient Réalité virtuelle

— Quels sont les auteurs des articles du bulletin 279 ?

— Je veux les ariticles de la rubrique du cote des piles ou en direct des labos qui palrnt de géotermie mais pas de chimie

— Je veux le nombre de rubriques qui parlent de cheval

— Je veux les auteurs qui écrivent sur le nucléaire

— je veux les auteurs et les rubriques et les articles des bulletins 288 et numero 289

B Schémas de notre grammaire

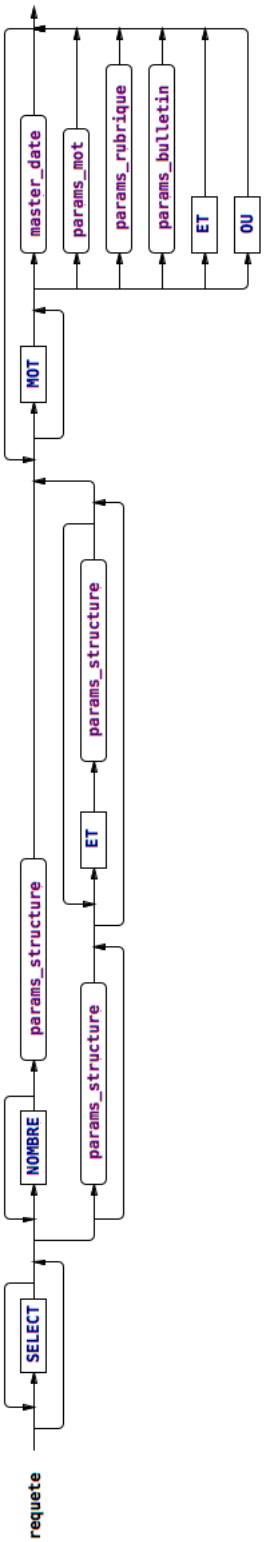

FIGURE 13 – Grammaire de requete

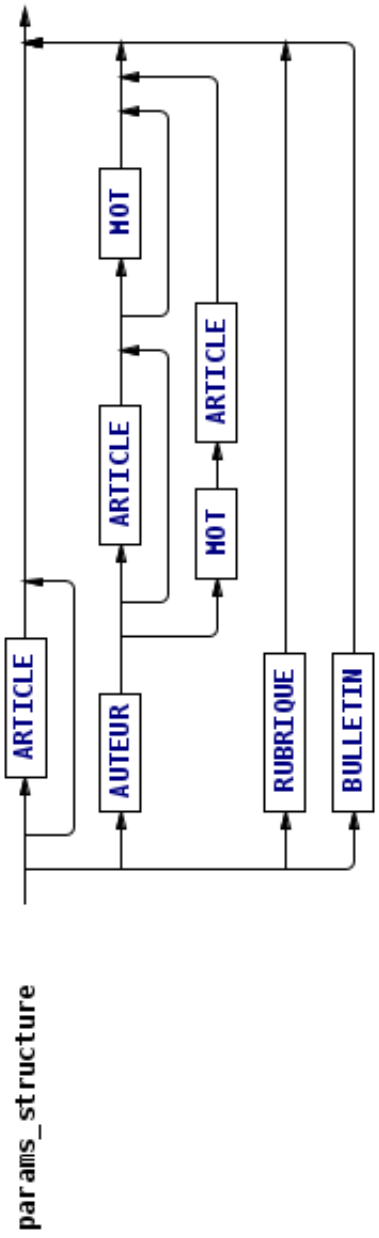

FIGURE 14 – Grammaire de params_structure