UNIVERSITE SORBONNE-NOUVELLE, PARIS 3
ILPGA
(INSTITUT DE LINGUISTIQUE ET PHONETIQUE GENERALES ET APPLIQUEES)

Apport de l'analyse en dépendances pour la génération de Q.C.M à partir de fichiers d'aide de logiciels

Roxane Anquetil

Directeur de mémoire : Kim Gerdes

Mémoire présenté en vue de l'obtention
du Master R&D « Ingénierie Linguistique »
et dans la perspective de l'obtention d'une
allocation doctorale.

Sorbonne-Nouvelle - Année 2010-2011

Remerciements

Ce mémoire n'aurait pas abouti sans une collaboration et un échange d'idées avec toutes les personnes qui y ont participé, directement ou indirectement, et je tiens à les remercier.

Tout d'abord, mes remerciements s'adressent à Kim Gerdes et Sylvain Kahane, pour leur encadrement dans la conduite de ces travaux.

Mes remerciements se portent aussi à l'université Paris 3 et plus particulièrement à l'ILPGA, qui a mis à ma disposition les conditions de travail nécessaires pour finaliser le présent mémoire.

Je tiens également à remercier Serge Fleury, directeur du master Ingénierie Linguistique, très à l'écoute de ses étudiants. J'ai apprécié sa disponibilité et le fait qu'il réponde toujours à mes mails, parfois inquiets, quant à la suite du cursus et à ses finalités.

Je remercie aussi Francis Corblin, un professeur de linguistique formidable que j'ai rencontré lors d'un séminaire à Paris 4 et dont l'enseignement m'a été très profitable.

Un grand merci à tous les professeurs qui m'ont suivi durant ces deux années, je pense notamment à ceux qui en particulier m'ont communiqué leur passion du TAL (Traitement Automatique des Langues) : Florence Amardeilh, Marcel Cori, Sylvie Despres, Sylvain Kahane, Kim Gerdes, Francis Corblin et Serge Fleury.

De manière générale, je suis formidablement heureuse d'avoir choisi le Master « Ingénierie Linguistique » et je remercie Serge Fleury de faire confiance et d'accepter dans le cursus, des personnes aux parcours atypiques.

Fées répandez partout La rosée sacrée des champs.
William Shakespeare - Le Songe d'une nuit d'été

Abstract

This work relates to Natural Language Processing (NLP), a scientific research field situated at the intersection of several disciplines such as computer science, linguistics, mathematics, psychology, and whose object is the creation of applications able to process linguistic data, oral or written.

In our case, we focus on the building of a syntax interface, using a parser, in order to generate Multiple-Choice Tests. The purpose of this work is to avoid the ambiguity generated by the bag-of-word approach. What is more the Multiple-Choice Tests we want to implement focus on tutorial corpora. This notion is of the utmost important to understand the purpose of this work, the main aim being to create an intelligent system which could judge whether or not the learner has understood the tutorial, lesson, notice etc. To achieve this goal, we have worked on a particular corpora, which we have carefully chosen, according to the theme of this work. This corpora is the help files of the software "Writer" developed by Open Office.

In the first part, we put forward the importance of dependency tree mapping, by explaining the whole theory. We also explain which kind of approach we have used to be able to analyze our whole corpora using Dependency Tree Mapping.

In the second part, we present what we exactly mean by Multiple-Choice Tests and Question Answering, and how we go beyond a key-word based matching in selecting answers to questions using dependency trees, augmented with semantic information such as named entities.

In the third part, we explain the limitations of our system and how we could go further using a semantic-syntax interface of our corpora.

Practically, this work makes two contributions to NLP and linguistics research :

-A complete syntactic analyze of help files of the software Open Office Writer

-An implementation of a Multiple-Choice Test system able to identify causative structures in tutorials and going beyond a key-word based matching.

The major future development which we would like to entertain during our thesis deals with the extension of our system to the semantic interface in order to explore new possibilities : more precised and well-focused generated questionnaires, summaries, presentations etc.

Résumé

Le présent travail s'inscrit dans le cadre du Traitement Automatique des Langues Naturelles (TALN), un domaine de recherche actuellement en plein essor, situé à l'intersection de plusieurs disciplines (informatique, linguistique, mathématiques, psychologie) et dont l'objectif est la conception d'outils informatique permettant de traiter des données linguistiques écrites ou orales.

Dans notre cas, nous nous appliquerons à construire une interface syntaxique dite en dépendances[1] d'un corpus de fichiers d'aide en utilisant un parseur syntaxique, dans le but de se servir de cette base pour générer des Q.C.M. Le but de ce travail est d'éviter l'ambiguïté générée par l'approche du mot à mot. De plus les Q.C.M que nous souhaitons implémenter sont créés à partir de tutoriels. Le but est en effet de créer un système intelligent, qui permettrait de savoir à quel point l'apprenant a une connaissance juste du sujet, grâce au pourcentage de bonnes ou mauvaises réponses données par ce dernier. Pour mettre en valeur ces principes, nous avons décidé de travailler sur le corpus des fichiers d'aide en ligne du logiciel OpenOffice Writer. Les fichiers d'aide de ce corpus se découpent en deux parties : l'une est composée des fichiers d'aide OpenOffice Writer en tant que tel et l'autre est composé de fichiers d'aide communs à tous les logiciels de la suite.

Dans la première partie, nous expliquons la nécessité de créer des Q.C.M à partir de textes contenant des connaissances. Pour cela, nous montrons quelles limites sont posées par une simple segmentation (chunking), avant de présenter les nombreux avantages de l'analyse en dépendances et la nécessité de liens fonctionnels.

Dans la seconde partie, nous expliquons plus en détails en quoi consiste l'analyse en dépendances et dressons un état de l'art des systèmes d'analyse syntaxique actuels. En outre, nous expliquons pourquoi nous avons décidé de choisir une grammaire de dépendance, plutôt qu'une grammaire syntagmatique. Nous expliquons ensuite que ces liens fonctionnels nous permettent d'extraire plus facilement des structures phrastiques intéressantes pour la création d'un système de questions-réponses. Ayant arrêtés notre choix sur les causations, nous poursuivons en analysant la structure des causations de notre corpus, comment nous les avons extraites et en quoi elles nous permettent de générer des Q.C.M.

[1] Nous reviendrons sur ce terme dans la Partie 2 de ce mémoire

Dans la troisième partie, nous établissons une ouverture vers la sémantique en expliquant notamment la théorie Sens-Texte, les grammaires d'unification, l'importance d'un lexique intelligent et soutenons que l'apport d'une interface sémantique-syntaxe permettrait de rendre notre système encore plus performant.

Ce travail apporte trois contributions au TALN :
-une analyse syntaxique en dépendance complète des fichiers d'aide du logiciel OpenOffice Writer
-une implémentation d'un système de QCM capable d'identifier des structures causatives dans les tutoriels et permettant une précision bien plus grande que la traditionnelle approche du « mot-à-mot ».

Durant notre thèse, nous aimerions entreprendre la création d'une interface syntaxe-sémantique dans le but de créer une fois notre tutoriel sémantisé, de nombreuses applications permettant entre autres : la génération de Q.C.M plus précis et mieux orientés, la génération de résumés, la création automatisée de supports d'apprentissage orientés etc.

Table des matières

PARTIE 1 - Création de Q.C.M à partir d'un texte contenant des connaissances

I. Introduction

Le but du présent mémoire est de présenter un système d'extraction automatique de questions-réponses grâce à l'apport d'une analyse syntaxique en dépendances et permettant de valider les connaissances d'un apprenant. Cette extraction est effectuée sur un corpus bien spécifique, puisqu'il s'agit des fichiers d'aide du logiciel OpenOffice. Deux critères ont orientés notre choix vers ce corpus :

1. Les fichiers devaient être libres de droit pour pouvoir présenter le projet librement et sans contraintes aussi dans une perspective à plus long terme de disposer d'un corpus annoté sémantiquement
2. Le corpus devait être issu d'un logiciel de bureautique ou d'informatique, puisque notre but était de nous concentrer sur l'extraction de QCM dans des fichiers d'aide

Le choix des fichiers d'aide comme base de notre projet n'est pas anodin. Un fichier d'aide, c'est un mode d'emploi du type tutorat, destiné en particulier au domaine informatique et permettant d'aider l'utilisateur novice à se former de manière autonome à un logiciel ou à un langage de programmation. Il possède plusieurs qualités pour notre travail :

1. Chaque logiciel informatique possède un tutoriel
2. La terminologie et la structure syntaxique partage souvent des traits communs d'un tutoriel à l'autre
3. C'est un produit déjà structuré, complet, possédant toutes les connaissances nécessaires pour la prise en main d'un logiciel
4. L'aspect technique nous permet d'éviter des phrases trop complexes à analyser pour un parseur syntaxique

Le corpus est automatiquement accessible à un utilisateur lambda , si ce dernier télécharge la version d'OpenOffice basique, dans laquelle les fichiers d'aide sont inclus. C'est un ensemble de dossiers dans lesquels se trouvent des

fichiers au format Xml. En terme de taille, le corpus est de 7, 97 MO. Il compte 1296 fichiers, 28 771 phrases et 299 966 mots. Les fichiers Xml ont tous une structure interne similaire, qui se découpe en deux parties : les métadonnées, entre balises « meta » et le corps entre balises « body ». Dans les métadonnées, nous trouvons le titre du chapitre auquel correspond le fichier, ainsi que le chemin vers le fichier. Dans le corps, il y a un découpage en sections (balises « section »), qui correspondent aux différents chapitres de l'aide, puis dans la section, nous trouvons des paragraphes (balises « paragraph »), des listes (balises «listitem »), ou encore des liens vers d'autres fichiers du corpus (balises « link »).

OpenOffice, depuis le rachat de Sun, appartient à Oracle. C'est un logiciel Open Source, ce qui signifie que c'est toute une communauté de développeurs, qui participent gratuitement à sa création et à son développement. Les fichiers d'aide ont donc été réalisé par la communauté. Le but énoncé est d'offrir une alternative à Microsoft Office en utilisant la communauté pour « créer la suite bureautique internationale leader, tournant sur les principales plates-formes, et fournissant l'accès aux fonctionnalités et aux données via des composants et API ouverts et un format de données XML ».

Dans le présent travail, nous convertissons volontairement le corpus au format Xml en un corpus au format Texte brut, précisément pour permettre une analyse syntaxique en dépendances[2]. Nous verrons que le balisage Xml, si tant est qu'il puisse être utile dans un premier temps pour créer facilement une ontologie ou résoudre certaines anaphores, est en réalité une alternative à éviter. Dans la perspective d'un système d'extraction automatique qui fonctionnerait sur tout type de tutoriels, notre input serait le tutoriel au format texte brut, puisque nous ne pourrions pas prévoir qu'un format Xml soit disponible.

II. Du problème du simple chunking

En partant d'une phrase de notre corpus et en la découpant en mots, nous découvrons que la génération de questions-réponses pose problème, du fait du manque de liens fonctionnels connus entre les mots. Prenons par exemple la phrase suivante :

- « Pour centrer une image sur une page HTML , insérez l' image, ancrez-la "comme caractère ", puis centrez le paragraphe.»

[2] Nous exliquons ce terme plus en détail dans la Partie 2 de ce mémoire

Le découpage en mots ne nous permet pas de construire des questions-réponses véritablement cohérentes à partir de cette phrase. La question que nous souhaiterions générer est la suivante :

- « Que faire pour centrer une image sur une page HTML? »

Pour cela, si nous n'avions que le découpage en mots, nous pourrions extraire la première proposition allant de « Pour » jusqu'au mot précédant le verbe conjugué à la deuxième personne du pluriel de la seconde proposition. On se rend déjà compte que cela est bien loin d'une analyse linguistique et qu'il suffit que la première proposition comporte déjà un verbe conjugué à la deuxième personne du pluriel pour que le système ne retrouve pas la bonne question. Il aurait été possible de se passer de l'analyse syntaxique, mais les cas spéciaux auraient été beaucoup plus nombreux et l'on aurait dû prévoir tout ce que le parseur sait déjà faire. Cela nous aurait amené à construire un parseur spécialisé pour la reconnaissance de certaines structures. La situation se complique encore plus lorsqu'il s'agit de générer les réponses associées aux questions. On aimerait en effet générer la réponse :

- « Il faut insérer l'image, l'ancre "comme caractère", puis centrer le paragraphe.»

Sans analyser la phrase syntaxiquement, il n'est pas possible de savoir que le verbe principal « insérer » est relié à la préposition « Pour », de telle sorte que la proposition introduite par la préposition « Pour » dépend syntaxiquement du verbe principal « insérer». Autrement dit, sans la deuxième proposition, la première n'a pas lieu d'être. Il y a ici un mécanisme de cause à effet, qui n'est pas détectable par le simple découpage en mots. Au contraire une analyse syntaxique en dépendance nous permettrait d'extraire tous les liens fonctionnels de la phrase.

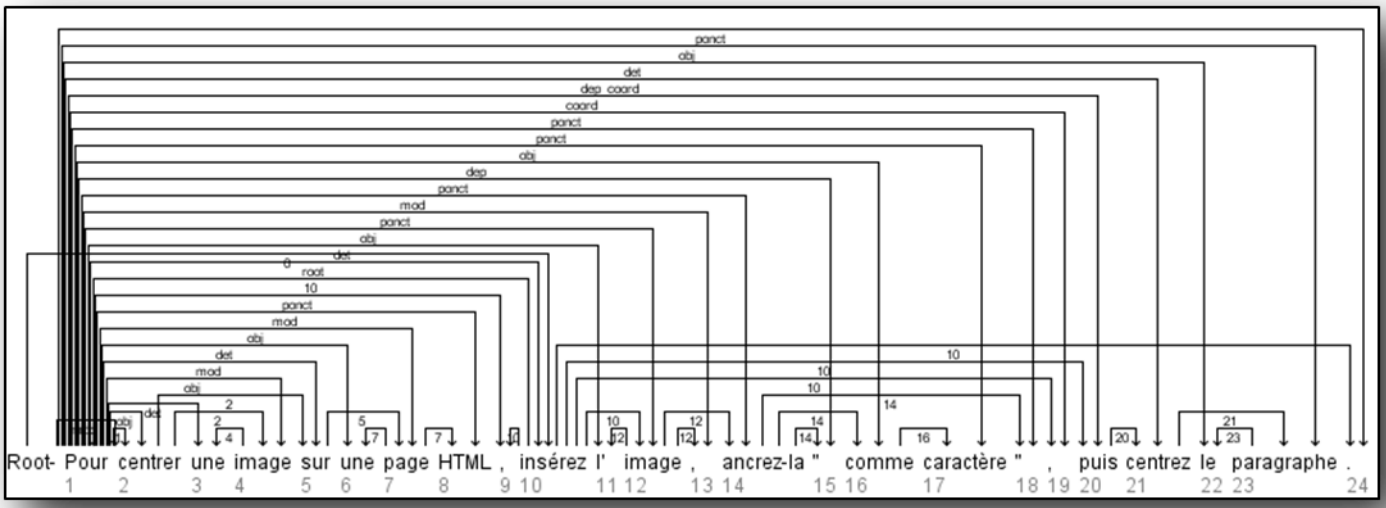


Figure 1 – Exemple d'une analyse en dépendances

Dans *Vasin Punyakanok, Dan Roth, Wen-tau Yih*, article intitulé *Natural Language Inference via Dependency Tree Mapping : An Application to Question Answering* , l'importance de l'analyse en dépendances est mise en valeur par un exemple pertinent. Dans cet article, la question dont les auteurs aimeraient avoir la réponse à l'aide d'un corpus est la suivante :

- "What is the fastest car in the world?"

La technique dite du « sac de mots[3] » ne permet pas une bonne analyse, puisque lors de la recherche dans le corpus, il n'y a aucun moyen de savoir que « fastest » dépend syntaxiquement de « car » ou non, ce qui occasionne les réponses candidates suivantes :

1. "The jaquar XJ220 is the dearest, fastest and most sought after car in the world."
2. "… will stretch Volswagen's lead in the world's fastest growing vehicle market"

[3] Bag of words en anglais

Tandis que dans la première réponse, « fastest » dépend directement syntaxiquement de « car », ce n'est pas le cas dans la deuxième, dans laquelle « fastest » est relié syntaxiquement à « market ». Seule une analyse en dépendance nous permettrait d'éviter cette erreur et de reconnaître le bon schéma. Dans l'illustration ci-dessous, nous remarquons que l'analyse en dépendances nous permet de visualiser que « fastest » est bien relié à « car » dans la première phrase, mais pas dans la seconde. Cependant, on observe qu'il y a tout de même une proximité plus grande entre les deux mots dans la

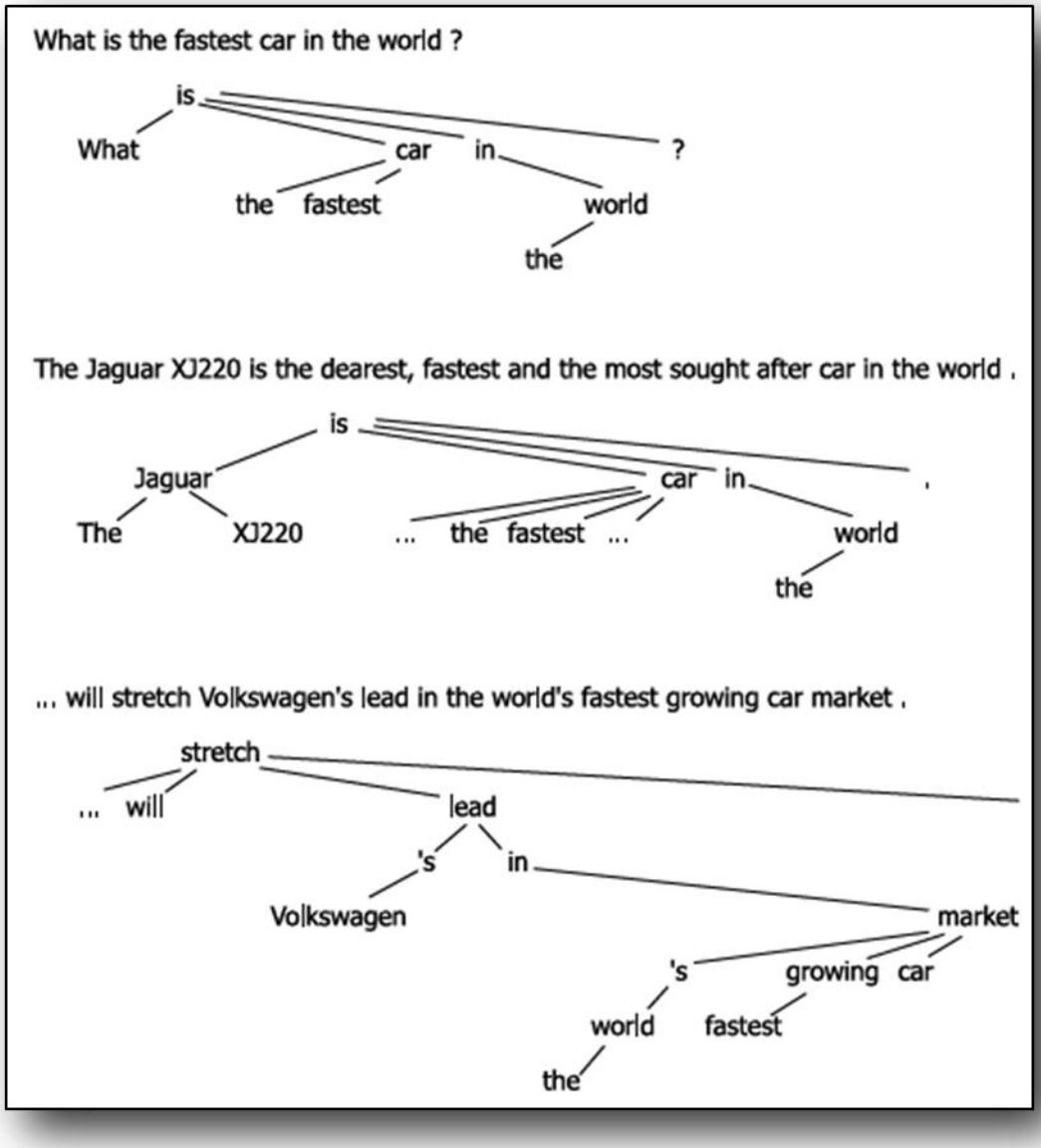


Figure 2 - Simple chunking vs. Analyse en dépendances

première phrase. Donc, grâce à des méthodes statistiques, nous pourrions réussir à extraire les bons candidats. Mais le but de notre recherche n'est pas basé sur l'extraction. Au contraire, nous voulons utiliser précisément le contenu d'une phrase, dans le but de privilégier la réalité linguistique de cette dernière.

III. La nécessité de liens fonctionnels pour extraire des questions-réponses cohérentes

Notre but est de mettre en place un système qui permettrait à des apprenants de tester leurs connaissances d'OpenOffice. Dans une perspective plus lointaine, le but est de pouvoir appliquer ce système à tous les fichiers d'aide de logiciels. Donc, notre système doit être capable de générer des questions-réponses cohérentes syntaxiquement et sémantiquement. Le sens de la phrase doit être restitué dans ces questions-réponses. Nous avons besoin pour cela de plusieurs choses.

Tout d'abord, une représentation des phrases qui capture l'information essentielle dans le but de permettre une corrélation questions-réponses plus pertinente. Ensuite, un système qui à une question extraite va générer la bonne réponse associée, ainsi que deux ou trois mauvaises réponses. Comme établi précédemment, les liens fonctionnels sont nécessaires et seule une analyse syntaxique de nos phrases peut permettre une extraction pertinente de questions-réponses. Il s'agit maintenant de trouver quels « types » de phrases il serait le plus judicieux d'extraire en priorité pour que notre système soit le plus efficace possible. Dans cette perspective, il faut cependant garder en tête une chose fondamentale pour une recherche à plus long terme : l'analyse syntaxique n'est qu'un premier pas vers une analyse sémantique complète du corpus qui seule pourrait permettre la génération automatique de tutoriels complets, systèmes de questions-réponses etc. Prenons une autre phrase de notre corpus :

- « Pour prévisualiser le modèle ou le document, cliquez sur l'icône Aperçu de la zone d'aperçu à droite de la boîte de dialogue.»

Nous pouvons déjà affirmer qu'on a ici un cas d'une description d'un lien de cause à effet. D'un côté nous avons la cause:

- « cliquez sur l'icône Aperçu de la zone d'aperçu à droite de la boîte de dialogue»

De l'autre nous avons l'effet :

- «prévisualiser le modèle ou le document»

Cela signifie que l'accomplissement d'une action « x », que l'on nomme la cause, permet l'exécution d'une action « y », que l'on nomme l'effet. La détection des structures de cause à effet, que nous appellerons également « causations», est possible par l'intermédiaire d'une analyse syntaxique et semble un choix très judicieux puisque ce sont des phrases qui comportent toujours une question et une réponse. L'analyse syntaxique nous permet alors de séparer d'un côté la cause - qui n'est autre que le verbe racine relié à tous ses dépendants directs -, de l'effet – qui n'est autre que la proposition de la phrase, qui dépend syntaxiquement de la première.

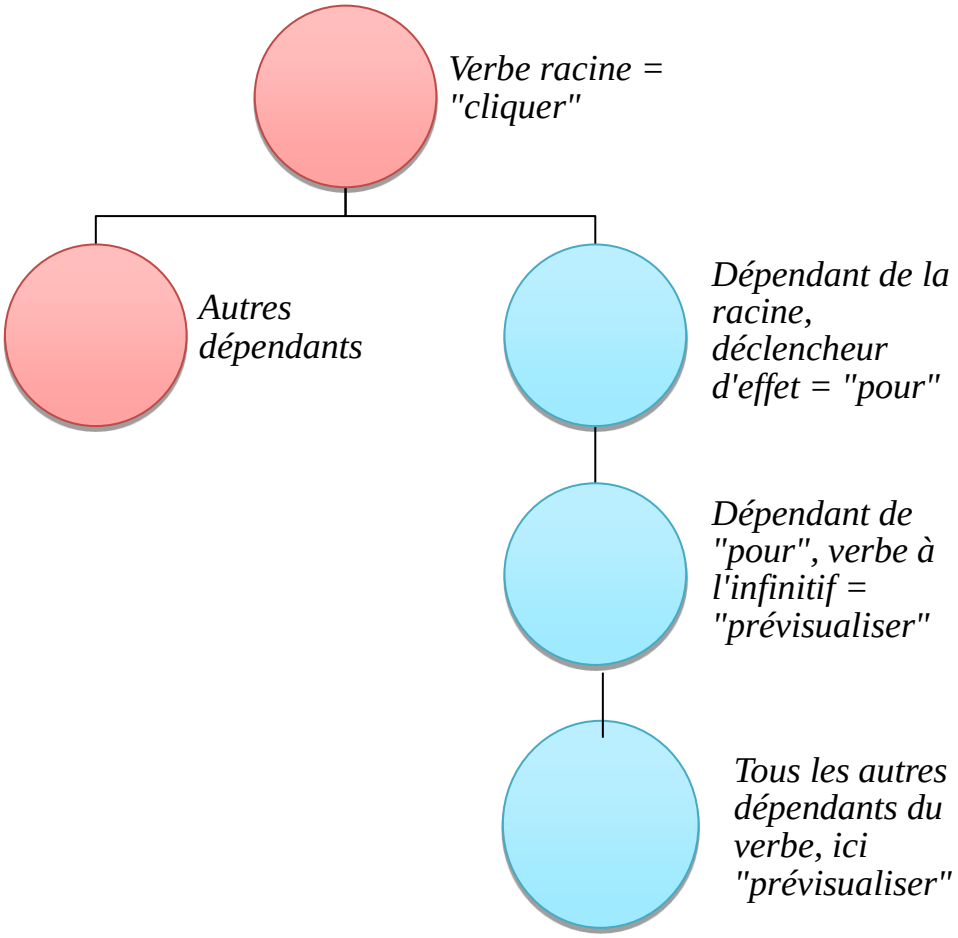


Figure 1 - En rouge LA CAUSE/ En bleu L'EFFET

Nous observons donc, qu'en ayant la possibilité de naviguer dans l'arbre syntaxique, il est ensuite possible d'isoler très facilement, dans une structure de cause à effet, la cause de l'effet.

A l'inverse, le système appelé « sac de mots », devrait, pour isoler par exemple les structures de cause à effet en « pour », utiliser une expression régulière qui chercherait de « pour » jusqu'à la virgule suivante. Mais un simple exemple nous prouve que ce bricolage, qui n'est basé sur aucune analyse linguistique, mènerait à de nombreuses incohérences.

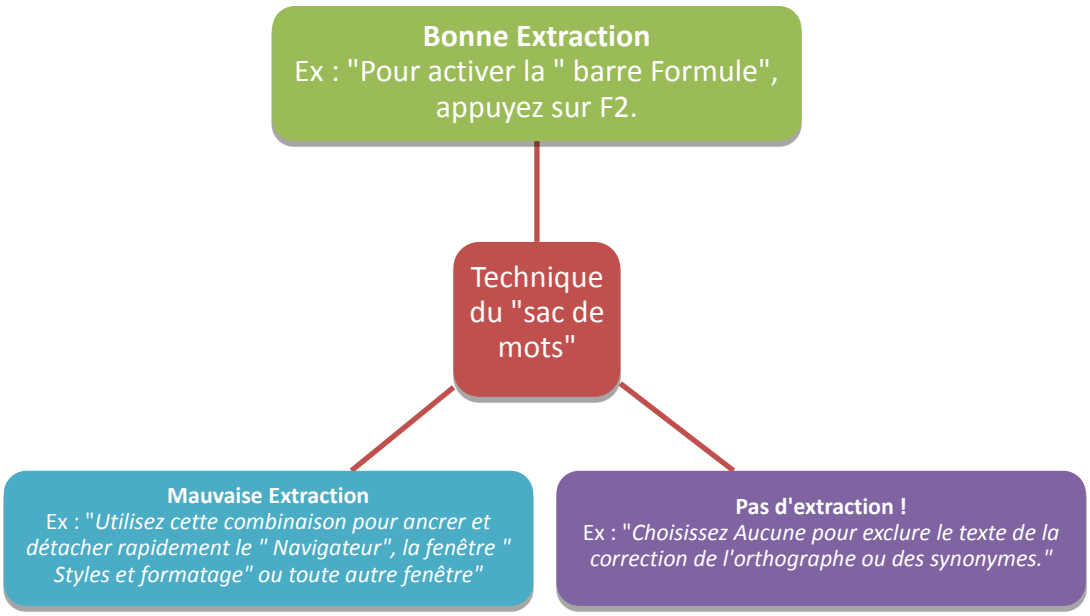


Figure 2 - La technique du "sac de mots", par le biais d'expressions régulières, conduit à un très mauvais résultat

Ces conclusions étant tirées, une analyse en dépendances est ce qui semble le plus cohérent pour mettre en place notre système. En effet, une fois les structures de cause à effet extraites du corpus, il est ensuite aisé d'en générer des questions-réponses, puisque la cause répond à l'effet et vice-versa. Dans la phrase précédemment présentée, nous pouvons donc générer deux questions :

- « Que faire pour prévisualiser le modèle ou document ? »
- « Que se passe-t-il lorsque vous cliquez sur l'icône Aperçu en haut de la zone d'aperçu à droite de la boîte de dialogue ? »

Les deux réponses associées sont respectivement :

- « Cliquer sur l'icône Aperçu en haut de la zone d' aperçu à droite de la boîte de dialogue. »
- « Vous prévisualisez le modèle ou document. »

Un programme informatique auquel nous donnons la structure syntaxique de cette phrase, générerait alors sans problème les questions-réponses associées, puisqu'une fois la cause et l'effet connus, il ne reste plus qu'à les transformer en interrogatives en ce qui concerne les questions et en déclaratives en ce qui concerne les réponses, la structure de base de la phrase n'étant alors que très légèrement modifiée. Pour nous en convaincre, nous avons dressé un état de l'art de l'analyse syntaxique, dans le but de justifier notre choix auprès du lecteur, puis nous avons mis au point un système permettant de mettre en pratique cette théorie.

PARTIE 2 – Analyse en dépendances et génération de QCM par l'extraction de structures causales

I. Introduction

Nous avons fait volontairement le choix d'une analyse syntaxique, dite « en dépendances » de notre corpus. Dans cette partie, nous expliquons donc en quoi consiste exactement cette analyse en dépendances, notamment en dressant un état de l'art exhaustif des différentes analyses syntaxiques possibles. Il y en a principalement deux : l'analyse en constituants et l'analyse en dépendances. Le choix de l'analyse en dépendance n'est pas anodin, puisque c'est une analyse syntaxique plus fine, permettant d'accéder plus facilement aux liens fonctionnels entre les différents mots d'une phrase donnée. Nous poursuivons ensuite ce chapitre, en expliquant quelles types de cause à effet nous avons pu isoler et comment les liens fonctionnels donnés par l'analyse syntaxique, nous ont permis d'extraire des relations de cause à effet de notre corpus. Enfin, nous expliquons comment ces causations nous permettent de générer un QCM cohérent. Une présentation de l'application finale est alors dressée.

II. Représentation syntaxique d'une phrase

Les grammaires de dépendance sont un sous-ensemble des grammaires permettant de générer la représentation syntaxique d'une phrase.
On dénombre deux types de grammaire permettant de générer la représentation syntaxique d'une phrase : les grammaires syntagmatiques et les grammaires de dépendance.

a. Les grammaires syntagmatiques

Ce type de grammaire a été introduit par Noam Chomsky entre 1957 et 1965. Ce sont des grammaires de constituants destinées à décrire la base de la composante syntaxique, par opposition aux « grammaires transformationnelles », qui transforment les phrases de base en phrases dérivées. Les règles des grammaires syntagmatiques caractérisent certaines catégories dans les termes de leurs constituants. Par exemple, le groupe nominal, GN ou

SN, est constitué d'un déterminant et d'un nom. Ces règles syntagmatiques sont des « règles de réécriture » de la forme générale XAY → XZY, qui signifie que l'élément A se réécrit Z dans le contexte X – Y. Et l'on distingue deux types de règles syntagmatiques, selon que X et Y sont, ou ne sont pas, nuls. Si X et Y sont nuls, les règles sont dites « indépendantes du contexte ». Par exemple la règle P → GN + GV (une phrase se réécrit groupe nominal + groupe verbal), est dite indépendante du contexte, cat elle s'applique sans aucune limitation contextuelle.

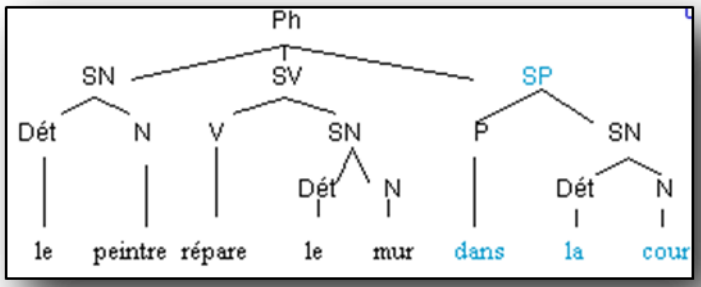


Figure 1 - Analyse Syntagmatique d'une phrase simple

Mais la grammaire générative transformationnelle de Chomsky a rapidement fait place aux grammaires d'unification à la fin des années 70. L'une des critiques majeures fut que la description de phénomènes syntaxiques dans des langues variées rencontrait un certain nombre de difficultés. Ainsi, les représentations arborescentes classiques des grammaires transformationnelles semblaient peu adaptées à la description des langues dites « non configurationnelles ». Ce sont des langues dont l'ordre des mots est assez libre et qui présentent de nombreux constituants discontinus. Par ailleurs, la grammaire générative transformationnelle se heurtait à des problèmes d'implémentation.

Les grammaires d'unification, qui visent à intégrer toutes les caractéristiques des langues naturelles sont également des grammaires génératives, puisqu'elles proposent des représentations explicites et formalisées des connaissances linguistiques et que ces représentations visent à rendre compte de phénomènes linguistiques universels valables pour toutes les langues. Elles se distinguent cependant de la grammaire générative transformationnelle de Chomsky sur plusieurs points :

- elles englobent tous les phénomènes linguistiques dans une seule et même sphère
- elles ne supposent pas de parallélisme entre structures syntaxiques, sémantiques et morphologiques d'une phrase donnée
- elles ne mettent pas en avant des représentations sous forme d'arbre

Cela amène à des difficultés de formalisation des grammaires transformationnelles. En témoigne la quasi-inexistence d'implémentations d'une grammaire transformationnelle. La plus proche est sans doute TAG, dont certaines descriptions linguistiques utilisent des arbres avec des traces, mais ils ne sont pas vraiment utilisés dans l'analyse proprement dite.

Dans la catégorie des grammaires d'unification, on peut citer les trois modèles les plus connus :

- **LFG** (lexical functional grammar) ou grammaire lexicale fonctionnelle définie à la fin des années soixante-dix par Joan Bresnan et Ronald Kaplan. C'est un modèle à la capacité générative plus restreinte que celle des grammaires transformationnelles, évitant ainsi la surgénération et permettant une plus grande efficacité. Le composant transformationnel est ainsi remplacé par un niveau « lexical-fonctionnel ». La structure syntaxique de la phrase est décrite par une représentation arborescente et par une structure de traits qui permet de coder directement les différentes fonctions grammaticales, appelées également « f-structure ». La structure des constituants syntaxiques est également représentée, « c-structure ». D'autres structures sont potentiellement représentables (structure morphologique, phonologique, sémantique etc.).
- **HPSG** (Head-driven Phrase Structure Grammar) a été conçu au début des années 80 par C. Pollard et I. Sag. Elle repose sur l'idée de représenter les règles, les éléments lexicaux et les principes eux-mêmes de la langue par des structures de traits. Le but de cette grammaire est de proposer une organisation des connaissances linguistiques qui soit modularisée, non redondante et implémentable. À la différence de la grammaire générative, qui ordonne les différents niveaux d'analyse de la langue les uns à la suite des autres, HPSG est une grammaire qui permet une analyse de la langue en parallèle. Cette grammaire est donc à la fois modulaire et multilinéaire. De ce fait, l'analyse est ascendante, c'est-à-dire que chaque mot contient

son programme linguistique. On part du mot pour arriver à la phrase finale.

- **TAG** (Tree Adjoining Grammar) ou grammaire d'arbres adjoints a été définie au milieu des années soixante-dix comme modèle mathématique par A. Joshi et al. Tout comme les concepteurs des grammaires LFG et HPSG, l'une des motivations était de construire un système à la capacité générative plus contrainte que celle des grammaires transformationnelles. En réalité, c'est une grammaire dont les règles sont des arbres élémentaires se combinant entre eux pour former des arbres plus grands. Deux opérations de base sont associées à ces arbres, à savoir la substitution et l'adjonction. De plus, il possède un typage lexical riche . Ces trois éléments, lexique, shémas d'arbres élémentaires et opérations d'adjonction ou de substitution, permettant de générer les arbres dérivés et les arbres de dérivation de la phrase.

b. Les grammaires de dépendance

Ce sont ces grammaires qui vont nous intéresser dans le présent mémoire. Non pas que nous rejetons les grammaires syntagmatiques, mais il se trouve que la notion de dépendance syntaxique est plus proche de la sémantique et le but est justement de mettre en place le début d'une analyse sémantique. A fortiori, elles sont mieux adaptées au système de Q.C.M que nous voulons mettre en place, puisque nous pouvons ensuite facilement naviguer dans les feuilles des arbres syntaxiques, à la manière en quelques sorte d'une navigation dans un arbre Xml.

Mais d'abord, qu'est-ce que la notion de dépendance syntaxique ? C'est le fait que la structure syntaxique soit déterminée par un ensemble de relations qui expriment la façon dont les mots dépendent les uns des autres. Lucien Tesnière est le premier à avoir introduit les grammaires de dépendance, en 1959, avec son ouvrage intitulé *Eléments de syntaxe structurale.* Dans la grammaire de dépendance, les relations se situent entre les mots et non plus entre les constituants. On distingue alors des mots source, que l'on appelle des « têtes » ou « gouverneur » et des mots cibles que l'on appelle des « dépendants » ou « gouvernés ».

c. Principes fondamentaux

Nous dénombrons cinq principes fondamentaux au cœur des grammaires de dépendance, à savoir :

- la valence : En linguistique, la valence d'un verbe, c'est le nombre d'actants qu'il peut recevoir ou qu'il doit recevoir pour être saturé, c'est-à-dire fournir un syntagme grammaticalement correct.
- la connexion : « Tout mot qui fait partie d'une phrase cesse par lui-même d'être isolé comme dans le dictionnaire. Entre lui et ses voisins, l'esprit aperçoit des connexions, dont l'ensemble forme la charpente de la phrase. »
- la rection : c'est le phénomène syntaxique par lequel les mots sont connectés entre eux de telle sorte qu'il y ait toujours entre deux mots un régissant et un recteur
- la jonction : c'est la relation entre deux éléments de fonction équivalente (c'est l'équivalent de la coordination en grammaire traditionnelle)
- le stemma : c'est l'arbre de dépendance syntaxique rendant compte de tous les liens de dépendance syntaxique qu'entretiennent les mots au sein d'une phrase

Le système de Tesnière est donc constitué de régissants et de subordonnés. Il est à noter qu'un mot peut être à la fois régissant et subordonné. Par exemple dans la phrase « Mon petit frère mange. », le mot « frère » est à la fois subordonné au verbe « manger » et régissant de l'adjectif « petit » et de l'article « mon ». On dit qu'il est régissant de ces deux mots, puisque sans lui, ces deux mots n'existent pas.

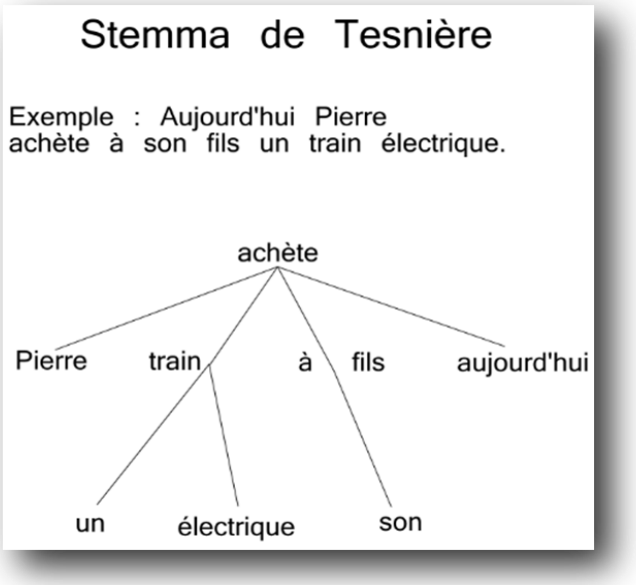


Figure 2 - Analyse en dépendances d'une phrase simple

d. Lien entre grammaires de dépendance et sémantique

D'un point de vue purement formel, les grammaires syntagmatiques avec tête et les grammaires de dépendance sont mathématiquement équivalentes (Robinson, 1970). Cependant, les grammaires de dépendance qui permettent une dissociation de l'ordre linéaire des mots et de la structure syntaxique, se rapprochent davantage de structures sémantiques, qui font fi de l'ordre des mots. C'est la raison pour laquelle la théorie Sens-Texte a jugé préférable d'utiliser la grammaire de dépendance dans le but de créer une interface Sémantique-Syntaxe. C'est également la raison pour laquelle ce système est plus approprié à la recherche de questions-réponses ; une fois les phrases analysées syntaxiquement nous pouvons facilement naviguer dedans pour trouver des schémas de questions, de même nous pouvons facilement retrouver les réponses à ces questions. Dans le présent travail, nous nous focalisons sur l'extraction d'une seule relation sémantique, la « cause », puisque pour le moment, nous pouvons seulement bénéficier de la structure syntaxique. Mais, dans le cadre de notre thèse, en mettant au point une interface Syntaxe-Sémantique, nous pourrions alors affiner notre recherche et observer qu'il y a différentes manières d'exprimer une causation.

Il est donc essentiel de comprendre en quoi consiste la représentation sémantique, dans le but de voir à plus long terme que le présent travail. Dans le cadre d'une thèse, notre système de Q.C.M bénéficierait en effet des avantages

d’une telle représentation. Or, pour pouvoir créer cette structure sémantique, une analyse syntaxique en dépendances est primordiale. Cela justifie encore une fois notre choix.

III. Extraction de structures causales

a. Que sont les structures de cause à effet ?

Ce sont des structures dans lesquelles le sujet est à l'origine de l'action, mais ce n'est pas lui qui l'accomplit pour autant ! Il la fait faire (cas à sens actif) ou la fait subir (cas à sens passif).

Il y a trois éléments fondamentaux dans une structure de cause à effet:

1. un événement
2. quelque-chose qui le cause
3. une relation causale entre les deux

Observons l’exemple suivant : « [Les images contenues dans un tableau et dont la taille n'est pas indiquée (='X')] peuvent **causer [**des problèmes d'affichage si la page parcourue utilise une ancienne norme HTML.(='Y')] »

1. événement => problèmes d'affichage
2. cause => images contenues dans un tableau et dont la taille n'est pas indiquée
3. relation causale => c'est le fait que les images contenues dans un tableau sans indication de taille causent des problèmes d'affichage

i. Les verbes de causation et les verbes causatifs

On distingue les verbes de causation et les verbes causatifs. Les verbes de causation expriment la causation en tant que telle (causer, entrainer, pousser, provoquer). L’exemple précédemment donné inclut un verbe de causation :

- « Les images contenues dans un tableau et dont la taille n'est pas indiquée peuvent **causer** des problèmes d'affichage si la page parcourue utilise une ancienne norme HTML. »

Quant aux verbes causatifs, ils incluent dans leur sens l'effet produit par la causation (construire, cuire, briser etc.). Nous n'avons pas rencontré de structures causatives dans le présent mémoire, donc nous n'en parlerons pas. Voici tout de même un exemple de structure causative :

- « Pierre fait chanter Marie. »

b. Classification des causations de notre corpus

Nous devons nous poser la question suivante : quelles conditions doivent être réunies pour que l'on puisse en français, parler d'une phrase exprimant la causation ? Si l'on représente la causation de façon formelle, on distingue toujours un déclencheur et un résultat :

Déclencheur	Résultat
Utiliser le navigateur	Déplacer des titres
Travailler dans une mise en page Web	Certaines options de numérotation non disponibles
Cliquer sur l'icône Aperçu : plusieurs pages	La boîte de dialogue Plusieurs pages s'ouvre

Dans les sémantèmes de causation en français, Kahane et Melcuk distinguent deux sens pour les verbes de causation :

1. être la cause de (non agentif) des exemples ?

2. être le causateur de (agentif)

Dans le cas de verbes de causation de type agentif[4], la structure qui en découle est la suivante:

- **causer2 (X,Y,[Z]) =>** « X cause Y[par Z] » : « [Vous (=X)]pouvez [utiliser le navigateur (=Z)][pour déplacer des titres (=Y)].»

Mais on trouve également des cas plus rares de l'emploi de causations non agentives. On le formule de cette façon :

- **causer1(X,Y,[Z])** => « X cause1 Y[par Z] » : « Les images contenues dans un tableau et dont la taille n'est pas indiquée [='X'] peuvent causer

[4] Les verbes de causation de type agentif sont les plus courants en français et dans notre corpus également

des problèmes d'affichage si la page parcourue utilise une ancienne norme HTML.[='Y'] »

Dans le cadre de notre travail, il est important d'avoir cette dichotomie en tête, puisque nous nous sommes focalisés sur les causations de type agentif, étant donné que les causations de type non agentif étaient quasiment inexistantes. Nous ne prétendons pas pouvoir extraire toutes les causations pour le moment. L'un des défauts majeurs de notre système actuel est d'ailleurs qu'il est difficile de comptabiliser le pourcentage de réussite d'extraction de causations. Pour cela, il faudrait prendre le temps de comptabiliser à la main toutes les causations du corpus, ce qui est assez difficile étant donné la taille de ce dernier. (299 966 mots). Nous pouvons cependant faire une estimation par l'intermédiaire d'un corpus de test.

Dans le cadre du présent mémoire, nous avons préféré nous focaliser sur la mise en place du système et sur sa description, plutôt que sur le pourcentage de causations extraites. Nous allons donc présenter ici les trois grands types de causation extraites par notre système.

i. Les causations dont le pivot est la conjonction de subordination « Pour »

Dans notre corpus, nous avons repéré de nombreuses causations ayant pour pivot central la conjonction de subordination « Pour ». Nous n'aborderons pas dans cette partie les causations en « pour », incluant une expression de la possibilité. Nous traitons ce cas dans un chapitre dédié à la modalité[5]. La structure syntaxique de ce type de causations est toujours sensiblement la même : une proposition principale possède un verbe, qui est la racine de la phrase, ayant dans ses dépendants directs le terme « pour », le terme « pour » ayant lui-même dans ses dépendants directs un verbe à l'infinitif.

[5] Partie III, sous-partie b, sous-partie vi

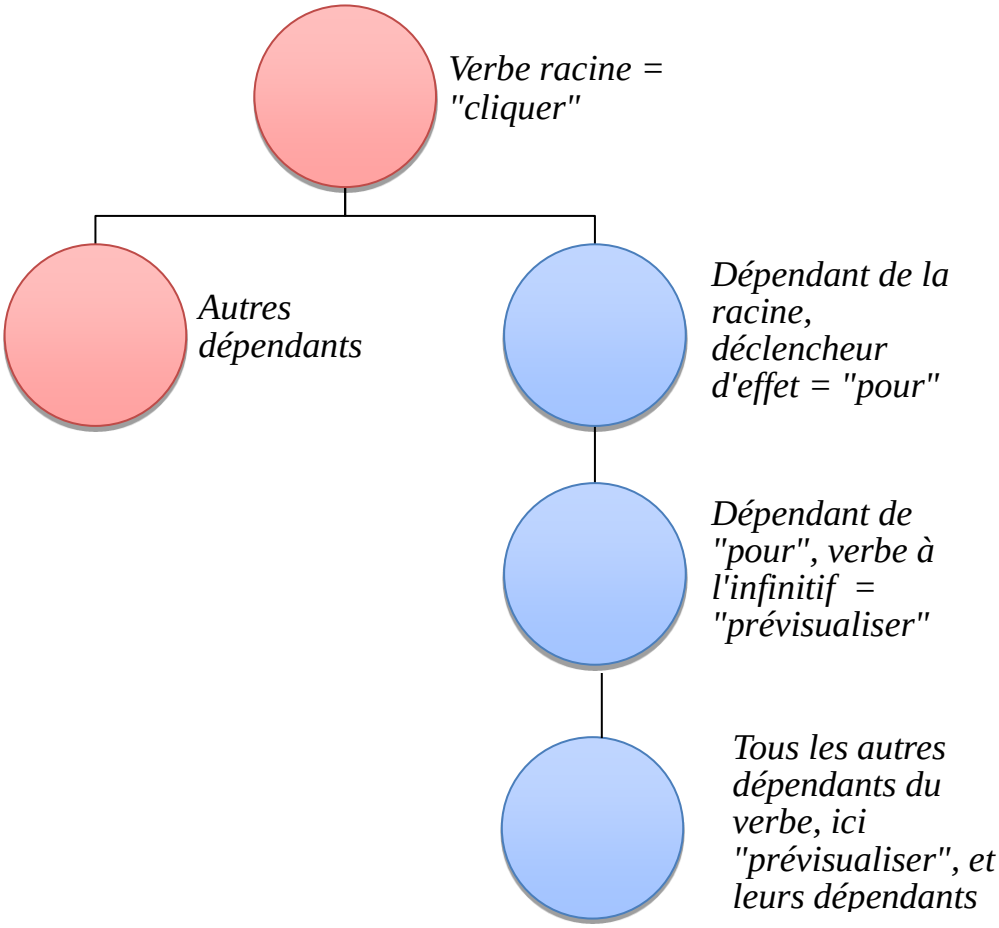


Figure 1 - En rouge LA CAUSE / En bleu L'EFFET

La proposition principale peut se situer avant ou après la proposition subordonnée :

1. **Avant la proposition subordonnée**

« Répétez les étapes 2 à 6 pour créer un deuxième Style de page personnalisé avec un en-tête différent. »

2. **Après la proposition subordonnée**

« Pour définir une bordure inférieure, cliquez plusieurs fois sur le bord inférieur jusqu'à ce qu'une ligne pleine apparaisse. »

Voici un exemple des questions-réponses qu'on souhaiterait générer pour ce type de causation :

- Que faire pour définir une bordure inférieure ?

- Cliquer plusieurs fois sur le bord inférieur jusqu'à ce qu'une ligne pleine apparaisse.

- Que faire pour créer un deuxième Style de page personnalisé avec un en-tête différent ?
  - Répéter les étapes 2 à 6

Ici, nous repérons déjà trois problèmes majeurs qu'il faudra traiter par la suite :

1. La résolution des anaphores[6]
2. La lemmatisation [7] des verbes
3. L'isolation des causations en « pour » sans inclusion des subordonnées en « pour » exprimant la destination, l'intérêt, la durée, le remplacement, le point de vue, la substitution etc.

La résolution des anaphores est expliquée plus loin[8]. Elle consiste à se servir de la structure Xml pour retrouver dans le texte qui précède l'anaphore, l'entité nommée à laquelle l'anaphore fait référence. Quant à la lemmatisation des verbes, elle est effectuée grâce à un dictionnaire des formes fléchies des verbes du français, prénommé le « lefff[9]». Ce dictionnaire permet, d'après une forme donnée d'un verbe, de retrouver sa forme canonique. En voici un extrait avec le verbe « cliquer » :

[6] Mot ou syntagme qui, dans un énoncé, assure une reprise sémantique d'un précédent segment appelé antécédent.

[7] La lemmatisation d'une forme d'un mot consiste à en prendre sa forme canonique. Celle-ci est définie comme suit : pour un verbe : ce verbe à l'infinitif, adjectif masculin, les autres ne l'ont pas. pour les autres mots : le mot au masculin singulier.

[8] Partie IV, sous-partie c

[9] Lexique des formes fléchies du français

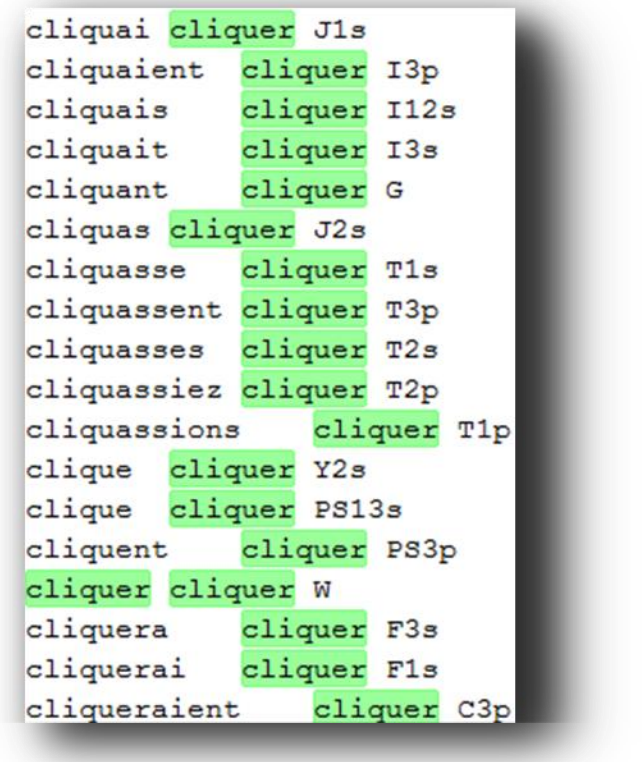


Figure 2 - Formes fléchies des verbes du français

Enfin, le problème du filtrage des subordonnées en « pour », dans le but de ne garder que les causations, est facile à résoudre. Pour nous en convaincre, prenons un contre-exemple :

- « OpenOffice Writer dispose d'une vaste gamme de modèles pour les principaux types de documents utilisés. »

Dans cette phrase, la proposition subordonnée en « pour » n'exprime pas un effet, mais est là pour marquer une destination. On peut ainsi paraphraser de la façon suivante :

- « OpenOffice dispose d'une vaste gamme de modèles. Cette vaste gamme de modèles a pour destination les principaux types de documents utilisés. »

Pour ne pas inclure toutes ces mauvaises propositions dans notre liste, il y a un moyen très simple. Dans toutes les subordinations en « pour » qui ne sont pas des causations, le dépendant direct de « pour » n'est pas un verbe à l'infinitif. Donc, si on ne garde que les phrases ayant un verbe principal dont l'un des dépendants directs est « pour », lui-même ayant dans ses dépendants directs un verbe à l'infinitif, on réussit à isoler toutes les causations en « pour ».

ii. Les causations dont le pivot est la conjonction de subordination « lorsque »

De même que pour les causations en « pour », les causations dont le pivot est la conjonction de subordination « lorsque », sont également nombreuses dans notre corpus. Il y a aussi des causations en « lorsque » incluant une possibilité, que nous traitons dans la partie dédiée à la modalité[10]. La structure syntaxique de ce type de causations est toujours sensiblement la même : une proposition principale possède un verbe, qui est la racine de la phrase, ayant dans ses dépendants directs le terme « lorsque », le terme « lorsque » ayant lui-même dans ses dépendants directs un verbe. Cependant, il faut être prudent, car contrairement aux causations en « pour », le **mécanisme cause/effet** est inversé. En effet, la cause est la proposition subordonnée introduite par « lorsque », tandis que l'effet escompté est la proposition principale.

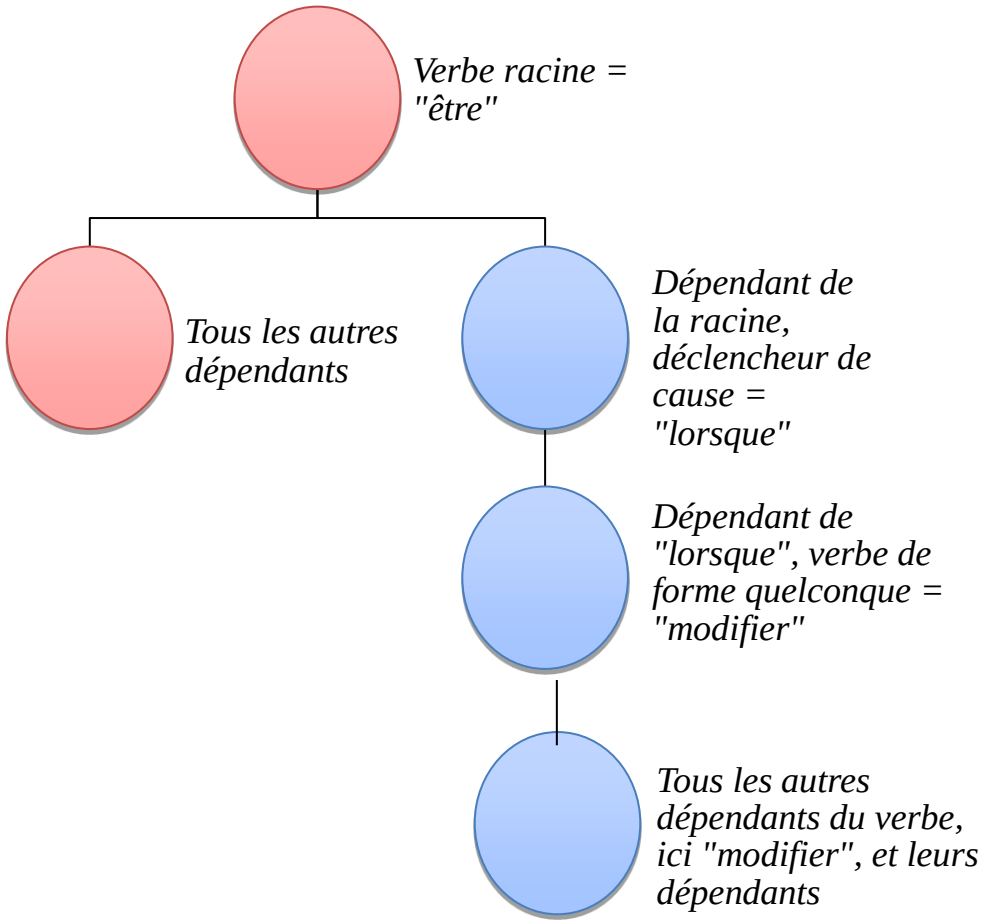


Figure 3 - En rouge L'EFFET / En bleu LA CAUSE

[10] Partie IV, sous-partie b, sous-partie vi

La proposition principale peut se situer avant ou après la proposition subordonnée :

1. Avant la proposition subordonnée

- « Certaines options de numérotation et de puces ne sont pas disponibles lorsque vous travaillez dans une mise en page Web. »

Les questions-réponses possibles sont les suivantes :

- Que se passe-t-il lorsque vous travaillez dans une mise en page Web ?
    - Certaines options de numérotation et de puces ne sont pas disponibles.
- A quel moment arrive-t-il que certaines options de numérotation et de puces ne soient pas disponibles ?
    - Lorsque vous travaillez dans une mise en page Web.

Dans ce cas précis et pour des raisons de temps, nous ne traiterons que le premier type de questions-réponses.

2. Après la proposition subordonnée

- « En effet, lorsque vous modifiez le format de numérotation d'un style, tous les paragraphes formatés avec ce style sont automatiquement actualisés. »
- Que se passe-t-il lorsque vous modifiez le format de numérotation d'un style ?
    - Tous les paragraphes formatés avec ce style sont automatiquement actualisés.
- A quel moment arrive-t-il que les paragraphes formatés avec un style soient actualisés ?
    - Lorsque vous modifiez le format de numérotation de ce style.

Dans le cas des causations en « lorsque », nous identifions les problèmes suivants à résoudre :

1. La résolution des anaphores
2. L'élimination des marqueurs de discours (propres à toutes les causations, mais clairement identifiés dans notre exemple par le marqueur « En effet »)

Comme mentionnée précédemment, la résolution des anaphores est traitée dans un autre chapitre[11]. Quant à l'élimination des marqueurs de discours, la solution trouvée pour le moment est de créer une liste de tous ces marqueurs, pour permettre ensuite au programme d'éliminer les potentiels marqueurs présents dans les causations alors extraites.

iii. Les causations dont le pivot est le gérondif

Le gérondif est la forme du participe présent. Il est, la plupart du temps, précédé de "**en**". Il exprime, par rapport au verbe principal, une action simultanée (par exemple, « Il marche en rêvant »). Dans notre corpus, nous trouvons de nombreuses causations ayant pour pivot cette forme verbale : dans ce cas, le fait d'exécuter une action produit un effet immédiat. Comme dans le cas des causations en « lorsque », la cause se situe dans la proposition subordonnée et l'effet dans la proposition principale.

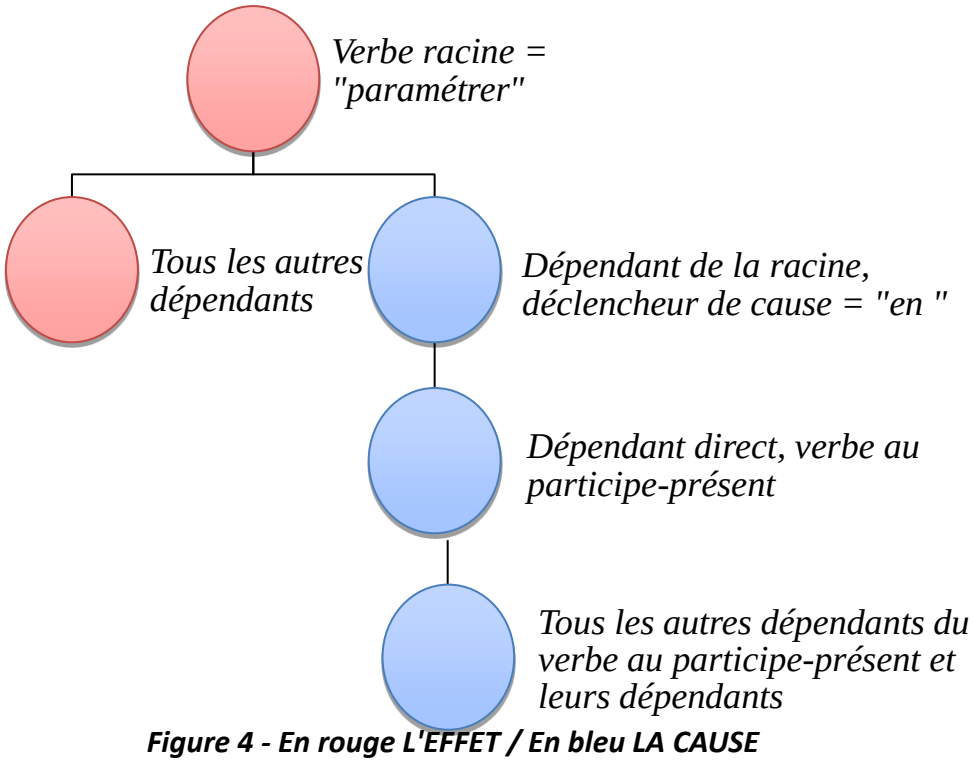


Figure 4 - En rouge L'EFFET / En bleu LA CAUSE

[11] Partie IV, sous-partie c

La proposition principale contient le verbe racine et peut se situer avant ou après la proposition subordonnée.

1. **Avant la proposition subordonnée**

- « Vous paramétrez cette propriété en définissant un autre style de page comme " style suivant " dans l' "onglet Format-Page-Gérer".»

2. **Après la proposition subordonnée**

- « Vous paramétrez cette propriété en définissant un autre style de page comme style suivant " dans l' "onglet Format-Page-Gérer".»

Les questions-réponses possibles sont les suivantes :

- Comment faire pour paramétrer cette propriété ?
    - En définissant un autre style de page comme "style suivant " dans l' "onglet Format-Page-Gérer".»
- Que se passe-t-il lorsque vous définissez un autre style de page comme " style suivant " dans l' "onglet Format-Page-Gérer".» ?
    - Vous paramétrez cette propriété.

Ici encore on identifie le problème de la résolution des anaphores[12], mais aussi le changement de la forme verbale du verbe de la proposition subordonnée pour en générer une question cohérente et la lemmatisation du verbe de la proposition principale.

iv. La confusion avec les conditionnelles

Il est important de ne pas confondre les causations avec les conditionnelles. En français et comme leur nom l'indique, les structures conditionnelles incluent une condition. Dans notre corpus, nous sommes confrontés la plupart du temps à des conditionnelles exprimant la potentialité. Par exemple :

- « Si la hauteur de cet élément est supérieure à la taille de police utilisée, la ligne contenant l'élément est agrandie en conséquence. »

Dans l'étude de notre corpus, nous avons pu remarquer que certaines phrases incluaient à la fois une conditionnelle et une causation. Dans ce cas,

[12] Partie IV, sous-partie c

nous sommes dans un triplet et il y a donc plus de possibilités de questions-réponses. Par exemple :

«**1**[Si vous êtes dans une mise en page Web], **2**[vous pouvez aussi utiliser le Navigateur] **3**[pour déplacer des titres et le texte associé vers le haut ou le bas du document.] »

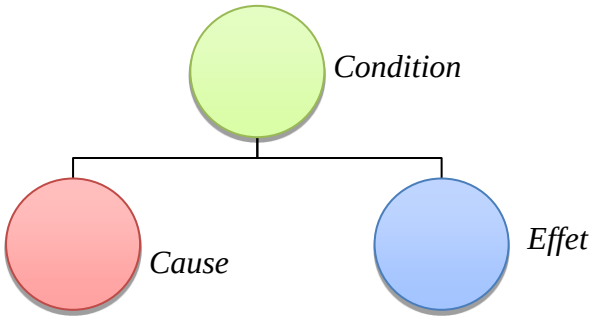


Figure 5 - Triplet avec conditionnelle incluant une structure de cause à effet

Pour des raisons de temps et de bonne délimitation du sujet, nous n'avons pas traité les conditionnelles dans le cadre de ce mémoire. Cependant, leur repérage et leur mise en exergue nous a permis de réaliser que dans le cadre d'un travail de thèse, il serait très judicieux de les inclure dans notre traitement.

v. Traitement de l'alternative

Le traitement des causations incluant une alternative a été effectué dans le présent travail. Nous pouvons donner l'exemple suivant :

- « Vous pouvez ensuite regrouper ou fusionner les données des adresses et les documents texte, soit pour imprimer les lettres, soit pour les envoyer par e-mail. »

Pour traiter ce type de causations, nous avons pris le parti de générer des questions-réponses du type de l'exemple suivant:

- « Que faire pour imprimer les lettres ou pour les envoyer par mails ? »
    - « Regrouper ou fusionner les données des adresses et les documents texte »

Il est vrai que nous aurions pu scinder la cause en deux, de manière à pouvoir générer deux paires de questions-réponses, mais nous avons pris le parti de laisser l'alternative intact, tout en réorganisant la structure syntaxique de manière à générer une question cohérente.

vi. Traitement de la modalité

Dans le traitement des causations, nous avons accordé une importance majeure aux causations incluant une possibilité, dans la mesure où elles étaient très nombreuses dans ce corpus. En linguistique, on définit la modalité comme « l'expression de l'attitude du locuteur par rapport au contenu propositionnel de son énoncé ». On trouve des modalités à valeur aléthique[13], épistémique[14], déontique[15] ou radicale[16]. Dans le cadre de notre corpus, nous traitons principalement les causations présentant une modalité à valeur déontique, puisqu'encore une fois, nous sommes dans le cadre d'une aide dans lequel un énonciateur fictif exprime la possibilité d'utiliser telle ou telle fonction de telle ou telle manière, en fonction de règles préétablies.

1. Les verbes de modalité

- Valeur déontique => l'énonciateur apprécie la relation prédicative positivement ou négativement en fonction de règles préétablies
    - Causations dont le pivot central est la conjonction de subordination « pour »
        - « Pour définir ce paramètre, ***vous pouvez*** également passer par l'onglet Adaptation du texte »

[13] Le sujet énonce des vérités logiques
[14] L'énonciateur considère les chances de réalisation de la relation prédicative.
[15] L'énonciateur apprécie la relation prédicative, positivement ou négativement, en fonction de règles pré-établie
[16] L'énonciateur autorise.

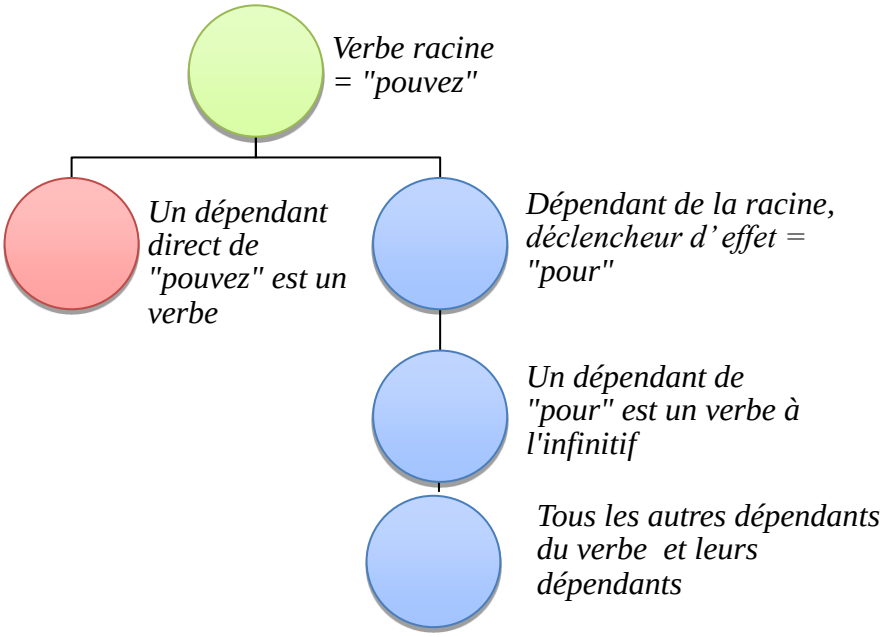


Figure 6 – En rouge LA CAUSE / En bleu L'EFFET / En vert, ELEMENT EXCLU

- Causations dont le pivot central est la conjonction de subordination « lorsque »
  - « ***Vous pouvez*** sélectionner un index personnalisé dans la zone " Type" lorsque vous insérez une entrée d'index dans le document. »

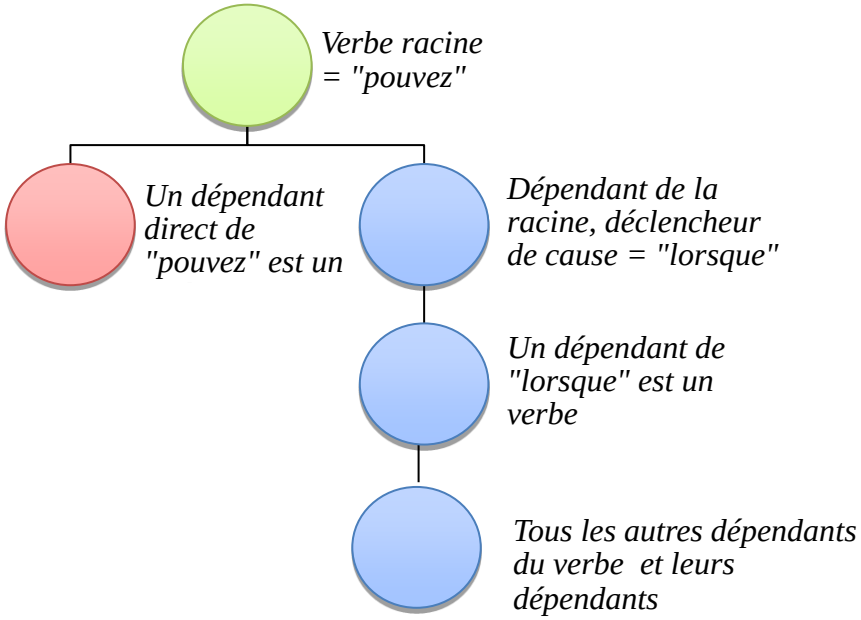


Figure 7 – En rouge L'EFFET / En bleu LA CAUSE / En vert, ELEMENT EXCLU

- Causations dont le pivot central est le gérondif
  - « ***Vous pouvez*** activer ou désactiver l'affichage du " Navigateur" en double-cliquant sur le "champ Numéro de page". »

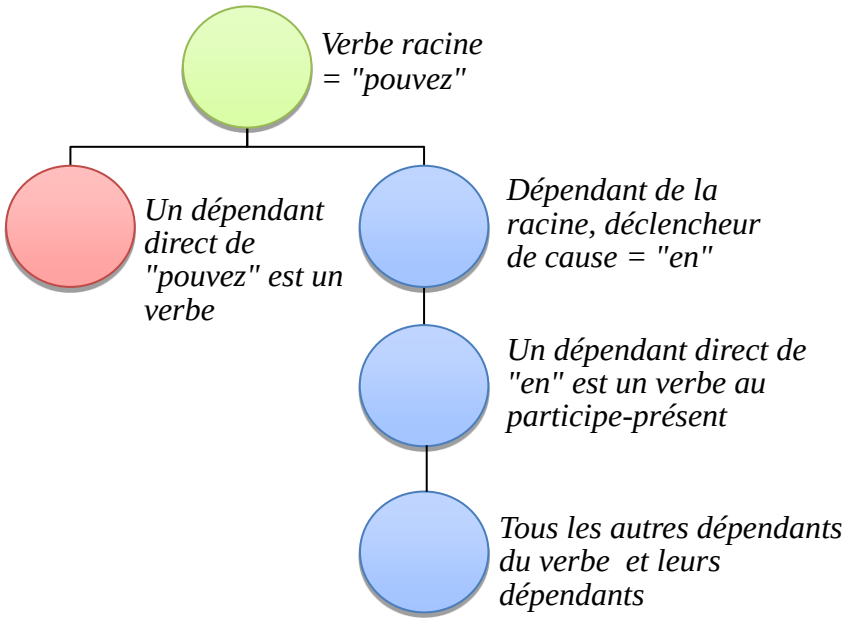


Figure 8 – En rouge L'EFFET / En bleu LA CAUSE / En vert, ELEMENT EXCLU

La structure syntaxique des causations à valeur déontique est donc sensiblement toujours la même dans notre corpus. Nous avons un nœud ayant pour fonction « modifieur » et se nommant « lorsque », « pour » ou « en ». Ce nœud possède un gouverneur qui est égal à « pouvez » et qui est la racine de l'arbre de dépendance. Il y a un dépendant du gouverneur « pouvez » qui est un verbe.

La valeur déontique implique parfois une autre alternative sous-entendue. Par exemple dans la phrase suivante : « Pour définir ce paramètre, ***vous pouvez*** également passer par l'onglet Adaptation du texte », l'énonciateur sous-entend qu'une autre alternative est possible pour définir « ce » paramètre. Dans le cadre du présent travail, nous avons pris le parti de générer à partir de ces causations modales, des questions-réponses conservant ce principe de modalité. Ainsi, voici la question-réponse construite à partir de la phrase précédente :

- « Que faire pour définir le paramètre X ? »

- « Une possibilité est de passer par l'onglet Adaptation du texte. »

Dans le cas précis de la modalité, nous nous sommes restreints à un seul type de questions-réponses par causation, contrairement aux précédentes structures présentées qui permettaient pour la plupart des cas de générer deux paires de questions-réponses. Cette restriction n'est pas anodine. Elle est due au fait que la question qui pourrait être générée à partir de l'autre pendant syntaxique, est assez alambiquée et n'a que peu de sens. Dans le cas de l'exemple précédent, nous aurions une question de type : « Pourquoi pouvez-vous passer par l'onglet Adaptation du texte ? ». Il y a clairement un problème de cohérence syntaxique et ce type de questions ne répond pas aux critères de clarté exigés dans un QCM.

À présent nous avons fait le tour de tous les types de causations extraites par notre système et nous avons expliqué pour chaque cas les questions-réponses générées. Dans le chapitre suivant, nous expliquons tout le cheminement nous ayant permis d'accéder à cette clarté et à cette extraction automatique.

IV. Prétraitements, analyse syntaxique du corpus et extraction des causations

a. Présentation générale

Désormais, le lecteur est en mesure de comprendre que l'analyse syntaxique en dépendances nous permet de naviguer à travers les structures causales de notre corpus. Le parseur syntaxique utilisé dans le présent travail se nomme « Mate Parser» et a été développé principalement par Bernd Bohnet. Il s'agit d'un projet Open Source et d'autres programmeurs sont intervenus dans le développement du logiciel. Cependant, avant de lancer l'analyse syntaxique du corpus, il a fallu effectuer divers prétraitements, le premier étant la création d'une ontologie et la résolution d'anaphores, ainsi que le passage d'un format Xml à un format Texte Brut. Les traitements informatiques ont été réalisés par le biais du langage de programmation Python.

b. Terminologie du monde OpenOffice

i. Qu'est-ce qu'une terminologie ?

C'est l'ensemble de tous les termes techniques spécifiques à une science, un art ou tout domaine en particulier. Dans le cadre de notre travail, nous avons besoin d'une terminologie du monde OpenOffice dans le but de pouvoir nous en resservir ensuite, pour résoudre les anaphores d'une part et pour isoler les termes spécifiques au monde OpenOffice d'autre part, afin que le parseur syntaxique les analyse tous comme des noms.

ii. Etapes d'extraction de notre terminologie

Les étapes nous ayant permis de créer notre terminologie sont au nombre de deux :

1. Extraction d'un grand nombre d'entités nommés grâce à la structure Xml

2. Extraction de tous les termes commençant par les mots clefs suivants :**"objet","icône","bouton","mode","touche","champ","barre","onglet", "boîte", "menu"**

Cette méthode nous a permis d'extraire un grand nombre d'entités nommés et nous obtenons d'assez bons résultats au niveau de l'application finale. Encore une fois, il est difficile d'évaluer à ce point de notre recherche la performance de notre système, puisque dans un corpus possédant environ 300 000 mots, le comptage à la main aurait pris énormément de temps. Nous réservons le calcul de la précision et du rappel à la suite de nos recherches.

En plus d'extraire ces termes grâce à un programme python, nous les identifions par des soulignements. Par exemple « base de données » devient «_base_de_données » et « langue » devient « _langue ». Ces soulignements sont très importants, puisqu'ils permettent au parseur de ne pas analyser ces entités comme plusieurs mots distincts. Si l'on donne au parseur « bases de données », il analysera, « bases », puis « de », puis « données », ce qui dans notre cas n'est pas souhaitable, puisque le groupe de mots « _base_de_données » a déjà une signification en tant que tel dans notre corpus et on risque d'obtenir des erreurs. Nous pouvons obtenir des mauvais rattachements (par exemple, un mauvais rattachement de « de données ») et inversement, nous risquons aussi de regrouper des termes qui ne devraient pas l'être (par exemple, « Sélectionnez

l'icône à côté » => l'icône_à_côté). Si au contraire on donne en entrée du parseur « _bases_de_données », il analysera cette entité comme un nom unique. Le soulignement présent en début de chaque entité nommée est voulu : il permet d'identifier qu'il s'agit bien d'une entité nommée et pas d'autre chose. Ainsi, dès que le programme rencontre un terme commençant par un soulignement, il sait qu'il est confronté à une entité nommée.

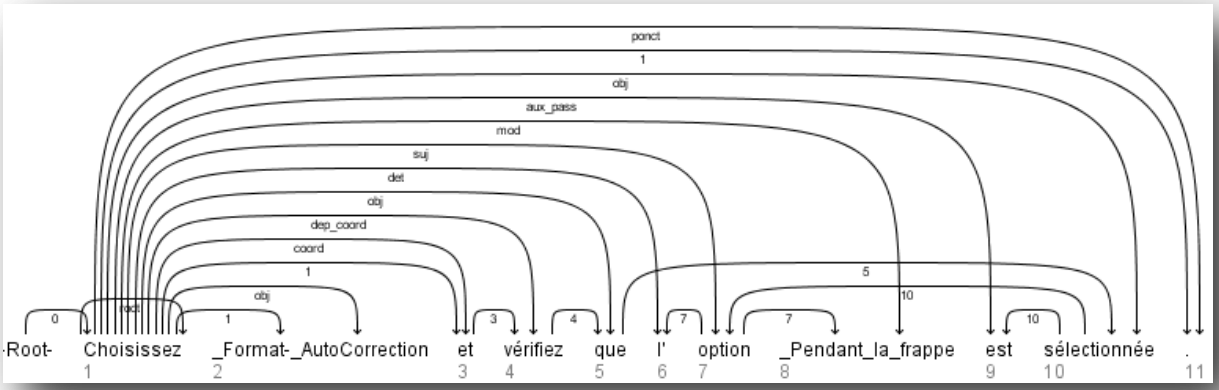


Figure 1 - Les termes spécifiques au monde OpenOffice sont bien analysés comme des noms uniques par le parseur

c. Résolution partielle des anaphores et des déictiques

La résolution des anaphores[17] et des déictiques a été une étape importante pour la bonne compréhension des questions-réponses générées par notre système. Avant de nous lancer dans une quelconque programmation, encore fallait-il analyser linguistiquement, quels types d'anaphores et de déictiques étaient le plus représentés dans notre corpus. Dans un premier temps il est important de bien percevoir la différence entre les deux concepts. Alors qu'une anaphore est un mot ou un syntagme, qui assure une reprise sémantique d'un antécédent, le déictique fait référence à la situation d'énonciation. Il s'appuie sur les paramètres du lieu, du temps ou de la personne.

i. Isolation des anaphores

Les phrases contenant des anaphores ne sont pas utilisables directement dans notre approche. Dans le but de les inclure, on se base sur certaines heuristiques

[17] Mot ou syntagme qui, dans un énoncé, assure une reprise sémantique d'un précédent segment appelé antécédent

pour résoudre des anaphores fréquentes dans le corpus. Cela est cependant une approche « ad hoc », qui génère aussi quelques erreurs.

Nous avons isolé les anaphores suivantes :

- Pronoms Démonstratifs « ce », « cet », « cette », « ces », « celui-ci », « celui-là »
    - « Utilisez ***cette*** combinaison pour ancrer et détacher rapidement le Navigateur , la fenêtre Styles et formatage ou toute autre fenêtre »
- Pronoms Personnels « il », « elle », « ils », « elles »
    - « ***Elle*** s' affiche lorsque vous placez le curseur dans un tableau . »
- Pronoms Possessifs « son », « sa », « ses »
    - « Lorsque vous ajoutez une légende à un tableau**, *son*** texte est inséré sous forme de paragraphe en regard du _tableau . »
- Phrases démarrant par un verbe à la 3ème personne du singulier
    - « Permet de saisir un URL ou de l'insérer par glisser-déposer à partir d'un document »

Notez que dans ce cas précis l'anaphore est inexistante. Il y a cependant tout de même une phrase isolée dans laquelle il manque un terme sans lequel la bonne compréhension de cette phrase est impossible.

ii. Isolation des déictiques

- Prépositions Locatives « ici », « ci-après », « suivante », « précédente », « dessus », « ci-dessous » etc.
    - « Vous pouvez définir les conditions des styles ***ici***. »
    - « Suivez les instructions suivantes lorsque vous créez un fichier de concordance : »

Ici encore, il est important de préciser que nous n'avons pas résolu toutes les anaphores et tous les déictiques, l'essentiel étant surtout de résoudre les anaphores fréquemment présentes dans les causations. Pour parvenir à cette résolution partielle, nous avons utilisé la structure du balisage Xml et nous avons navigué dans cet arbre pour aller chercher, de façon automatisée les termes représentant l'anaphore ou le déictique. De façon plus schématisée, voici

un exemple de ce que nous avons réalisé, ici avec la résolution du pronom démonstratif **« cette »** :

<paragraph role="heading" id="hd_id3149973" xml-lang="fr" level="1" l10n="CHG" oldref="58"><variable id="arrange_chapters"><link href="text/swriter/guide/arrange_chapters.xhp" name="Réorganisation d'un document à l'aide du Navigateur">Organisation de chapitres dans le Navigateur</link></variable></paragraph>

<paragraph role="paragraph" id="par_id3147795" xml-lang="fr" l10n="U" oldref="59">Vous pouvez utiliser le Navigateur pour déplacer des titres et le texte associé vers le haut ou le bas du document. Vous pouvez également hausser ou abaisser les niveaux de titre. Pour utiliser cette fonction, formatez les titres du document avec l'un des styles de paragraphe "Titre" prédéfinis. (…) </paragraph>

Dans le cas ci-dessus, nous remarquons que l'anaphore « cette » fait référence à un élément de type « heading », présent dans le précédent paragraphe. La manipulation consiste donc à créer une fonction qui nous permette de récupérer toutes les anaphores entre balises <paragraph role="**paragraph**"> et d'aller chercher dans le texte qui précède l'élément le plus proche entre balises <paragraph role="**heading**">. Il faut cependant faire attention à plusieurs choses :

- Cette méthode ne fonctionne pas pour toutes les anaphores précédemment répertoriées, mais uniquement sur les anaphores qui sont des pronoms démonstratifs : « ce », « cette », cet », « ces », « celui-ci »
- Même en restant dans ce cadre restreint, il arrive de temps en temps (assez rarement tout de même) que l'antécédent ne soit pas le segment compris entre balises <paragraph role="**heading**">. Par exemple, dans quelques phrases, nous avons répertorié un antécédent se trouvant dans la phrase précédente ou au sein de la même phrase.
    a. « Lorsque vous insérez ou supprimez des cellules, des lignes ou des colonnes dans un " tableau", les répercussions qu'ont ces opérations sur les éléments adjacents sont déterminées par les options "Comportement lors du déplacement" définies.»

Dans le cas précis d'une phrase comme cette dernière, notre système sera incapable d'en déduire que le terme reprenant l'anaphore est « l'insertion, la suppression des cellules, des lignes ou des colonnes dans un « tableau ».

Notre méthode permet donc une amélioration de la compréhension des causations incluant des anaphores de type « pronom démonstratif ». Cependant, cette méthode possède plusieurs défauts :

- Cette approche est idiosyncratique à notre problème et ne s'appliquerait pas à d'autres corpus comparables s'ils ne sont pas exactement de la même structure. Elle n'est pas universelle, puisqu'un autre tutoriel n'aura jamais le même arbre Xml et n'aura d'ailleurs très certainement aucun balisage de structuration !
- Elle ne permet de résoudre qu'une portion de toutes les anaphores répertoriées.

d. Parsing syntaxique

L'analyse syntaxique se déroule en quatre étapes principales :

1. **Tokenization**

C’ est l'action de séparer une phrase en mots. La tokenisation produit dans le corpus en sortie tous les fichiers nommés xxx-one-word-per-line.txt. En regardant la structure du fichier on observe que le parseur, pour chaque phrase, a placé un mot par ligne et qu'il a numéroté ces mots.

2. **Lemmatisation**

C’est l'action de trouver les lemmes des mots précédemment tokenisés. La lemmatisation produit dans le corpus en sortie tous les fichiers nommés xxx--lemmatized.txt. La lemmatisation d'une forme d'un mot consiste à en prendre sa forme canonique ; pour un verbe il s’agit du verbe à l’infinitif et pour les autres mots, il s’agit du mot au masculin singulier.

Il faut faire attention, car la sortie lemmatisée de notre corpus présente pour le moment encore de nombreuses erreurs et c’est d’ailleurs pour cette raison précise que nous utilisons le « leffe[18] » dans l’état actuel des choses.

3. **Tagging ou marquage syntaxique**

C’est l'action de déterminer la catégorie syntaxique d'un mot. Le marquage est défini comme suit : « V » pour verbe, « P » pour préposition, « D » pour déterminant, « N » pour nom, « C » pour coordination, « PONCT » pour ponctuation. Le tagging produit dans le corpus en sortie tous les fichiers nommés xxx-tagged.txt.

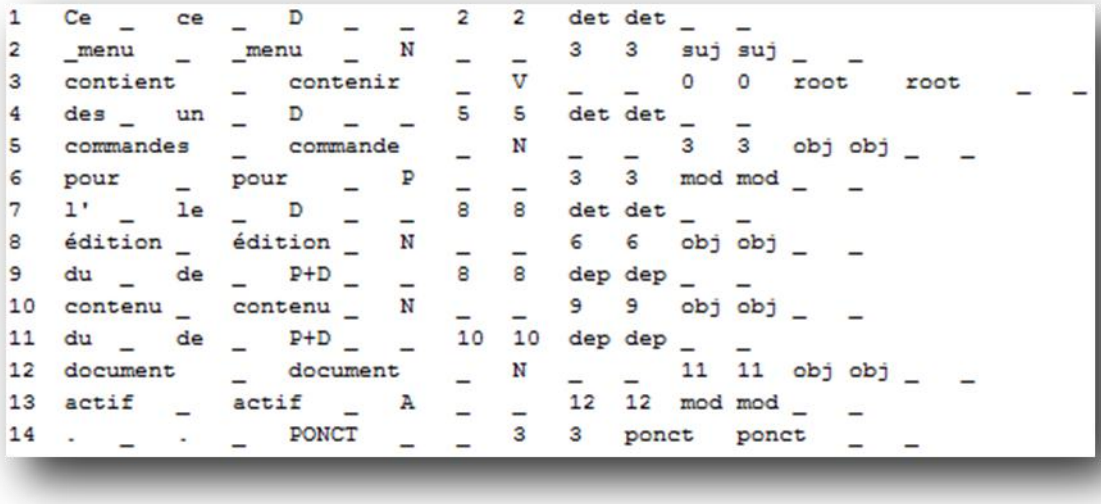
4. **Parsing syntaxique en dépendance**

C’est l'action de déterminer quels liens syntaxiques entretiennent les mots entre eux. Le parsing produit dans le corpus en sortie tous les fichiers nommés xxx-dependently.txt. Ce parsing syntaxique est au format CONLL, d'où le nom du programme permettant d'analyser ce corpus pour en extraire les liens de cause à effet.

[18] Dictionnaire des formes fléchies du français, décrit à la page 28

e. Le format CONLL

- **Phrase exemple au format CONLL**



1	Ce	_	ce	_	D	_	_	2	2	det	det	_	_
2	_menu	_	_menu	_	N	_	_	3	3	suj	suj	_	_
3	contient	_	contenir	_	V	_	_	0	0	root	root	_	_
4	des	_	un	_	D	_	_	5	5	det	det	_	_
5	commandes	_	commande	_	N	_	_	3	3	obj	obj	_	_
6	pour	_	pour	_	P	_	_	3	3	mod	mod	_	_
7	l'	_	le	_	D	_	_	8	8	det	det	_	_
8	édition	_	édition	_	N	_	_	6	6	obj	obj	_	_
9	du	_	de	_	P+D	_	_	8	8	dep	dep	_	_
10	contenu	_	contenu	_	N	_	_	9	9	obj	obj	_	_
11	du	_	de	_	P+D	_	_	10	10	dep	dep	_	_
12	document	_	document	_	N	_	_	11	11	obj	obj	_	_
13	actif	_	actif	_	A	_	_	12	12	mod	mod	_	_
14	.	_	.	_	PONCT	_	_	3	3	ponct	ponct	_	_

Figure 2 - Phrase au format CONLL

C'est un format comportant entre 10 et 14 champs séparés par des tabulations. Le format en question adhère à plusieurs règles :

1. Les fichiers CONLL comportent des phrases séparées par des lignes blanches.

2. Une phrase est faite de un ou plusieurs tokens[19], chacun commençant sur une nouvelle ligne.

3. Un token est composé de 10 champs : ces champs sont séparés par une tabulation.

4. Les fichiers sont encodés en UTF-8.

5. Les champs sont les suivants : ID, FORM, CPOSTAG, POSTAG, HEAD and DEPREL. En ce qui nous concerne, nous les nommons ID, WORD, LEMMA, TAG, HEAD. De plus, dans nos fichiers, les champs « head » et « rel » sont générés deux fois. Les champs 9 et 10 sont tous deux les « head » et les champs 11 et 12 sont tous deux les « rel ».

[19] Suite de caractères entre des espaces, item

Le tableau ci-dessous décrit de manière précise les champs de nos fichiers :

ID	Compteur de mots commençant à 1 pour chaque nouvelle phrase
WORD	Mot forme ou symbole de ponctuation
LEMMA[20]	Lemme du mot forme
TAG	Partie du discours, c'est-à-dire, catégorie syntaxique du mot
HEAD	Tête du mot courant, qui est soit une valeur d'un ID, soit 0 si le mot courant est la racine syntaxique de la phrase.
REL	Relation dépendante à la tête. L'ensemble des relations de dépendance dépend de la langue utilisée. Dans nos fichiers, la relation de dépendance peut être diverse :'root', 'obj','suj','mod','coord','det','dep', 'de_obj','dep_coord'.

i. Grammaire syntaxique utilisé par notre parseur

Tab 1. - Récapitulatif des fonctions syntaxiques utilisées dans notre grammaire syntaxique par le parseur Mate Parser

Dénomination	Fonction	Exemple
mod (epith)	Epithète du nom	Adjectif - Ex : un document *riche* Complément du nom introduit par une préposition - Ex : les titres *du document*
mod (attr)	Attribut du sujet de la copule Pour le différencier de l'adjectif, on a coutume de dire qu'il fait corps avec le verbe principal, qu'il sert à marquer le rôle verbal.	Attr. Adjectival - Ex : Le titre est *souligné.* Attr. Substantival - Ex : Les borgnes sont *rois.* Attr. Adverbial - Ex : Les sections sont *ensemble.* Attr. Introduit Par Une Préposition - Ex : Il reste *de glace.*

[20] Forme canonique d'un mot, comme l'infinitif pour un verbe

mod (adv)	Complément circonstanciel de l'adjectif ou de l'adverbe	Adverbe - Ex : un *très* joli titre Complément non adverbial, introduit par une préposition - Ex : un document riche *en commentaires*
mod (adv)	Complément circonstanciel du verbe	Adverbe – Ex : Vous pouvez *également* choisir cette option. Locatif circonstanciel – Ex : *Sur la barre d'outils*, il y a de nombreuses options. Autres compléments circonstanciels introduits par une préposition (but, manière, temps etc.) Complément circonstanciel sans préposition – Ex : Il fera ses devoirs *demain.*
coord	Fonction permettant de lier la conjonction au 2nd constituant	Ex : Elle clique sur le titre *et* le texte.
dep_coord	Fonction permettant de lier une conjonction à une préposition	Ex : et *de* définir
det	Déterminant du nom	Ex : *Les* titres, *Le* document
obj	Objet direct du verbe	Substantif - Ex : Jean utilise *cette fonction.* Clitique - Ex : Jean *l'*utilise.
de_obj	Déterminant de l'objet Préposition précédant l'objet	Déterminant - Ex : insérer *des* éléments Préposition - Ex : permettant *de* manipuler
suj	Sujet du verbe fini	Ex : *Vous* pouvez cliquer.
dep	Complément d'une préposition	Substantif – Ex : Après *cette manipulation,* vous pouvez tester le publipostage. Verbe – Ex : Vous pouvez utiliser

		le navigateur pour *déplacer des titres.*

Pour en savoir plus sur le format CONLL, il est également possible de lire l'explication de (Joakim Nivre, 2007).

ii. Lecture du format CONLL

Basons-nous sur la phrase d'exemple précédemment illustrée : **« Ce menu contient des commandes pour l'édition du contenu du document actif. »**

A présent, voici comment procéder pour lire l'arbre syntaxique de cette phrase :

- La racine de la phrase se nomme "root" et est numérotée "0". C'est le verbe « contenir », puisqu'il ne dépend de rien syntaxiquement ! C'est lui qui « commande » la phrase. C'est le verbe principal sans lequel aucune structure n'est possible. C'est pour cela qu'on l'appelle « racine ».
- L'ID de ce verbe est égal à « 3 » puisqu'il arrive en 3ème position dans la phrase, donc pour retrouver tous ses dépendants syntaxiques, l'exercice consiste à scanner la colonne des dépendants et à s'arrêter sur tous les tokens possédant un « 3 » dans cette colonne. Dans notre exemple, on trouve « menu », « commande » et « pour ».
- On recommence ensuite cet exercice pour trouver les dépendants syntaxiques de « menu », puis de « commande », puis de « pour » : l'ID de « menu » est égal à « 2 », donc on scanne la colonne des dépendants pour retrouver tous ses dépendants syntaxiques. On trouve le token « Ce », qui dépend donc syntaxiquement de « menu ». On réitère l'exercice jusqu'à la reconstruction totale de l'arbre syntaxique.

Illustration du résultat :

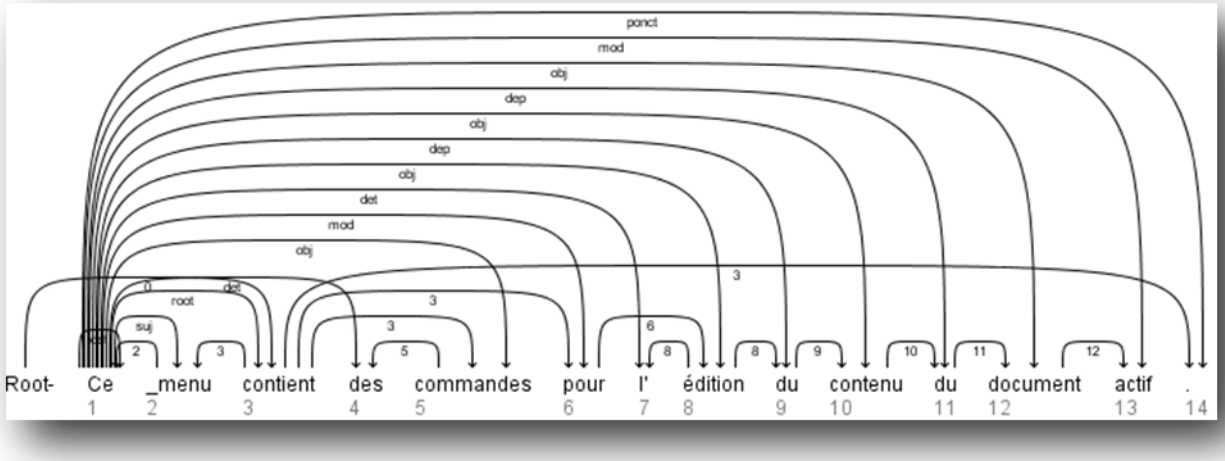


Figure 3 - Format CONLL illustré

f. Extraction automatique des causations

Après avoir parsé syntaxiquement tout notre corpus, nous nous sommes attelés à l'adaptation d'un début de script Python écrit par Kim Gerdes. Le programme nous permettant cette extraction se nomme *conll.py* et la fonction principale de navigation se nomme *conllfile2tree*. Le but du programme est d'extraire de façon automatique un maximum des relations de cause à effet présentes dans notre corpus. Le programme prend donc en entrée le corpus parsé syntaxiquement grâce à une boucle qui passe sur chaque fichier, puis il crée un fichier de sortie intitulé *causations.txt*, dans lequel sont répertoriés toutes les causations de cause à effet extraites, avec séparation entre la cause et l'effet, ainsi qu'annotation du type de causation.

Décrivons maintenant de manière plus précise les principales fonctions de ce programme :

- **def conllfile2tree(path)**
    - Permet de naviguer dans le fichier CONLL et d'en extraire une liste de dictionnaires, d'abord de chaque arbre et ensuite chacun de ces dictionnaires contient un dictionnaire de chaque token. Ces dictionnaires, mis bout à bout dans la liste, sont de la forme suivante :

        - {'id':nr,'word': word,'lemma': lemma, 'tag': tag, 'head':head, 'rel': rel,'deps':nodedic.get(int(nr),{}).get('deps',[])}

    - Chaque nœud a donc un dictionnaire avec les entrées suivantes :

        - Id
        - Word
        - Lemma
        - Tag
        - Head
        - Rel
        - Deps

    - On retrouve donc la structure du fichier CONLL, mais mis sous forme de dictionnaire, permettant un accès direct aux entrées lors du traitement informatique. La dernière case du dictionnaire intitulée « deps » permet de retrouver tous les dépendants de chaque token, et cela pour chaque dictionnaire.

- **def getDependents(nodedict,nodenum,deplist=[],cutat=[],cutpoints=[])**

    - Permet d'extraire les dépendants de chaque token et retourne une liste de dépendants. Elle est utilisée dans la fonction « conllfile2tree(path) » : c'est en effet cette fonction qui permet d'extraire la liste de dépendants pour chaque token.

- **def lefff**

    - Permet de rechercher dans le leffe la forme canonique et de remplacer dans la causation le verbe conjugué par sa forme canonique

- **def CauseaEffet**

    - Permet d’extraire toutes les causations décrites dans le chapitre intitulé « Extraction de structures causales »

        - **Causations en « pour »**

            - Simples

            - Avec modalité déontique : verbe « pouvez »

        - Avec alternative « soit », « soit »
    - **Causations en « lorsque »**
        - Simples
        - Avec modalité déontique : verbe « pouvez »
        - Avec alternative « soit », « soit »
    - **Causations en « en » + participe-présent**
        - Simples
        - Avec modalité déontique : verbe « pouvez »
        - Avec alternative « soit », « soit »

- **def nettoie**
    - Permet de nettoyer les phrases du corpus parsé : remplacement des soulignements par des guillemets, retrait des espaces mal placés
- **def sortirCauseEffet**
    - Permet d'écrire dans notre fichier de sortie causations.txt, chaque structure de cause à effet extraite selon le quadruplet suivant :
        - Ligne séparatrice
            - Nom du fichier dans lequel se trouve la causation
            - Type de causation
            - Cause
            - Effet

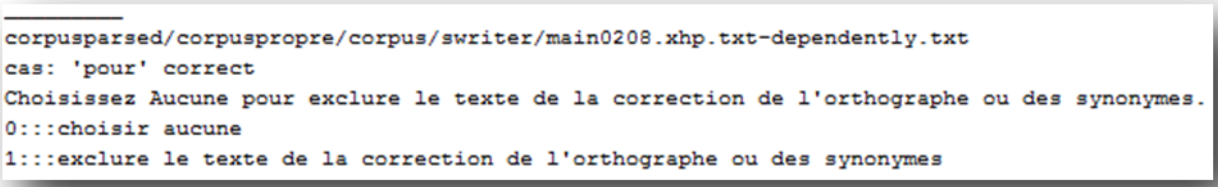


Figure 4 – Format de sortie du fichier de causations

Il est à préciser que dans la fonction « CauseaEffet », la recherche qui s'effectue dans les différentes phrases du corpus parsé, est effectuée grâce à la fonction « conllfile2tree ». Par exemple, si nous prenons le cas des causations en « pour » à modalité déontique, nous recherchons dans les structures CONLL de chaque phrase, un nœud courant égal à « pour » de fonction « mod » ayant dans ses dépendants directs un verbe à l' infinitif , dont le gouverneur est égal à « pouvez » et à la fonction « root », ce gouverneur ayant lui-même dans ses dépendants un verbe. La fonction nous permet de faire une telle extraction puisque nous avons accès aux liens syntaxiques entre les mots. C' est une sorte de grammaire de graphes dans laquelle nous reconnaissons certains graphes.

Il a également fallu faire face aux erreurs d'analyse syntaxique du parseur, qui n'est pas fiable à 100%. Ainsi, concernant les causations en « pour » et en « lorsque » à modalité déontique, nous avons recherché deux arbres syntaxiques différents pour chaque type de causation. Par exemple, le second type d'arbre principal des causations en « pour » à modalité déontique n'est pas syntaxiquement cohérent, mais dans le but d'extraire un maximum de causations, nous avons malgré tout du gérer ce cas. Il correspond à un nœud courant égal à « pouvez » ayant la fonction « dep », dont le gouverneur est égal à « vous » et à la fonction « suj » ou « aff », dont le grand-père a la fonction « root », le grand-père ayant dans ses dépendants le terme « pour ».

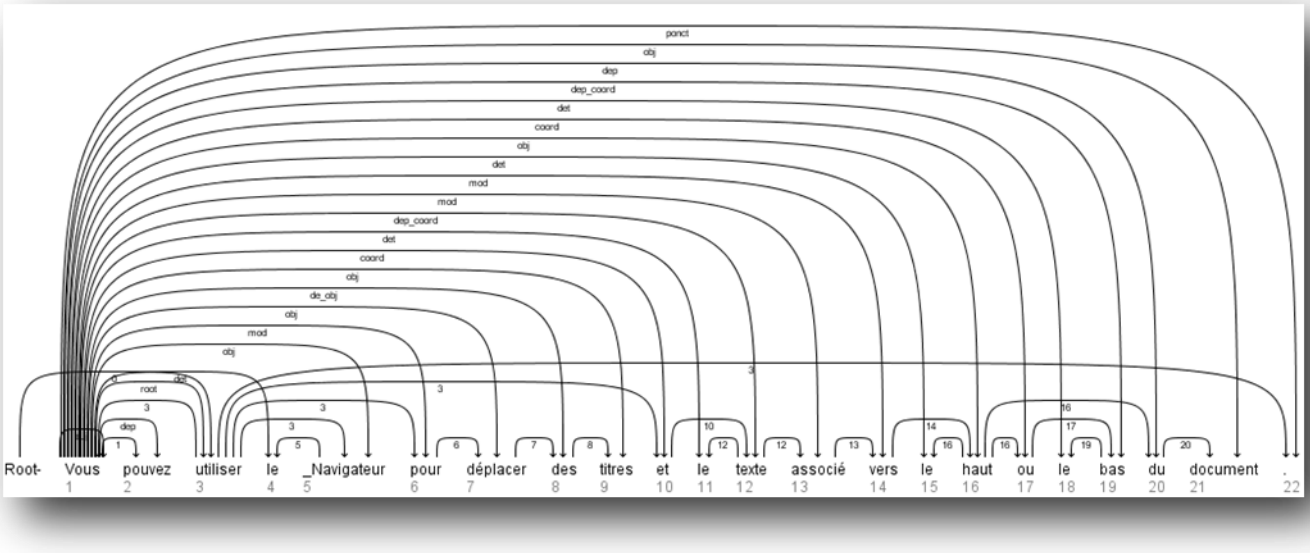


Figure 5 - Exemple d'une mauvaise analyse syntaxique générée par le parseur

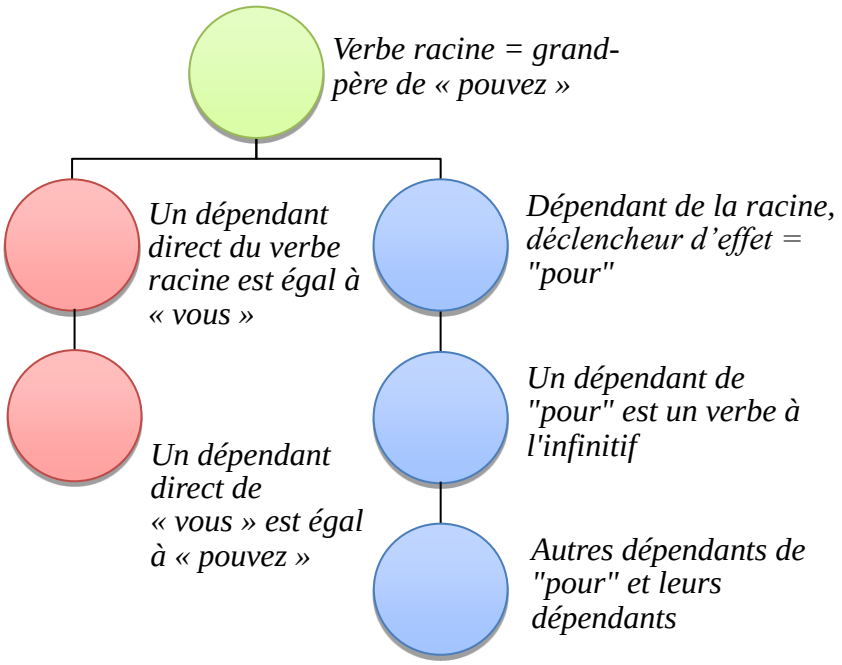


Figure 6 - Extraction des mauvaises analyses modales du parseur syntaxique
(pour la bonne analyse, se reporter à Partie IV, sous-partie b, sous-partie vi)

V. Extraction automatique des questions-réponses

Maintenant que nous avons extrait les liens de cause à effet précédemment décrits, nous aimerions pouvoir les utiliser pour créer des questions-réponses. L'entrée correspond donc à nos phrases exprimant un lien de causalité, dont la cause est isolée de l'effet.

```
corpusparsed/corpuspropre/corpus/swriter/main0208.xhp.txt-dependently.txt
cas: 'pour' correct
Choisissez Aucune pour exclure le texte de la correction de l'orthographe ou des synonymes.
0:::choisir aucune
1:::exclure le texte de la correction de l'orthographe ou des synonymes
```

Figure 1 – Format de sortie du fichier de causations, causations en «pour»

```
corpusparsed/corpuspropre/corpus/swriter/01/04090003.xhp.txt-dependently.txt
cas: 'lorsque' correct
Lorsque vous cliquez sur un "champ" substituant dans le document, il vous est demandé d'insérer l'élément manquant.
0:::cliquez sur un "champ" substituant dans le document
1:::il vous est demandé d'insérer l'élément manquant.
```

Figure 2 – Format de sortie du fichier de causations, causations en «lorsque»

```
corpusparsed/corpuspropre/corpus/swriter/guide/finding.xhp.txt-dependently.txt
cas: 'participepresent' correct
En donnant la valeur 1 aux trois nombres, vous obtenez des résultats satisfaisants.
0:::donner la valeur 1 aux trois nombres
1:::obtenir des résultats satisfaisants
```

Figure 3 – Format de sortie du fichier de causations, causations en «participe-présent»

Il nous faut donc pouvoir associer à chaque type de causation une interrogative et une réponse à cette interrogative. Nous avons donc élaboré un programme spécialement pour effectuer cette manipulation et nous l'avons intitulé *qcmtofile.py*. Plus qu'un simple programme de génération de questions et de réponses à partir de causations, il permet de générer des QCM complets de dix questions, chaque question incluant dans ses choix de réponse, la bonne

réponse, ainsi que trois réponses fausses ayant une similarité syntaxique avec la bonne réponse, mais générées aléatoirement.

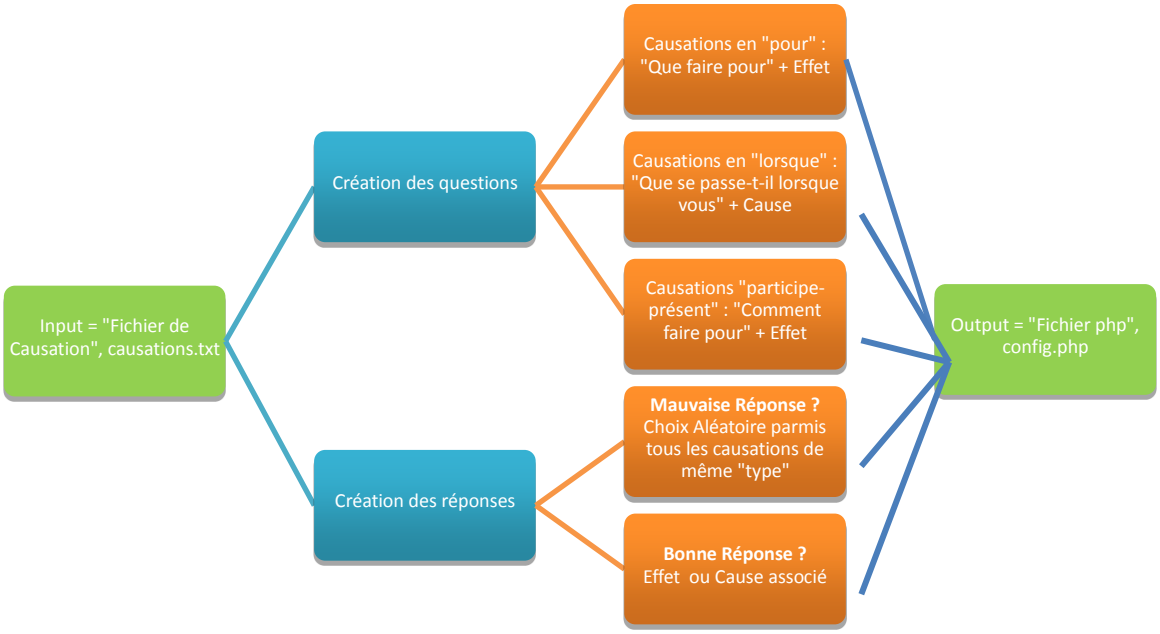
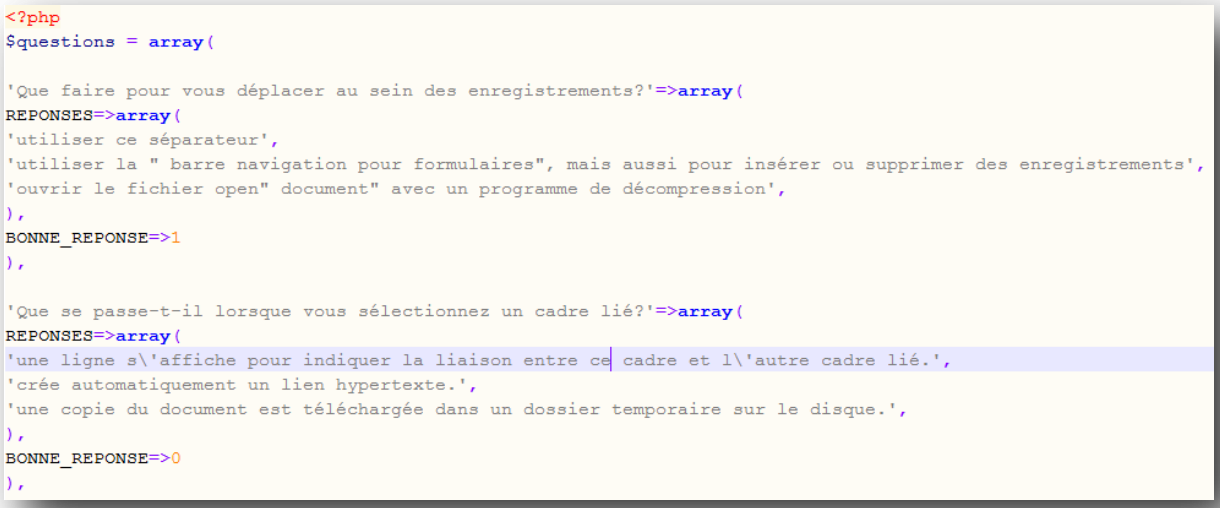


Figure 4 - Explication du fichier de création de questions-réponses, "qcmtofile.py"

Il est important de spécifier que la sortie de ce fichier est générée en langage php, fichier "config.php", ce qui permettra à notre application d'être visualisable depuis un navigateur web. Ainsi, ce système n'est pas vain. Chaque corpus parsé et analysé par notre système pourrait ainsi générer ensuite automatiquement des qcm. Un apprenant lambda, suivant un cursus de bureautique en particulier, pourrait donc valider ou non ses connaissances et avoir une estimation de son niveau, orientant la suite de son apprentissage vers un niveau adapté à ses compétences. Par ailleurs, un tel système pourrait également être utilisé par un organisme de délivrance de certifications. L'utilisateur passerait le test à l'issu duquel une certification lui serait ou non délivrée. Les interrogatives, ainsi que les réponses associées à tel ou tel type de causations sont décrites dans le chapitre intitulé « Extraction de structures causales ».



```php
<?php
$questions = array(

'Que faire pour vous déplacer au sein des enregistrements?'=>array(
REPONSES=>array(
'utiliser ce séparateur',
'utiliser la " barre navigation pour formulaires", mais aussi pour insérer ou supprimer des enregistrements',
'ouvrir le fichier open" document" avec un programme de décompression',
),
BONNE_REPONSE=>1
),

'Que se passe-t-il lorsque vous sélectionnez un cadre lié?'=>array(
REPONSES=>array(
'une ligne s\'affiche pour indiquer la liaison entre ce cadre et l\'autre cadre lié.',
'crée automatiquement un lien hypertexte.',
'une copie du document est téléchargée dans un dossier temporaire sur le disque.',
),
BONNE_REPONSE=>0
),
```

Figure 5 - Extrait des questions-réponses dans l'output en php, "config.php"

VI. Mise en place de l'application finale

a. Création d'une application web

Une fois le processus de création de qcm terminé, nous avons mis en place une application web, permettant à un utilisateur de pouvoir générer un qcm et soumettre ses réponses en vue d'une évaluation. A chaque consultation de l'application, le qcm est généré à nouveau à partir du fichier *"qcmtofile.py"*, dans le but de créer un nouveau questionnaire avec de nouvelles questions-réponses.

L'utilisateur se connecte sur la page d'accueil de l'application. Cette page est constituée d'un script principal écrit en php, appelé *index.php*, exécutant les opérations suivantes :

- Création du fichier de configuration du qcm, appelé *config.php*, généré à partir du fichier d'extraction de qcm par le script *qcmtofile.py.* Ce fichier est exécuté par un appel de type « web service » par le script principal.
- Inclusion de ce fichier *config.php*, dans le script principal de l'application web. Ce fichier *config.php* contient en réalité un tableau PHP reprenant la liste des questions, la bonne réponse et des mauvaises réponses.
- Lorsque l'utilisateur soumet son questionnaire rempli, le script calcule le score en se référant au fichier *config.php.* Pour l'exemple, la certification est obtenue si l'utilisateur obtient un minimum de 70% de réponses correctes.

En outre, le script principal inclut le fichier *generate_ppt.php*. Ce fichier permet la *c*réation automatique d'un document MS Powerpoint, chaque diapositive de ce document correspondant à une question-réponse de notre qcm. Il est à noter que cela fait appel à une librairie open source présente dans le dossier *quiz*, dans le sous-dossier intitulé *ppt*. Dans cette librairie, une classe, qui se nomme *PHPPowerpoint* permet de générer un PowerPoint à partir d'un code php .

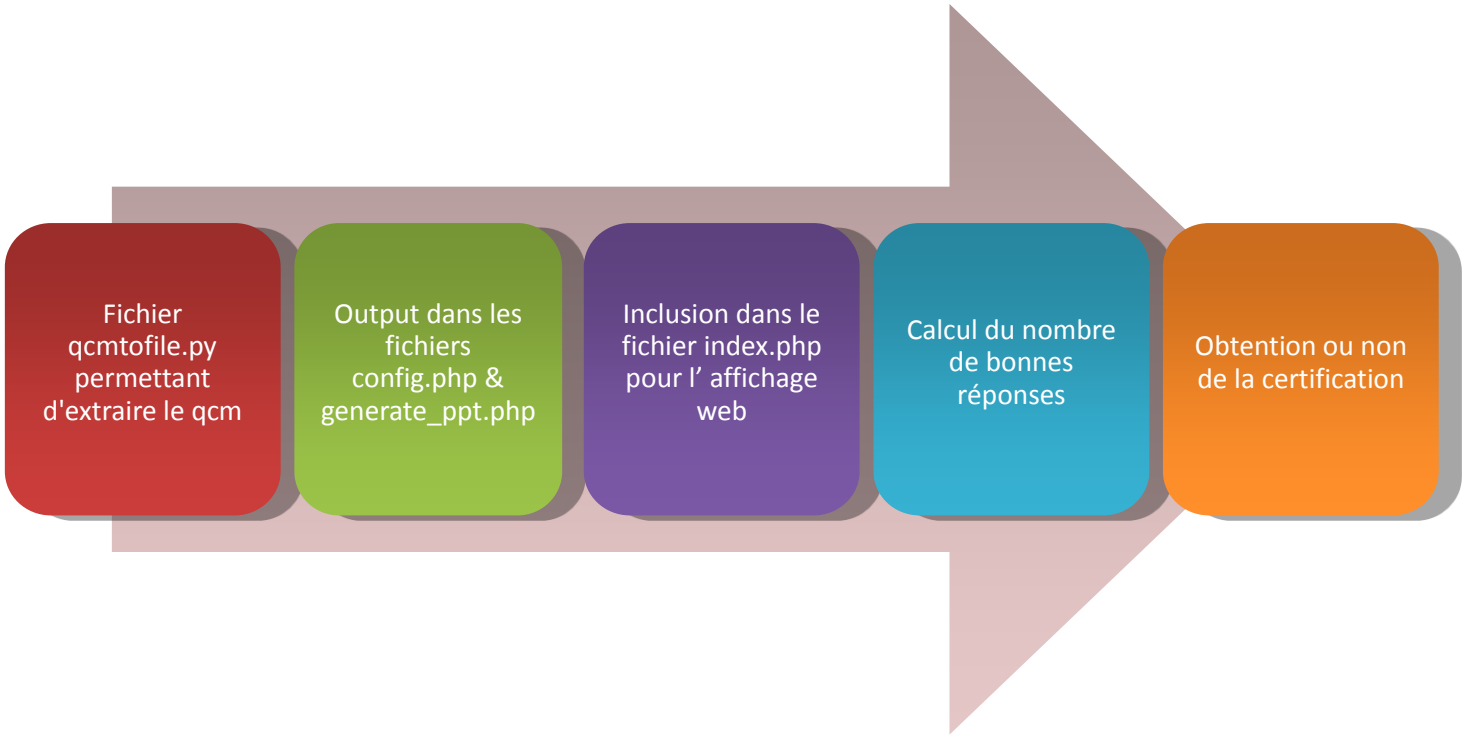


Figure 1 - Du fichier de questions-réponses à la mise en place de l'application web

b. Génération des qcm vers différents formats

Les deux formats précédemment choisis (page web et Powerpoint) répondent à des critères de facilité d'utilisation. Ce n'est cependant qu'un exemple parmi tous les formats de sortie que nous pourrions imaginer. Le but de cette petite application est de comprendre que le système d'extraction de causations que nous avons mis en place est ensuite réutilisable à l'infini pour générer différents types de documents en sortie.

Figure 2 - Interface de l'application web

Figure 3 - Génération d'un Powerpoint du qcm

PARTIE 3 – Problèmes à résoudre et Solutions envisageables

I. Création automatique d'une ontologie

La création automatique d'une ontologie à partir du texte brut du corpus, apporterait un gain conséquent à notre application. Pour le moment, l'extraction que nous effectuons est simplement « terminologique ». Par ailleurs, elle n'est que partiellement automatisée. Elle est automatique mais adaptée au corpus. Il faut prendre connaissance du corpus pour être au courant des termes récurrents permettant de prétendre qu'un groupe de mots est un concept propre au monde OpenOffice.

Une première idée serait donc d'utiliser les définition inclues dans le texte. Par exemple, « L'outil correction automatique, permet de corriger automatiquement des textes. » On extrait alors « outil », qui est un hyperonyme de « correction automatique de texte » et ainsi de suite pour les autres termes.

Dans l'article intitulé *Génération automatique des représentations ontologiques*, (Johannes Heinecke, 2006), une autre démarche est présentée. Elle inclut :

- Des lexiques complets pour le français
- Un parseur syntaxique
- Des représentations sémantiques indépendantes de la langue grâce à un thésaurus multilingue comportant plus de 100 000 relations sémantiques
- Des règles sémantiques permettant de construire un tissu sémantique à partir de l'arbre syntaxique

Bien que cette méthode puisse marcher sur des corpus plus généralistes, elle n'est pas adaptée à notre projet. En effet, dans le cadre de corpus traitant d'un sujet bien précis comme le nôtre, l'ontologie est

toujours particulière au corpus en question, ce qui lui donne par ailleurs son originalité. Ainsi, le terme « langue », dans un dictionnaire généraliste, fait référence à l'ensemble des signes oraux et écrits qui permet à un groupe donné de communiquer. Mais le terme « langue », dans le monde OpenOffice, possède une acception légèrement différente. L'on parle dans ce cas, de toutes les langues qu'il est possible de paramétrer pour écrire un texte avec OpenOffice.

Figure 1 - Entités propres au monde OpenOffice

Dans ce cas précis, comment pouvons-nous faire pour qu'un programme puisse extraire de façon automatique le terme « langue » spécifique au monde OpenOffice ? Etant donné que nous avons besoin de notre ontologie, avant le parsing syntaxique, puisque la délimitation des termes spécifiques au corpus est primordial pour une bonne analyse syntaxique, nous comprenons que l'analyse sémantique ne peut pas nous servir, puisque c'est une étape post analyse syntaxique. En conclusion, à partir du texte brut, notre système doit être capable de reconnaître quels termes sont spécifiques au monde OpenOffice et quels termes ne le sont pas.

Pour ce faire, le dernier recours est d'utiliser une méthode statistique d'extraction ontologique, c'est-à-dire de créer une ontologie basée sur les motifs fréquents du corpus. Cette méthode nous est utile puisqu'elle permet alors d'isoler tous les termes spécifiques au monde OpenOffice, pour pouvoir procéder au parsing syntaxique. Dans l'article *Mise à jour d'ontologie basée sur des motifs fréquents*, (Lisa Di Jorio, 2007), nous retrouvons cette philosophie. Toutes les relations fréquentes entre même termes sont alors isolées et placées dans l'ontologie.

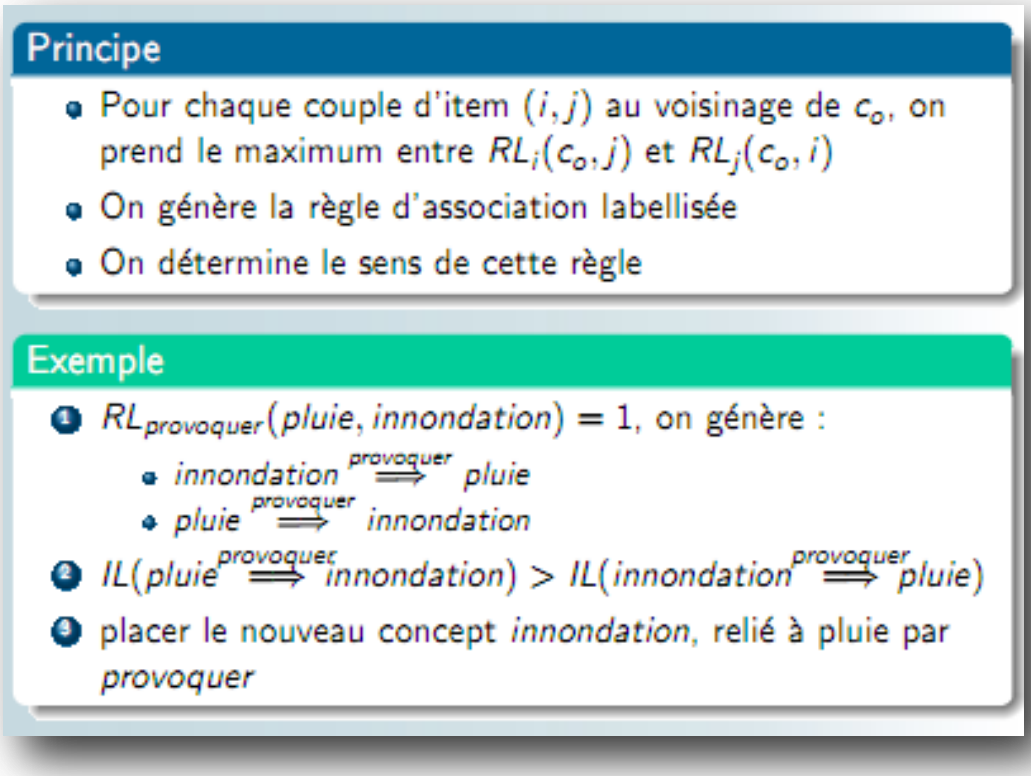


Figure 2 - Système statistique de recherche de relations fréquentes (Lisa Di Jorio, 2007)

II. Résolution complète des anaphores

Nous avons déjà expliqué dans un précédent chapitre consacré aux anaphores[21], que ces dernières peuvent se situer au niveau phrastique, mais peuvent également faire référence à un terme se situant dans une autre phrase.

Figure 3 - Anaphore au niveau phrastique

A ce stade, la construction de l'ontologie spécifique au corpus étant établie grâce à une méthode statistique couplée à une méthode de recherche de

[21]Anaphores : Partie IV, sous-partie c

relations fréquentes, nous avons imaginé un algorithme qui nous permettrait de résoudre les anaphores. A une anaphore rencontrée, il consisterait à rechercher le terme référencé dans l'ontologie, le plus proche spatialement. Par exemple :

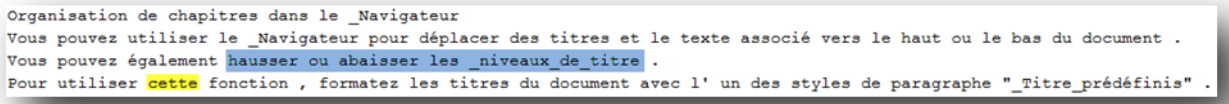


Figure 4 - Résolution des anaphores

Dans cet exemple, le pronom démonstratif anaphorique « cette », fait référence à l'expression verbale « hausser ou abaisser les niveaux de titre ». Nous aimerions donc que l'expression « cette fonction », puisse être automatiquement remplacée par « la fonction niveaux de titre. »

Pour cela, la terminologie précédemment crée a isolé « niveaux de titre », comme étant un terme spécifique au monde OpenOffice. Elle pourrait également avoir placé une relation ontologique entre « hausser », « abaisser » d'un côté et « niveaux de titre » de l'autre. L'algorithme recherche donc dans un premier temps, dans l'ontologie le terme qui a le plus de similarités avec les mots qui entourent l'anaphore, contexte précédant et suivant inclus. Plusieurs cas se présentent :

- L'ontologie contient le terme « la fonction haussement ou abaissement des niveaux de titre », «haussement ou abaissement des niveaux de titre ». L'algorithme repère que la sémantique qui entoure « cette » se rapproche le plus de ce terme en particulier. L'algorithme procède donc au remplacement de « cette fonction », par le terme extrait de l'ontologie.
- L'ontologie ne contient pas le terme « haussement ou abaissement des niveaux de titre », mais elle a identifié une relation récurrente entre « niveaux de titre » d'un côté, « hausser » ou « abaisser » de l'autre, grâce à la méthode de récurrence statistique précédemment expliquée. L'algorithme remplace donc « cette fonction » par « la » + « fonction » + l'expression verbale « qui hausse ou abaisse les niveaux de titre ».
- L'ontologie ne contient pas le terme « haussement ou abaissement des niveaux de titre » et n'a pas identifié de relation récurrente entre « niveaux de titre » d'un côté et « hausser » ou « abaisser » de l'autre. Seul le terme « niveaux de titre » est donc présent dans l'ontologie.

L'algorithme remplace « cette fonction », par « la » + « fonction » + « niveaux de titre ». ça me paraît bien.

III. Amélioration du parsing syntaxique

Une autre amélioration pourrait être apportée au niveau de la réduction d'erreurs d'analyse du parsing syntaxique. Comme nous avons pu l'étudier, le parseur syntaxique que nous utilisons actuellement commet des erreurs d'analyse. C'est un parseur qui utilise un modèle du français qui a été entraîné par Alexis Nasr et Frédéric Bechet sur une version dépendantielle du french tree bank. Cependant, il n'a pas été adapté à notre domaine. Pour améliorer les résultats du parseur, il faudrait donc faire une annotation corrigée à la main de notre corpus et l'ajouter à l'entraînement de Mate Parser.

IV. Pourcentage d'extraction de causations amélioré

Au fur et à mesure que nous serions confrontés à des corpus différents, nous pourrions identifier de nouveaux types de causation et les ajouter au programme d'extraction de causations. Cette étape nécessite donc un repérage humain des types de causation d'un corpus. Notre avantage cependant, c'est que dans le cadre de corpus liés à un apprentissage de la bureautique et de l'informatique, nous sommes toujours confrontés plus ou moins aux mêmes types de causations. Mais il ne faut pas nécessairement s'arrêter à ce type de corpus. Notre approche d'affinement du système de corpus en corpus vise à pouvoir extraire également des causations d'autres types de corpus. La démarche serait toujours la même :

1. Identification des causations spécifiques à un corpus lambda
    a. Ces causations sont déjà présentes dans notre programme ? Nous laissons le programme tel quel.
    b. Certaines de ces causations ne sont pas repérées par notre programme ? Nous permettons leur reconnaissance par l'intermédiaire de notre fonction de recherche de causations.

Par ailleurs, il est très important de souligner que l'analyse sémantique nous permettrait de générer un nombre accru de type de questions-réponses et d'en générer même à partir de phrases qui ne seraient pas des causations. Nous pourrions alors volontairement extraire des questions liées à la localité, au but, à la provenance, à la manière, à la temporalité etc. Avec une phrase toute simple, nous pouvons déjà observer les bénéfices d'une telle approche :

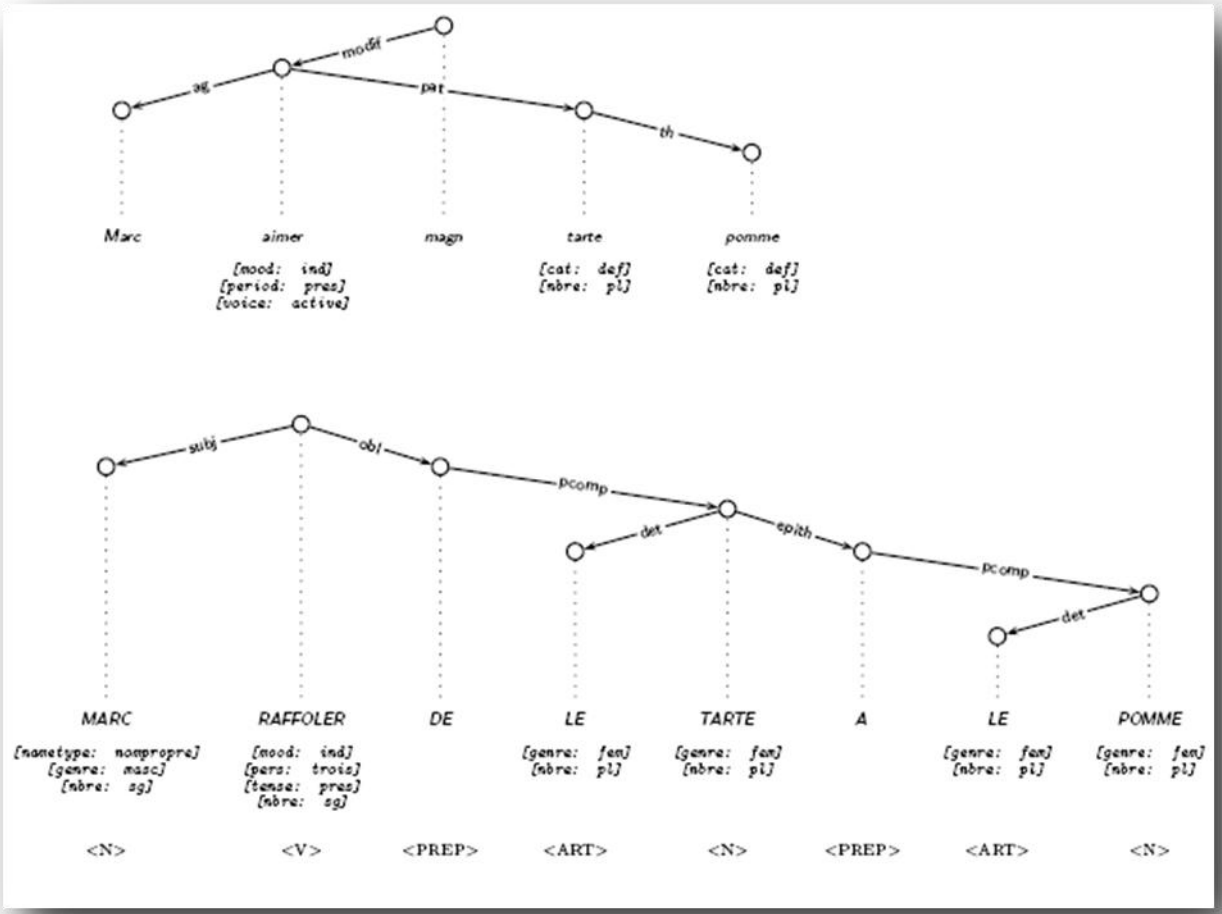


Figure 1 – Représentation des graphes sémantique et syntaxique de l'Interface Sémantique-Syntaxe mise en place par Pierre Lison

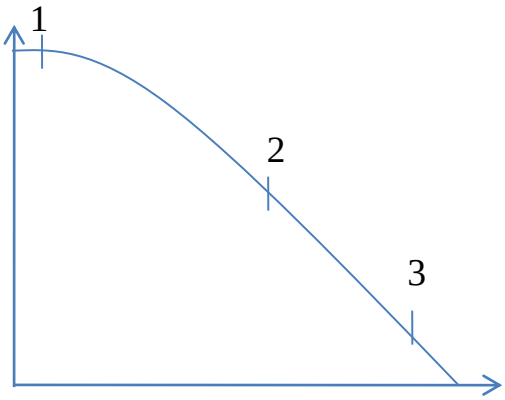
Dans cette phrase très simple, qui n'est pourtant pas une causation, nous pouvons déjà extraire de nombreuses questions-réponses :

- ✓ De quoi raffole Marc ?
- o De la tarte aux pommes
- ✓ Qui raffole de la tarte aux pommes ?
- o Marc
- ✓ Qu'est-ce qui relie Marc aux tartes aux pommes ?
- o L'amour

V. Performance du système sur divers types de corpus

L'une des particularités du système que nous pourrions mettre au point à l'issue de notre thèse, est que la grammaire d'interface Syntaxe-Sémantique que nous pourrions élaborer, serait spécifique aux corpus de bureautique et d'informatique. Ce ne serait donc pas un système de sémantisation automatique « universel ». Il resterait dans un cadre bien spécifique et dès que l'on soumettrait au système un corpus possédant une terminologie et une thématique légèrement différente, l'analyse deviendrait de plus en plus mauvaise à mesure que nous nous éloignerions de notre thématique.

Causations extraites en %



Légende :
1. Corpus de fichiers d'aide de logiciels de bureautique et d'informatique
2. Corpus d'explication de traitements médicaux
3. Corpus de textes poétiques

Bien évidemment, le but ultime serait de pouvoir extraire automatiquement toutes les causations de n'importe-quel corpus soumis au système. Dans ce cas, il faudrait, au fur et à mesure que nous présentons un nouveau type de corpus au système :

- Analyser la structure sémantique du corpus et élargir l'interface syntaxe-sémantique, ce qui inclut donc un travail linguistique préalable pour chaque nouveau type de corpus, mais ce qui reste un gain de temps considérable et profitable, puisque le système d'analyse et d'annotation sémantique s'élargirait au fur et à mesure que de nouveaux corpus seraient analysés

Pour nous convaincre de l'amélioration conséquente qu'une analyse sémantique apporterait à notre système, nous dressons dans le chapitre suivant un état de l'art de la théorie Sens-Texte, grande pionnière en terme de recherche de sens dans un texte.

PARTIE 4 – La Théorie Sens-Texte - Etat de l'art

I. Introduction

Tout au long de ce mémoire, nous avons insisté sur le besoin d'une analyse sémantique complète du corpus. Le but ultime est en effet de créer un corpus annoté à la fois en syntaxe et en sémantique. Le cas des causations n'est qu' un cas de relation sémantique que nous avons isolé pour présenter le projet.

Pour faire ce corpus sémantique, il faut un modèle qui propose :

1. des représentations sémantiques de toutes les phrases
2. qui s'intéresse au lexique, autrement dit qui sait encoder ces relations sémantique-syntaxe : X-cause->Y peut être réalisé syntaxiquement de manière a, b, c, ...

L'état de l'art de la Théorie Sens-Texte présenté ici, permettra donc au lecteur de mieux appréhender la façon dont nous pourrions réaliser un tel corpus.

Le langage, c'est la faculté d'expression que possède l'homme et qui lui permet de communiquer sa pensée grâce à la parole ou à l'écriture. Depuis la nuit des temps, l'homme utilise le langage. Dans le présent chapitre, nous dressons un état de l'art des grammaires de dépendance et de la théorie Sens-Texte avant de décrire notre approche pour créer un système de questions-réponses basé sur les grammaires de dépendance. Enfin, nous expliquons les limites rencontrées dans ce travail et pourquoi nous envisageons de construire l'interface Sens-Texte dans le cadre de notre thèse et dans le but d'améliorer notre système de Q.C.M.

II. Le Traitement Automatique du Langage Naturel

Les recherches dans le domaine du TALN ont commencé en 1950, lorsqu'Alan Turing proposa le fameux test de Turing comme critère d'intelligence. Ce critère dépend de la capacité d'un programme informatique de personnifier un humain dans une conversation écrite en temps réel. Un peu plus tard durant la guerre froide, alors que les américains cherchaient un moyen de traduire automatiquement des textes russes, la première expérience, baptisée

expérience de Georgetown-IBM, fut développée conjointement par l'Université de Georgetown et Ibm et consistait à traduire automatiquement plus de soixante phrases russes en anglais. Malgré l'enthousiasme et la confiance des chercheurs qui clamaient que le problème de la traduction automatique serait résolu d'ici trois à cinq ans, le projet tomba à l'eau. Les recherches en TALN reprirent de façon sérieuse beaucoup plus tard, dans les années quatre-vingt, mais les modèles alors développés furent basés sur les statistiques.

A l'heure actuelle, les applications TAL sont multiples. On y retrouve la traduction automatique, la correction orthographique, la recherche d'information, le résumé automatique, la reconnaissance d'entités nommées, la résolution d'anaphores, la génération automatique de textes, la synthèse de la parole, la reconnaissance vocale ou encore les agents conversationnels et la reformulation. Autant d'applications qui dénotent un intérêt grandissant dans un monde où l'information écrite ou orale devient de plus en plus lourde à traiter.

On notera cependant que malgré les nombreuses possibilités du TAL, le problème de la compréhension même du langage par la machine n'a jamais été résolu. Les logiciels actuels sont incapables de comprendre le langage naturel. Cependant de nombreuses théories et cadres linguistiques ont été développés à partir des années quatre-vingt, le but étant alors de « modéliser » la langue de façon formelle. C'est alors une approche symbolique et non plus statistique qui est envisagée par ces théories, dont la plus connue reste la théorie Sens-Texte, développé par Igor Mel'cuk.

III. L'importance de la modélisation en linguistique

Les approches symboliques se basent sur des descriptions et des modélisations explicites de la langue naturelle, basées sur des ressources linguistiques élaborées manuellement. Le but est alors de créer un système qui puisse représenter au mieux la vérité sémantique du domaine étudié et pour cela il faut se prémunir d'un modèle.

La notion de modèle est fondamentale en sciences. Elle l'est tout particulièrement dans le domaine de la linguistique. En effet, comme le souligne le linguiste Claude Hagège, «La linguistique est la seule science actuelle dont l'objet coïncide avec le discours qu'elle tient sur lui.» De plus, tous les phénomènes de la linguistique sont inobservables et impalpables. Tout au plus est-il possible d'en capter la modularité des sonorités dans le cas de segments oraux. La linguistique, c'est comme le souligne le linguiste Igor Mel'cuk, une boîte noire dont la structure interne est inobservable et que nous devons

modéliser. C'est dans cette perspective qu'Igor Mel'cuk a commencé à mettre au point la théorie Sens-Texte, il y a plus de trente ans, en créant des définitions rigoureuses et fines pour chaque concept du dit modèle linguistique. Dans le cadre de cette théorie, Igor Mel'cuk énonce la supériorité des grammaires de dépendance sur les grammaires syntagmatiques pour capter les relations syntaxiques d'une phrase.

IV. Représentation Sémantique

La sémantique est définie, de façon traditionnelle, comme la science ou la théorie des significations. Bien évidemment, nous entendons par « signification », tout ce qui se rapporte aux significations linguistiques seulement. En ce sens, il ne faut pas confondre la sémantique avec la sémiologie, qui est quant à elle la science des procédés ou systèmes de communication. Dans le présent mémoire, lorsque nous parlerons de sémantique, nous désignons donc la science des significations linguistiques uniquement. Il est également primordial de ne pas confondre la sémantique avec la lexicographie, qui elle ne s'occupe que de la description de la signification des mots, telle qu'elle apparaît dans un dictionnaire. Ce point est important si l'on veut comprendre la notion de lexique sémantique telle qu'elle est représentée dans ce que l'on nomme un DEC (Dictionnaire Explicatif et Combinatoire). Nous reviendrons sur ce point dans la partie consacrée à cet effet.

Maintenant que nous avons défini ce qu'était la sémantique, il serait bon de se demander ce qu'est une signification en linguistique. C'est un problème majeur et qui a été la source de nombreux désaccords entre linguistiques au cours du XXème siècle. Depuis Saussure, nous admettons qu'un signe linguistique est une entité à deux facettes, le signifiant, sa forme phonique, et le signifié, sa signification intrinsèque. En plus de cette double facette, le signe renvoie, dans la réalité non linguistique, à quelque-chose qui n'est pas lui et que l'on appelle communément « concept » ou « référent du signe ».

La sémantique est en réalité, encore aujourd'hui, un domaine difficile à aborder pour tout linguiste. Buyssens disait lui-même en 1960, « L'étude du signifié est la partie la plus difficile de la linguistique. ».Il est ici important de se demander pourquoi la sémantique a tant résisté et résiste encore aux méthodes structurales qui ont si bien triomphé dans les autres disciplines linguistiques. En réalité, de nombreuses théories et types d'analyses sémantiques ont vu le jour

depuis Saussure. George Mounin, dans son ouvrage intitulé « La sémantique », nous en dresse un descriptif exhaustif. Il distingue quatre types d'analyses majeures.

- Les analyses sémantiques « formelles » qui posent que les vrais rapports entre signifiants et signifiés constituent un système où la valeur de chaque terme est déterminée « par la présence ou l'absence des autres ». Saussure en était un partisan.

- Les analyses sémantiques « conceptuelles » qui se basent sur les rapports associatifs entre les mots. Le « concept » est à la base de ces analyses. Par exemple, le fait de classer les termes « avoir peur », « craindre », « redouter » dans le champ conceptuel de la notion de peur relève de ces analyses. Par contre, le souci majeur, c'est la subjectivité, puisque le linguiste est le seul à décider de son champ conceptuel.

- Les analyses sémantiques « logiques » effectuées par des logiciens, consistent à diviser toute étude des signes en trois parties : a) les rapports entre signes et choses signifiées (sémantique); b) les rapports des signes entre eux (syntaxe) ; c) les rapports entre signes et leurs utilisateurs (pragmatique)

- Les analyses sémantiques « artificielles » qui visent plutôt à la normalisation des terminologies techniques. Nous nous rapprochons plus ici de la notion d'ontologie, qui consiste à réduire à un système toutes les notions référencées d'un domaine par le linguiste et de déterminer ensuite tous les rapports qui existent entre les notions, ou « objets » de ce système. Le problème de ces analyses, c'est qu'elles semblent en effet efficaces dans les sciences physiques et naturelles, dont elles sont capables de créer de véritables taxinomies, mais elles restent inefficaces dans les sciences humaines où le domaine semble beaucoup trop large.

Ce descriptif est important dans le but de comprendre dans quelle optique se situe la théorie Sens-Texte, qui est clairement dans l'optique d'une analyse sémantique formelle, dans la lignée Saussurienne. Le DEC (Dictionnaire Explicatif et Combinatoire) est en ce sens primordial, puisqu'il permet de

modéliser les principales propriétés lexicales d'un sème. Nous reviendrons sur ce point.

V. Théorie Sens-Texte

a. Les grands principes

L'interface sémantique-syntaxe est ancrée dans le cadre de la théorie Sens-Texte (TST). Nous décrivons dans ce chapitre l'appareil conceptuel élaboré dans le Cours de morphologie général de Igor Mel'čuk (1993, 1994, 1996a, 1997b, 2000). Toutefois, le formalisme que nous souhaiterions utiliser, si nous étendions notre application, est celui de la grammaire d'unification Sens-Texte, que nous décrivons au chapitre suivant.

Le modèle Sens-Texte est un modèle modulaire dont le but est de pouvoir décrire les langues. C'est un système fini de règles qui spécifient une correspondance multivoque entre un ensemble infini dénombrable de sens et un ensemble infini dénombrable de textes. Dans un modèle Sens-Texte, également appelé MST, les sens apparaissent sous forme d'objets symboliques appelés RSem et les textes sous forme d'objets symboliques appelés RPhon. Dans sa leçon inaugurale au collège de France, Igor Mel'čuk formalise cela de la façon suivante :
(1) {RSem i } langue ⇔{RPhonj} | 0<i,j

Le modèle Sens-Texte tente de se rapprocher un maximum de la façon dont le locuteur traduit le sens qu'il a dans sa tête sous la forme de signes linguistiques. C'est le passage du signifié, c'est-à-dire du sens, au signifiant, c'est-à-dire au son, qu'Igor Mel'čuk cherche à modéliser par le biais de la théorie Sens-Texte. Le sous-niveau P (profond) est orienté vers le sens, tandis que le sous-niveau S (de surface) est orienté vers le texte. Le principal avantage d'un modèle Sens-Texte, c'est son architecture modulaire.
Un modèle Sens-Texte complet comporte sept niveaux :

Sem ⇔SyntP ⇔SyntS ⇔ MorphP ⇔ MorphS ⇔ PhonP⇔PhonS

Grâce à l'architecture ci-dessus, on comprend mieux l'interface entre niveaux de surface et niveaux profonds. Le passage de la représentation profonde à la représentation de surface inclut la hiérarchisation et la lexicalisation. On peut donc dire que la représentation profonde est la représentation universelle du module linguistique alors en jeu, tandis que la représentation de surface, c'est la représentation linguistique propre à chaque langue du module linguistique alors décrit.

Si nous faisons abstraction des structures profondes, nous pouvons réduire notre modèle Sens-Texte à quatre niveaux : **Sem⇔Synt⇔Morph⇔Phon.**

Chaque représentation utilisée par le MST est constituée de plusieurs objets appelés des structures. Une RSém comprend donc trois niveaux :

- **Structure sémantique :** c'est le sens propositionnel de l'énoncé ; un graphe orienté dont les nœuds sont étiquetés par des sémantèmes.
    - Les sémantèmes peuvent être de deux types. Ce sont soit des sémantèmes lexicaux, c'est-à-dire des mots ou ensemble de mots (locutions par exemple), soit des sémantèmes grammaticaux, aussi appelés morphèmes flexionnels (<singulier>, <pluriel>, <passif>, <futur> etc.)
- **Structure sémantico-communicative :** c'est le sens communicatif ou subjectif de l'énoncé, l'itinéraire que le locuteur suit à travers la structure sémantique
- **Structure rhétorique :** c'est ce qui reflète les intentions artistiques et stylistiques du locuteur, comme l'ironie, l'emphase etc. Il est également important de spécifier que le noyau d'une RSem, c'est la structure sémantique SSem d'une famille de phrases (quasi) synonymes. Le fait que le sens soit défini par la TST comme un invariant de paraphrase est un avantage majeur, résolvant le problème des nombreuses structures syntaxiques ayant le même sens. En français par exemple, une phrase

Comme nous le verrons, toute interface Sémantique-Syntaxe d'un modèle Sens-Texte est constituée d'un lexique et d'une grammaire. Le lexique d'un tel modèle est appelé un DEC (dictionnaire explicatif et combinatoire) et est représentable informatiquement. La grammaire d'un tel modèle n'a pas de nom particulier, mais elle peut être implémentée grâce au formalisme GUP (grammaires d'unification polarisées). Nous commencerons par expliquer la notion de lexique, avant de nous intéresser à la grammaire.

b. Lexique d’un Modèle Sens-Texte

Le lexique d’un modèle Sens-Texte est basé sur ce le DEC (dictionnaire explicatif et combinatoire). Chaque entrée de ce dictionnaire se divise en trois parties :

- La partie *sémantique* donne la définition lexicographique de la lexie ;
- La partie *syntaxique* donne le tableau de régime de la lexie, ou plus explicitement, la correspondance entre les actants sémantiques, les actants syntaxiques profonds et leur expression de surface ;
- La partie de *cooccurrence lexicale* donne les valeurs des fonctions lexicales pour la lexie concernée, c’est-à-dire les collocations formées avec cette lexie et les dérivations sémantiques de la lexie en question. L’entrée se termine par une série d’exemples.

Il existe à l’heure actuelle plusieurs DEC. Les plus complets sont ceux développés par l’Observatoire de Linguistique Sens-Texte de Montréal (OLST). Il y a notamment le **DiCouèbe** administré par Igor Mel'čuk et Alain Polguère et qui proposent plus de 26000 liens lexicaux. On retrouve également le **DiCoInfo,** base de données lexicales contenant des termes fondamentaux appartenant aux domaines de l’informatique et de l’Internet. Ce DEC est administré par Marie-Claude L’Homme. C’est celui-ci qui pourrait convenir à la création de notre interface, dans la perspective d’une thèse.

Structure d'un article :

Les articles sont découpés en une dizaine de rubriques, dont certaines sont obligatoires :

-*Entrée* ***(comprenant un numéro d'acception)***
-*Information Grammaticale* ***(partie du discours à laquelle appartient l'entrée)***
-*Statut* ***(Statut 0 =Rédaction terminée ; Statut 1 = Rédaction très avancée ; Statut 2 = Rédaction avancée)***
-*Définition*
-*Structure Actancielle* ***(Actants sémantiques du terme et leurs rôles par rapport au terme décrit)***

D'autres rubriques peuvent venir se rajouter à cette base :

-la rubrique *Rédacteur(s)*
-la rubrique *Date de mise à jour*
-la rubrique *Synonyme(s)* (qui comprend également Variante(s) et féminin)
-la rubrique *Liens lexicaux*

Exemple d'article:

Rôles Actanciels

Les étiquettes les plus couramment utilisées dans le DiCoInfo sont AGENT, PATIENT, DESTINATION, SOURCE, INSTRUMENT et LIEU définies ci-après.

AGENT	Actant qui renvoie à l'origine de l'action exprimée par le terme ou à l'élément responsable de l'existence ou de l'utilisation d'une entité exprimée par le terme. **Ex** : STOCKER 1b, v. tr. : **AGENT** ~ PATIENT dans DESTINATION (réalisation possible de l'agent : *homme*)
PATIENT	Actant qui renvoie à l'entité subissant l'action exprimée par le terme, actant qui désigne l'élément créé ou utilisé par un agent ou sur lequel l'agent intervient. **Ex** : ABSORPTION 1, n. f. : ~ de **PATIENT** par DESTINATION (réalisations possibles du patient : *carbone, chaleur*)
DESTINATION	Actant qui renvoie au but visé par une action entreprise par un agent ou encore actant qui renvoie au but visé par la fonction typique d'une entité. **Ex** : PIÉGEAGE 1b, n. m. : ~ de PATIENT dans **DESTINATION** par AGENT ou CAUSE (réalisations possibles de la destination : *écosystème, sol*)
SOURCE	Actant qui renvoie à l'élément à partir duquel une activité est effectuée ou encore actant qui renvoie à l'élément à partir duquel la fonction typique d'une entité est réalisée. **Ex** : ÉMISSION 1, n. f. : de PATIENT par **SOURCE** (réalisations possibles de la source *: soleil, terre*)
CAUSE	Actant qui renvoie à une action qui détermine la réalisation d'une autre action ou l'existence d'une entité. **Ex** : AUGMENTATION 1b, n. f. : ~ de PATIENT par **CAUSE** (réalisations possibles de la cause : *changement, émission*)
INSTRUMENT	Actant qui renvoie à l'entité utilisée par l'agent pour réaliser l'action exprimée par le terme ou pour créer ou utiliser l'entité dénotée par le terme. **Ex** : CLIQUER 1, v. intr. : AGENT ~ sur PATIENT avec **INSTRUMENT** (réalisations possibles de l'instrument : *bouton, souris, mini-souris*)
	Actant qui renvoie à l'endroit où se déroule une activité ou encore à

LIEU	l'endroit où a lieu la fonction typique rattachée à une entité. **Ex** : un NAVIGATEUR 1, n. m. : ~ utilisé par AGENT pour aller dans **LIEU** (réalisations possibles du lieu : *réseau, Internet, Web*)

Accès :

Tout le monde peut consulter le DiCoInfo en se rendant sur le site internet du dictionnaire : http://olst.ling.umontreal.ca/cgi-bin/dicoinfo/search.cgi
Son utilisation est très pratique. Il suffit de rentrer le mode, la langue, la précision et de taper un mot recherché, puis de cliquer ensuite sur « Rechercher ». Si vous désirez plus de détails, vous pouvez consulter facilement la documentation du DiCoInfo : http://olst.ling.umontreal.ca/dicoinfo/manuel-DiCoInfo.pdf

c. Grammaires d'Unification Sens-Texte

La précédente présentation de la théorie Sens-Texte et de la structure du DiCoInfo nous a permis de comprendre l'importance de l'utilisation d'un formalisme dans le domaine de la linguistique. En effet, comme le souligne Claude Hagège « la linguistique est la seule science actuelle dont l'objet coïncide avec le discours qu'elle tient sur lui. » (Hagège, 1986). Les linguistes sont donc confrontés à un phénomène de mise en abîme dans lequel ils sont contraints d'utiliser les langues naturelles pour décrire et expliquer les langues naturelles.

Le formalisme GUST est une modélisation mathématique de la théorie Sens-Texte. (Kahane, 2001). Ce formalisme permet de représenter l'interface entre différents modules linguistiques. Dans le cadre de notre travail, nous réduirons notre champ d'étude aux niveaux de représentation sémantique et syntaxique, en mettant principalement l'accent sur l'interface entre les deux niveaux et sur les règles permettant de passer du niveau syntaxique au niveau sémantique. Dans ce formalisme, la structure syntaxique est un arbre de dépendance et la structure sémantique est un graphe. Il est important de souligner le fait que chaque module va utiliser la polarité des modules adjacents pour son articulation avec eux. Dans notre modèle basé sur les Grammaires d'Unification Polarisée.(Kahane, 2001), nous devons donc construire une grammaire syntaxique, une grammaire sémantique et une grammaire d'interface qui viendra décrire les règles de correspondance entre les deux niveaux de représentation.

d. Grammaire Syntaxique (G_{Synt})

La représentation syntaxique est assurée par des arbres de dépendance (*Eléments de syntaxe structurale*, Lucien Tesnière, 1976), dont les nœuds sont étiquetés par des lexèmes. Des fonctions grammaticales vont alors venir lier ce nœud à d'autres objets de type grammème. Notre grammaire syntaxique nommée G_{Synt} possède une polarité nommée P_{Synt}, permettant de vérifier quels sont alors les objets construits par la règle et si les fonctions associées à chaque objet sont bien instanciées : la source et la cible pour les arcs, l'étiquette, la partie du discours pour les nœuds et les fonctions grammaticales si il y a lieu. De plus, notre grammaire syntaxique possède une structure initiale, tête ou racine de la représentation syntaxique. En ce sens, elle s'apparente tout à fait à une structure de dépendance selon l'acception de Lucien Tesnière. Nous avons déjà traité ce niveau d'analyse dans le présent mémoire.

Puis, viennent alors s'ajouter à cette structure de départ des règles lexicales, sagittales et grammaticales. Les règles lexicales se composent de parties du discours et de grammèmes (morphèmes grammaticaux). Les règles sagittales viennent décrire différentes relations syntaxiques possibles, donc elles font office de fonctions, mais des fonctions qui ne sont pas totalement saturées par leurs parties du discours respectives. Les règles grammaticales sont quant à elles des règles qui dépendent du contexte : un verbe à l'indicatif attend un sujet, un nom pluriel attend logiquement un numéral ou un article pluriel etc. Pour réaliser notre surface syntaxique, nous avons déjà expliqué que nous utilisons un parseur syntaxique en dépendance nommé « Mate Parser ».

Tab 1. - Récapitulatif des fonctions syntaxiques utilisées dans notre grammaire syntaxique

Dénomination	**Fonction**	**Exemple**
mod (epith)	Epithète du nom	Adjectif - Ex : un document *riche* Complément du nom introduit par une préposition - Ex : les titres *du document*
mod (attr)	Attribut du sujet de la copule Pour le différencier de l'adjectif, on a coutume de dire qu'il fait corps avec le verbe principal, qu'il sert à marquer	Attr. Adjectival - Ex : Le titre est *souligné.* Attr. Substantival - Ex : Les borgnes sont *rois.* Attr. Adverbial - Ex : Les sections

	le rôle verbal.	sont *ensemble.* Attr. Intr. Par une Prép - Ex : Il reste *de glace.*
mod (adv)	Complément circonstanciel de l'adjectif ou de l'adverbe	Adv - Ex : un *très* joli titre Complément non adverbial, introduit par une préposition - Ex : un document riche *en commentaires*
mod (adv)	Complément circonstanciel du verbe	Adverbe – Ex : Vous pouvez *également* choisir cette option. Locatif circonstanciel – Ex : *Sur la barre d'outils*, il y a de nombreuses options. Autres compléments circonstanciels introduits par une préposition (but, manière, temps etc.) Complément circonstanciel sans préposition – Ex : Il fera ses devoirs *demain.*
coord	Fonction permettant de lier la conjonction au 2nd constituant	Ex : Elle clique sur le titre *et* le texte.
dep_coord	Fonction permettant de lier une conjonction à une préposition	Ex : et *de* définir
det	Déterminant du nom	Ex : *Les* titres, *Le* document
obj	Objet direct du verbe	Substantif - Ex : Jean utilise *cette fonction.* Clitique - Ex : Jean *l'*utilise.
de_obj	Déterminant de l'objet Préposition précédant l'objet	Det - Ex : insérer *des* éléments Prép- Ex : permettant *de* manipuler
suj	Sujet du verbe fini	Ex : *Vous* pouvez cliquer.
dep	Complément d'une préposition	Substantif – Ex : Après *cette manipulation*, vous pouvez tester le publipostage. Verbe – Ex : Vous pouvez utiliser

		le navigateur pour *déplacer des titres.*

Dans le cadre d'une représentation syntaxique en dépendance, il est important de prendre en compte la notion de projectivité. Un arbre de dépendance peut ou non être projectif, en fonction des constructions de la phrase qu'il représente. Ainsi, en français, nous relevons différente structures non projectives, à savoir les dépendances non bornées (principalement les extractions : relativisation, interrogation directe et indirecte, etc.) et la montée des clitiques. La puissance d'un arbre projectif vient du fait qu'il est facilement linéarisable par un algorithme. Le formalisme GUP permet de représenter de façon très complète la structure syntaxique de la phrase. En effet, en plus des parties du discours et des fonctions syntaxiques, viennent s'ajouter des règles grammaticales permettant alors la génération de différentes structures syntaxiques.

e. Grammaire Sémantique (G_{Sem})

Les représentations syntaxiques utilisées en GUP sont basées principalement sur un graphe de relations prédicat-argument entre sémantèmes. Il est possible de superposer sur ce graphe d'autres informations dans le but d'encoder la structure informationnelle ou les relations de portée. (Mel'cuk 2001, Kahane 2005). Les nœuds d'une représentation sémantique représentent les sémantèmes et les arcs représentent les relations prédicat-argument. La grammaire $G_{Sém}$ ainsi constituée est en conclusion une grammaire de graphes dont les objets portent chacun une polarité, qui indique quel est le nœud construit par la règle (polarisé en noir) et quels sont les nœuds constituant la valence sémantique du prédicat (polarisée en blanc). La valence est un trait syntaxique concernant principalement les verbes mais aussi quelques noms et adjectifs. On nomme valence d'un tel terme le nombre d'actants qu'il peut recevoir ou qu'il doit recevoir pour être saturé, c'est-à-dire fournir un syntagme grammaticalement correct.

La représentation sémantique qui découle de cette grammaire a pour but de représenter le sens linguistique d'un énoncé, c'est-à-dire l'organisation des signifiés des signes profonds de l'énoncé. Ce sont ces signifiés que l'on appelle des sémantèmes. Il est important de souligner que la représentation sémantique de ce système se résume à un graphe prédicatif. Elle n'a pas pour objectif de

modéliser la structure communicative, ni la hiérarchie logique, ni les aspects liés à la rhétorique ou à la référence.

Pour déterminer les arguments sémantiques d'un mot, un critère va jouer un rôle important. Il faut décider si l'argument potentiel est sémantiquement obligatoire ou non. Si nous considérons la phrase, « Vous pouvez utiliser le navigateur pour déplacer etc. », en appliquant ce critère, on attribue au verbe « utiliser », deux arguments, à savoir « vous » et « navigateur ». En effet, ils sont obligatoires pour que le verbe « utiliser » prenne tout son sens, tandis que le 3ème argument ne l'est pas. Cela signifie que tout emploi du verbe « utiliser » impose l'existence de deux arguments. Dans le cas précis de cette phrase nous avons alors établi le fait que « utiliser » avait deux arguments, un de type agent, l'autre de type patient, mais également un but. A ce stade de notre recherche, le lecteur pourra commencer à entrevoir le lien entre le DiCoInfo (DEC) et l'interface grammatical de notre système. Puisque le DiCoInfo nous donne les informations sémantiques des termes de notre corpus, grâce à la rubrique « Structure Actancielle », il va de soi que nous devons trouver un moyen d'intégrer à notre grammaire un lien vers les articles du DiCoInfo décrivant des termes de notre corpus. Pour tous les termes de notre corpus non présents dans le DicoInfo, cela n'est pas un problème, puisque l'ontologie est là pour compléter les informations données par le DiCoInfo. D'ailleurs vous constaterez que dans la construction de notre grammaire sémantique, nous nous sommes appliqués à choisir des dénominations que nous retrouvions dans le DiCoInfo, à savoir : Agent, Patient, Destination, Source, Cause, Instrument, Lieu. Voici ce que pourrait être à terme les fonctions utilisées dans une grammaire sémantique de notre corpus :

Tab 1. - Récapitulatif des fonctions sémantiques utilisées dans notre grammaire sémantique

Dénomination	Fonction	Exemple
Agent	Agent d'un procès	Ex : un navigateur : ~ utilisé par ***Agent {internaute 1}*** pour aller dans Lieu{Internet 1}
Destinataire	Destinataire d'un procès	Ex : un blogue : ~ créé par Agent{auteur 1} dans Lieu{Internet 1} pour offrir Patient{information 1} à

		Destinataire{internaute 1}
Cat	Catégorie définie ou indéfinie d'une entité (qui s'exprimera syntaxiquement par l'utilisation d'un article défini ou indéfini)	Catégorie indéfinie - Ex : ***une*** touche Catégorie définie – Ex : ***la*** touche
Cause	Actant qui renvoie à une action qui détermine la réalisation d'une autre action ou l'existence d'une entité.	CHANGEMENT 1b, n. f. : ~ de PATIENT par **CAUSE** (réalisations possibles de la cause : *utilisateur, concepteur*)
Cond	Introduction d'une condition logique	Ex : ***Si le curseur se trouve dans un paragraphe,*** cette icône n'est pas visible.
Consequence	Evènement ayant lieu si et seulement si une condition est remplie. Le déclenchement est donc typique des phrases conditionnelles.	Ex : Si le curseur se trouve dans un paragraphe, ***cette icône n'est pas visible.***
Destination	But visé par un procès	Ex : ***Pour déplacer les titres,*** vous pouvez utiliser cette icône.
Instrument	Instrument d'un procès	Ex : Formatez le document ***avec un style de paragraphes..***
Lieu	Cadre spatial d'un procès	Ex : un navigateur : ~ utilisé par Agent {internaute 1} pour aller dans ***Lieu {Internet 1}***
Manière	Manière dont un procès est exécuté	Ex : Formatez le document ***sans toucher***

Mode	Mode d'un procès (<indicatif>,<infinitif>, <impératif> etc.)	*Infinitif* – Ex : ***Baisser*** les titres *Indicatif* – Ex : ***Vous baissez*** les titres. *Impératif* – Ex : ***Baissez*** les titres.
Modifieur	Modifieur d'une entité	Ex : Haussez ***les*** niveaux ***de titres.***
Nombre	Nombre (<sing>,<pl>,<partitif>)	*Sing* – Ex : ***Un*** choix *Plur* – Ex : ***Des*** choix
Patient	Patient d'un procès	Ex : une page : ~ créée par Agent{auteur 1} dans Lieu{Internet 1} pour offrir ***Patient{information 1}*** à Destinataire{internaute 1}
Période	Période d'un procès exprimée relativement à la situation d'énonciation (<présent>,<passé>,<futur>)	*Présent* – Ex : ***Cliquez*** sur l'option. Future – Ex : Vous ***cliquerez*** sur l'option. Passé – Ex : Vous ***cliquiez*** sur l'option.
Source	Actant qui renvoie à l'élément à partir duquel une activité est effectuée ou à l'élément à partir duquel la fonction typique d'une entité est réalisée.	***Ex :*** *ÉMISSION 1, n. f. : de PATIENT par* ***SOURCE*** *(réalisations possibles de la source : ordinateur)*
Temps	Cadre temporel d'un procès, qui peut prendre différentes valeurs : a. *simultaneous*, pour identifier deux procès	a. *Simultaneous* – Ex : Activez la commande *lorsque vous positionnez la souris sur l'icône.* b. *Before* - Ex : Activez la

	simultanés b. *before*, pour identifier un procès ayant lieu avant un autre c. *after*, pour identifier un procès ayant lieu après un autre d. *act 1, act 2, act 3 etc.* lorsque plusieurs procès sont présents et qu'il faut les classer dans l'ordre d'exécution temporel. e. *frequence*, lorsqu'une même action se répète f. *date*, repère calendaire concret g. *duration* ,durée d'un évènement	commande *avant de saisir les données* c. *After* - Ex : Activez la commande a*près avoir saisi les données* d. *Act 1, Act 2, Act 3 etc.* – Ex : Choisissez *Fichier – Envoyer – Créer* e. *Frequence* – Ex : « Cliquez deux fois. » ; « Sauvegardez toutes les 10 minutes. » f. *Date* – Ex : « Sauvegardez le 8 Mai prochain. » g. Duration – Ex : « Attendez *15 minutes* avant de continuer. »
Theme	Thème d'un procès	Ex : Répondez ***« oui »*** à la question posée.
Voice	Voix d'un procès (<actif> ou <passif>)	*Voix Active* - Ex : L'utilisateur choisit une options *Voix Passive* - Ex : L'option est choisie par l'utilisateur.

f. Grammaire de correspondance Syntaxe-Sémantique

La grammaire de correspondance G que nous pourrions élaborer dans une perspective lointaine, permettrait de mettre en relation les structures des deux grammaires précédemment crées. Dans notre cas, puisque nous souhaiterions traiter l'interface Sémantique- Syntaxe, elle mettra en relation les structures de la grammaire syntaxique avec les structures de la grammaire sémantique.

L'interface Sémantique-Syntaxe $I_{\text{sém-synt}}$ est donc une grammaire de correspondance entre l'ensemble des graphes sémantiques décrits par $G_{\text{sém}}$ et l'ensemble des arbres syntaxiques décrits par G_{synt}.

On fournit à la grammaire un graphe sémantique, à la différence que chacun des objets reçoit maintenant une double polarisation ($P_{.\text{sém}}$, $P_{.\text{sém-synt}}$). Dans l'interface que nous désirons concevoir pour lier les deux grammaires précédemment crées, nous remarquons que le problème du verbe modifieur « pouvoir » a été traité à la fois en syntaxe et en sémantique, mais de façon bien évidemment différente. L'interface se charge alors de trouver une règle permettant de passer d'une représentation à une autre.

Conclusion

Au cours du travail de ce mémoire, nous avons pu constater que l'analyse syntaxique en dépendances apportait une nette amélioration quant à l'extraction des causations comparée à la technique dite du « sac de mots ». Nous avons vu que l'analyse syntaxique en dépendances nous permettait de naviguer dans les arbres syntaxiques et d'isoler telle ou telle structure syntaxique. En l'occurrence, nous avons arrêté notre choix sur les causations qui sont intéressantes d'une part car elles se prêtent bien à une intégration dans un qcm et d'autre part linguistiquement, car il existe un grand nombre de réalisations syntaxiques du lien sémantique « causer », mais nous aurions très bien pu choisir d'autres structures, les conditionnelles par exemple, qui font d'ailleurs l'objet de quelques explications. L'application subséquente à l'extraction, prouve l'utilité d'un tel système dans une perspective d'apprentissage et de validation des connaissances, tant pour le professeur, qui peut alors soumettre son étudiant à un test, que pour l'étudiant qui peut se rendre compte de son niveau de connaissances. Dans l'état actuel, les qcm générés contiennent encore trop d'erreurs pour envisager une utilisation directe. Par contre, on pourrait imaginer que le professeur choisisse parmi toutes les questions celles qui lui semblent bonnes, au lieu de les écrire lui-même à la main.

La troisième partie propose des solutions concrètes et envisageables, notamment dans la perspective d'une thèse, qui permettraient d'améliorer le système de manière conséquente, tout en continuant à utiliser l'analyse syntaxique. La quatrième partie, qui présente la théorie Sens-Texte, va plus loin encore, puisqu'elle propose de mettre cette théorie en application pour notre système. Plus particulièrement, elle propose de partir de l'analyse syntaxique, pour aboutir à une analyse sémantique, cela grâce à une grammaire de réécriture de graphes, accompagnée d'un lexique. Nous proposons toutes les fonctions sémantiques que nous pourrions utiliser dans notre grammaire sémantique, ainsi qu'un lexique. Le but serait alors, après analyse sémantique de notre corpus, d'imaginer quelles transformations nous permettraient de passer du graphe syntaxique, celui présentement utilisé, au graphe sémantique, celui présenté dans le dernier chapitre.

Nous terminons ce travail l'esprit ouvert, puisque de nombreuses questions ont été soulevées. Dans la perspective d'une thèse, nous aimerions pouvoir réaliser cette grammaire d'interface syntaxe-sémantique, adaptée au type de corpus sur lequel nous travaillons (fichiers d'aide de logiciels d'informatique). Nous avons démontré qu'une analyse syntaxique permettait

d’augmenter de façon conséquente le repérage de structures linguistiques. Nous avons expliqué que l’analyse sémantique permettrait bien plus de maniements linguistiques, puisqu’elle capterait le « sens » de la phrase. La thèse serait donc l’occasion de mettre cela en place.

Annexes

I. Annexe 1 – Recherche de causations – Corpus Test

P. 1) Pour définir ce paramètre, vous pouvez également passer par l'onglet Adaptation du texte. (swriter/main0214.xhp)

p.2) Pour définir une bordure inférieure, cliquez plusieurs fois sur le bord inférieur jusqu'à ce qu'une ligne pleine apparaisse. (swriter/guide/sections.xhp)

p.3)Répétez les étapes 2 à 6 pour créer un deuxième Style de page personnalisé avec un en-tête différent.
(swriter/guide/change_header.xhp)

p.4) Pour modifier l' espacement entre le trait de séparation et le contenu de l' en-tête ou du pied de page , désactivez la case à cocher _Synchroniser , puis saisissez une valeur dans la zone _En_bas .
(swriter/guide/header_with_lines.xhp)

p.5) En effet, lorsque vous modifiez le format de numérotation d'un style, tous les paragraphes formatés avec ce style sont automatiquement actualisés.
(swriter/guide/using_numbering.xhp)

p.6) Les retraits peuvent être modifiés en faisant glisser les trois petits triangles situés sur la règle horizontale.. (swriter/guide/ruler.xhp)

p.7) Pour modifier le retrait de paragraphe à gauche ou à droite , sélectionnez chaque paragraphe concerné , puis faites glisser le triangle inférieur droit ou gauche de la règle horizontale vers la position souhaitée .
(swriter/guide/ruler.xhp)

p.8) Si la hauteur de cet élément est supérieure à la taille de police utilisée, la ligne contenant l'élément est agrandie en conséquence.(aide/guide/anchor_object.xml)

P. 9) Vous pouvez utiliser le Navigateur pour déplacer des titres et le texte associé vers le haut ou le bas du document. (aide/guide/arrange_chapters.xml)

P. 10) Pour utiliser cette fonction, formatez les titres du document avec l'un des styles de paragraphe "Titre" prédéfinis. (aide/guide/arrange_chapters.xml)

P. 11) Pour modifier les options d’ancrage d’un élément, cliquez avec le bouton droit de la souris sur l’élément, puis choisissez une option dans le sous-menu ancrage. (aide/guide/anchor_object.xml)

P. 12) Certaines options de numérotation et de puces ne sont pas disponibles lorsque vous travaillez dans une mise en page Web. (aide/02/02110000.xml)

P. 13) Lorsque vous avez cliqué sur l'icône Aperçu : plusieurs pages, la boîte de dialogue Plusieurs pages s'ouvre. (aide/02/10070000.xml)

p.14) Pour centrer une image sur une page HTML , insérez l' image , ancrez-la "comme caractère " , puis centrez le paragraphe . (aide/guide/anchor_object.xml)

P. 15) Pour assigner un autre format d'heure, ou modifier les paramètres horaires actuels, choisissez Insertion - Champ - Autres et apportez les changements souhaités dans la boîte de dialogue Champs. (aide/02/18030200.xml)

P. 16) Lorsque l'icône Afficher/masquer les images de la barre Outils est activée, des cadres vides sont affichés comme substituants des images. (aide/02/18120000.xml)

P.17) « Elle n'est visible que lorsque vous placez le curseur dans le texte numéroté ou à puce. » (aide/02/06140000.xml)

II. Annexe 2 – Extrait de la terminologie du monde OpenOffice (commandes et entités)

(La terminologie complète comporte 2133 termes.)

Apparence

Automatique

Draw

Fichier

Fichier - Envoyer - Envoyer par e-mail

Fichier - Nouveau

Fichier - Ouvrir

Fichier – Aperçu avant impression

Format - Page - Bordures - Espacement avec le contenu

Format - Zones d'impression

Impress

Itérations

MAP-CERN, MAP-NCSA

Modifier - Entrer dans le groupe

Modifier - Groupe

Modifier - Quitter le groupe

Options

Précédent

Recherche

Table

OPEN-OFFICE Calc

OPEN-OFFICE Developer's Guide

OPEN-OFFICE Setup

OPEN-OFFICE Writer

(Dés)activer la numérotation

(Dés)activer le mode Plein écran

(Dés)activer le mode Ébauche

(Dés)activer les assistants

(Dés)activer les puces

III. Annexe 3 – Programme permettant de passer du xml au texte brut, tout en repérant les entités et en effectuant les prétraitements nécessaires à l'analyse syntaxique

```
# -*- coding: utf-8 -*-
import re, codecs, os, sys, string
import re, string

from BeautifulSoup import BeautifulSoup, Comment, BeautifulStoneSoup

#Ce script permet d'effectuer un tri global sur notre corpus brut, pour
ne garder que l'information exploitable par le parsing syntaxique.
#Il permet également le repérage des entités relatives au monde Open
Office et les identifie par des underscores pour empecher le parseur
syntaxique
#de les analyser comme une suite de mots.

#passage des regex
corpus = "corpus"
exprn=re.compile(r'( +[\r\n]+ +[\r\n]+)+')
exppoubelle=re.compile(r'\\+')
expespaces=re.compile(r'\s+')
expunder=re.compile(r'_+')
expmaj=re.compile(r'[A-Z]')
expchapeaux=re.compile(r'(\^\^\^)+')
exppoint=re.compile(r' \d\d*? \.*? \d\d*? \.*? \d\d*? \.*? |\d\d*? \.*?
\d\d*? \.*? \d\d*?|\d\d*? \.*? \d\d*? \.*?|\d\d*? \.*? \d\d*?')
expq = re.compile(r"((?<!\s[A-
ZÀÈÌÒÙÁÉÍÓÚÝÂÊÎÔÛÄËÏÖÜÃÑÕÆÅÐÇØ])\.\s|·)",re.UNICODE+re.M)
expi = re.compile(r'[Ii]ci')
splitter="====roxane===="


#passage des fichiers
interessantf=codecs.open("interessant.txt","w","utf-8")
anaphf=codecs.open("anaph.txt","w","utf-8")
soupf=codecs.open("soup.txt","w","utf-8")


outdir="corpuspropre_test"
if not os.path.exists(outdir): os.makedirs(outdir)
#for root, dirs, files in os.walk(corpus):
rempl={"%PRODUCTNAME":"OPEN-OFFICE","$[officename]":"OPEN-OFFICE","Ctrl
+":"Ctrl+","Ctrl+ ":"Ctrl+","Alt +":"Alt+"}
interes-
res-
sant=[u"objet",u"objets",u"Objets",u"Objet",u"icône",u"Icône",u"icônes"
,u"Icônes",u"bouton",u"Bouton",u"boutons",u"Boutons",u"mode
",u"Mode",u"modes",u"Modes",u"touche",u"Touche",u"touches",u"Touches",u
"champ",u"Champ",u"champs",u"Champs",u"barre",u"Barre",u"barres",u"Barr
es",u"onglet",u"Onglet",u"onglets",u"Onglets",u"boîte",u"Boîte",u"boîte
```

```
s",u"Boîtes",u"menu",u"Menu",u"menus",u"Menus",u"assistant",u"assistant
s",u"Assistant",u"Assistants",u"style",u"styles",u"Style",u"Styles",u"o
ption",u"options",u"Option",u"Options",u"fonction",u"fonctions",u"Fonct
ion",u"Fonctions",u"commande",u"commandes",u"Commande",u"Commandes"]


expcmd=re.compile(r' - ')
expcmd2=re.compile(r' – ')
expcmd3=re.compile(r'([A-Z]\w+-[A-Z]\w+)+')

entites=[]
entitf=codecs.open("entites_uniq_maj.txt","r+","utf-8")
for phrase in entitf:
        phrase=phrase.strip()
        phrase = re.sub('([\".,:;!?()]-)', r' \1 ', phrase)
        phrase = re.sub('\s{2,}', ' ', phrase)
      phrase = re.sub(r' \. xhp','.xhp',phrase)
      phrase=re.sub(r' \. doc','.doc',phrase)
      phrase=re.sub(r' \. docx','.docx',phrase)
      phrase=re.sub(r' \. txt','.txt',phrase)
      phrase=re.sub(r' \. pl','.pl',phrase)
      phrase=re.sub(r' \. zip','.zip',phrase)
      phrase=re.sub(r' \. jar','.jar',phrase)
      phrase=re.sub(r' \. jpg','.jpg',phrase)
      phrase=re.sub(r' \. gif','.gif',phrase)
      phrase=re.sub(r' \. png','.png',phrase)
      phrase=re.sub(r' \. rtf','.rtf',phrase)
      phrase=re.sub(r' \. xls','.xls',phrase)
      phrase=re.sub(r' \. xlsx','.xlsx',phrase)
      phrase=re.sub(r' \. odt','.odt',phrase)
      phrase=re.sub(r' \. ott','.ott',phrase)
      phrase=re.sub(r' \. ots','.ots',phrase)
      phrase=re.sub(r' \. otg','.otg',phrase)
      phrase=re.sub(r' \. otp','.otp',phrase)
      phrase=re.sub(r' \. odm','.odm',phrase)
      phrase=re.sub(r' \. oth','.oth',phrase)
      phrase=re.sub(r' \. odt','.odt',phrase)
      phrase=re.sub(r' \. ods','.ods',phrase)
      phrase=re.sub(r' \. odp','.odp',phrase)
      phrase=re.sub(r' \. org','.org',phrase)
      phrase=re.sub(r' \. oxt','.oxt',phrase)
      phrase=re.sub(r' \. ppt','.ppt',phrase)
      phrase=re.sub(r' \. dll','.dll',phrase)
      phrase=re.sub(r' \. xml','.xml',phrase)
      phrase=re.sub(r' \. xsl','.xsl',phrase)
      phrase=re.sub(r' \. html','.html',phrase)
      phrase=re.sub(r' \. htm','.htm',phrase)
      phrase=re.sub(r' \. asp','.asp',phrase)
      phrase=re.sub(r' \. com','.com',phrase)
      phrase=re.sub(r' \. csv','.csv',phrase)
      phrase=re.sub(r' \. lst','.lst',phrase)
      phrase=re.sub(r' \. jsp','.jsp',phrase)
      phrase=re.sub(r'www \. ','www.',phrase)
      phrase=re.sub(r'http : //','http://',phrase)
      phrase=re.sub(r'mysql : //','mysql://',phrase)
      phrase=re.sub(r'file : ///','file:///',phrase)
      phrase=re.sub(r'otn \. ','otn.',phrase)
      phrase=re.sub(r'oracle \. ','oracle.',phrase)
```

```
      phrase=re.sub(r'java \. ','java.',phrase)
      phrase=re.sub(r'C : ','C:',phrase)
      phrase=re.sub(r"c'","c' ",phrase)
      phrase=re.sub(r"d'","d' ",phrase)
      phrase=re.sub(r"l'","l' ",phrase)
      phrase=re.sub(r"m'","m' ",phrase)
      phrase=re.sub(r"n'","n' ",phrase)
      phrase=re.sub(r"s'","s' ",phrase)
      phrase=re.sub(r"t'","t' ",phrase)
      phrase=expcmd.sub("-",phrase)
      phrase=expcmd2.sub("-",phrase)
        if phrase:      entites+=[phrase]
entitf.close()

def nettoie(s):
      #print s
      undere=re.compile(r"(_[^ ]*?) ",re.U)
      s=undere.sub(r'"\1" ',s)
      return s.replace("  "," ").replace(" ,",",").replace("- ","-
").replace("_"," ").replace("  ","
").replace("( ","(").replace(" )",")").replace(" .",".").replace("'
","'")

def bonnesoupe(soup):
      recette=re.compile(r"\b(ce|cette|ces|cet)\b")
      for para in
soup.findAll(text=re.compile(r"\b(ce|cette|ces|cet)\b")):
            tag=para.parent

            good =
tag.previousSibling.previousSibling.contents[0].contents[0].contents[0]

            m = recette.search(para)
            #print para[:m.start()],"_______",para[m.end():]
            b=re.compile(r"(\W+)",re.U)
            motsapres=b.split(para[m.end():]) #.split()
            #print motsapres
            #1/0
            dett=para[m.start():m.end()]
            if dett=="cette": det="la"
            elif dett=="ce": det="le"
            elif dett=="cet":     det="l'"

            novstring=nettoie(para[:m.start()]+det+"
".join(motsapres[:3])+" "+good+" "+" ".join(motsapres[3:]))

            print novstring
            print "\n"
            print tag
            para.replaceWith(novstring)
            #tag.replaceWith(recette.sub(novstring,para))
            #print "hhhhhhhh",tag
            #print soup
      return soup


compteur=0
for root, dirs, files in os.walk(corpus):
```

```
        for d in dirs:
            if not os.path.exists(outdir+"/"+root+"/"+d):
os.makedirs(outdir+"/"+root+"/"+d)


        for filename in files:
            filename=root+"/"+filename
            compteur+=1
            if unicode(filename).endswith("~"):continue
            #filename="corpus/swriter/main0110.xhp"
            ##print filename,root, dirs

            f = codecs.open(filename,"r","utf-8")
            texte=f.read()
            soup=BeautifulStoneSoup(texte, convertEnti-
ties=BeautifulStoneSoup.HTML_ENTITIES)

            soup3=bonnesoupe(soup)

            soup3=BeautifulStoneSoup(texte, convertEnti-
ties=BeautifulStoneSoup.HTML_ENTITIES)


            comments = soup3.findAll(text=lambda text:isinstance(text,
Comment))
            [comment.extract() for comment in comments]
            subtree = soup3.history
            if subtree:subtree.extract()
            subtree = soup3.filename
            if subtree:subtree.extract()
            subtree = soup3.comment
            if subtree:subtree.extract()


            ##########" ici on passe la soupe en texte
            texte=' '.join([e for e in soup3.recursiveChildGenerator()
if isinstance(e,unicode)])
            #print texte


            ts=re.findall(expi,texte)
            if ts:
            for i in ts:
                 soupf.write(filename+"\n"+i)


            texte=exppoubelle.sub("",texte)
            #texte=expespaces.sub(" ",texte)
            texte=exprn.sub("\n",texte)

            out=codecs.open(outdir+"/"+filename+".txt","w","utf-8")

            for ligne in texte.split("\n"):
                ligne=ligne.strip()

                if ligne.startswith("<"): #"&lt;" in ligne:
                    aregarder=codecs.open("aRegarder.txt","a","utf-8")
```

```
                    aregard-
er.write(filename+"::::::::::::::::"+ligne+"\n")
                    aregarder.close()
                elif ligne.startswith(u"xml version") or
ligne.startswith(u"UFI") or ligne==u"Icône" or ligne==u"icône":
                    continue
                elif ligne:
                    for r in rempl:
                        if r in ligne:
                                ligne=ligne.replace(r,rempl[r])
                    ligne = expq.sub(r"\1"+splitter,ligne)


                    for phrase in ligne.split(splitter):
                        phrase=phrase.strip()
                        phrase = re.sub('([\".,:;!?()])', r' \1 ',
phrase)
                        phrase = re.sub('\s{2,}', ' ', phrase)
                        phrase = re.sub(r' \. xhp','.xhp',phrase)
                        phrase=re.sub(r' \. doc','.doc',phrase)
                        phrase=re.sub(r' \. docx','.docx',phrase)
                        phrase=re.sub(r' \. txt','.txt',phrase)
                        phrase=re.sub(r' \. pl','.pl',phrase)
                        phrase=re.sub(r' \. zip','.zip',phrase)
                        phrase=re.sub(r' \. jar','.jar',phrase)
                        phrase=re.sub(r' \. jpg','.jpg',phrase)
                        phrase=re.sub(r' \. gif','.gif',phrase)
                        phrase=re.sub(r' \. png','.png',phrase)
                        phrase=re.sub(r' \. rtf','.rtf',phrase)
                        phrase=re.sub(r' \. xls','.xls',phrase)
                        phrase=re.sub(r' \. xlsx','.xlsx',phrase)
                        phrase=re.sub(r' \. odt','.odt',phrase)
                        phrase=re.sub(r' \. ott','.ott',phrase)
                        phrase=re.sub(r' \. ots','.ots',phrase)
                        phrase=re.sub(r' \. otg','.otg',phrase)
                        phrase=re.sub(r' \. otp','.otp',phrase)
                        phrase=re.sub(r' \. odm','.odm',phrase)
                        phrase=re.sub(r' \. oth','.oth',phrase)
                        phrase=re.sub(r' \. odt','.odt',phrase)
                        phrase=re.sub(r' \. ods','.ods',phrase)
                        phrase=re.sub(r' \. odp','.odp',phrase)
                        phrase=re.sub(r' \. org','.org',phrase)
                        phrase=re.sub(r' \. oxt','.oxt',phrase)
                        phrase=re.sub(r' \. ppt','.ppt',phrase)
                        phrase=re.sub(r' \. dll','.dll',phrase)
                        phrase=re.sub(r' \. xml','.xml',phrase)
                        phrase=re.sub(r' \. xsl','.xsl',phrase)
                        phrase=re.sub(r' \. html','.html',phrase)
                        phrase=re.sub(r' \. htm','.htm',phrase)
                        phrase=re.sub(r' \. asp','.asp',phrase)
                        phrase=re.sub(r' \. com','.com',phrase)
                        phrase=re.sub(r' \. csv','.csv',phrase)
                        phrase=re.sub(r' \. lst','.lst',phrase)
                        phrase=re.sub(r' \. jsp','.jsp',phrase)
                        phrase=re.sub(r'www \. ','www.',phrase)
                        phrase=re.sub(r'http : //','http://',phrase)
                        phrase=re.sub(r'mysql : //','mysql://',phrase)
                        phrase=re.sub(r'file : ///','file:///',phrase)
```

```
                        phrase=re.sub(r'otn \. ','otn.',phrase)
                        phrase=re.sub(r'oracle \. ','oracle.',phrase)
                        phrase=re.sub(r'java \. ','java.',phrase)
                        phrase=re.sub(r'C : ','C:',phrase)
                        phrase=re.sub(r"c'","c' ",phrase)
                        phrase=re.sub(r"d'","d' ",phrase)
                        phrase=re.sub(r"l'","l' ",phrase)
                        phrase=re.sub(r"m'","m' ",phrase)
                        phrase=re.sub(r"n'","n' ",phrase)
                        phrase=re.sub(r"s'","s' ",phrase)
                        phrase=re.sub(r"t'","t' ",phrase)
                        phrase=expcmd.sub("-",phrase)
                        phrase=expcmd2.sub("-",phrase)
                        phrase=exppoint.sub(string.whitespace,phrase)

                        for inter in interessant:
                                if inter in phrase:

                                        inte-
ressantf.write(filename+":::"+phrase+"\n")
                                        liste_mots=phrase.split()
                                        if inter in liste_mots:
                                              entp=liste_mots.index(inter)
                                              ent1=liste_mots[entp]

                                              n = len(liste_mots)

                                              if n <=  abs(entp+1):
                                                    entok="".join(ent1)

phrase=phrase.replace(entok,"_"+entok.replace(" ","_"))


                                              elif -n <= entp+1 < n: # same as
<= n-1, but avoids subtraction
                                              #if len(phrase) >= abs(entp+1):
                                                    ent2=liste_mots[entp+1]
                                                    if ent2!="de":
                                                          if ent2=="." or
ent2==";" or ent2=="," or ent2=="!" or ent2=="?" or ent2=="(" or
ent2==")" or ent2=="/" or ent2=="\\":
                                                                en-
tok="".join(ent1)

phrase=phrase.replace(entok,"_"+entok.replace(" ","_"))
                                                          elif ent2[0]=='A' or
ent2[0]=='B'or ent2[0]=='C' or ent2[0]=='D' or ent2[0]=='E' or
ent2[0]=='F'  or ent2[0]=='G' or ent2[0]=='H' or ent2[0]=='I' or
ent2[0]=='J' or ent2[0]=='K' or ent2[0]=='L' or ent2[0]=='M' or
ent2[0]=='N' or ent2[0]=='O' or ent2[0]=='P' or ent2[0]=='Q' or
ent2[0]=='R' or ent2[0]=='S' or ent2[0]=='T' or ent2[0]=='U' or
ent2[0]=='V' or ent2[0]=='W' or ent2[0]=='X' or ent2[0]=='Y' or
ent2[0]=='Z':

                                                                en-
tok="".join(ent1+" "+ent2)

phrase=phrase.replace(entok,"_"+entok.replace(" ","_"))
```

```
                                                      else:
                                                            en-
tok="".join(ent1)

phrase=phrase.replace(entok,"_"+entok.replace(" ","_"))

                                                elif ent2=="de" or
ent2=="d'" or ent2=="du" or ent2=="des":
                                                      if -n <= entp+2 < n:
                                                      #if len(phrase) >=
abs(entp+3):

ent3=liste_mots[entp+2]
                                                            en-
tok="".join(ent1+" "+ent2+" "+ent3)

phrase=phrase.replace(entok,"_"+entok.replace(" ","_"))
                                                else:
                                                      continue

                                          else:
                                                continue



                        for en in entites:
                              if en in phrase:

      phrase=phrase.replace(en,"^^^"+en.replace(" ","^^^"))
                        phrase=expchapeaux.sub("_",phrase)
                        phrase=expunder.sub("_",phrase)


                        out.write(phrase+"\n")

            out.close()
            #print compteur,"files done:",filename

interessantf.close()
```

IV. Annexe 4 – Programme permettant de lancer le parseur syntaxique sur notre corpus

```
# -*- coding: utf-8 -*-
import os, sys, re, codecs, glob, unicodedata, subprocess, glob
from operator import itemgetter
from heapq import nlargest
from optparse import OptionParser
import pickle,marshal, time, re

#cette fonction prend 2 arguments, le fichier de phrases en entrée et
le dossier de sortie
def mate(sentencefile, outfolder="."):

      #p1 = subprocess.Popen(["pwd"], stdout=subprocess.PIPE)
      #print p1.stdout.read()

      sentencefile=unicode(sentencefile)
      #création du dossier de sortie
      if outfolder[-1]!="/":outfolder=outfolder+"/"

      #nom du dossier de sortie suivi du nom du fichier de phrases
      filebase=outfolder+os.path.basename(sentencefile)


      ########### français

      #on prend en entrée le fichier à parser et on le met dans le dos-
sier de sortie préalablement crée
      p1 = subprocess.Popen(["java -cp anna-2.jar is2.util.Split
"+sentencefile+" > "+filebase+"-one-word-per-line.txt"],shell=True,
stdout=subprocess.PIPE)
      print p1.stdout.read()
      p1 = subprocess.Popen(["java -Xmx2G -cp anna-2.jar
is2.lemmatizer.Lemmatizer -model ftp-lemmatizer.model -test
"+filebase+"-one-word-per-line.txt -out   "+filebase+"-
lemmatized.txt"],shell=True)
      out, err = p1.communicate()
      print out, err
      p1 = subprocess.Popen(["java -Xmx2G -cp anna-2.jar is2.tag3.Tagger
-model ftp-tagger_cpos.model -test  "+filebase+"-lemmatized.txt -out
"+filebase+"-tagged.txt"],shell=True,  stdout=subprocess.PIPE)
      out, err = p1.communicate()
      print out, err
      p1 = subprocess.Popen(["java -Xmx2G -classpath anna-2.jar
is2.parser.Parser -model ftp-parser_cpos.model -test  "+filebase+"-
tagged.txt -out  "+filebase+"-dependently.txt"],shell=True,
stdout=subprocess.PIPE)
      print p1.stdout.read()

      ################
def parse(infolder, outfolder):
      if not os.path.exists(infolder): raise WrongDirectoryError
      if not os.path.exists(outfolder): os.makedirs(outfolder)
```

```
	for infile in glob.glob(os.path.join(infolder, '*.*')):
		if unicode(infile).endswith("~"):continue
		print "parsing",os.path.basename(infile)
		mate(infile,outfolder)


if __name__ == "__main__":

	#corpus oo
	#for root, dirs, files in os.walk(corpus):
	#for filename in files:
		#if not os.path.exists(newcorpus+"/"+infile):
os.makedirs(newcorpus+"/"+infile)

	#corpus="corpuspropre/corpus/"
	#newcorpus="corpusparsed/corpuspropre/corpus/"

	#dirs=["swriter","swriter/00","swriter/01","swriter/02","swriter/g
uide","shared","shared/00","shared/01","shared/02","shared/04","shared/
05","shared/07","shared/autokorr","shared/autopi","shared/explorer/data
base","shared/guide,shared/optionen"]


	#for d in dirs:
		#infolder=corpus+d
		#outfolder=newcorpus+d

		#parse(infolder,outfolder)

	#corpus causations

	infolder="corpuspropre"
	outfolder="corpusparsed"

	parse(infolder,outfolder)
```

V. Annexe 5 – Programme permettant d'extraire les causations

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys,cgi,cgitb,json, codecs, re
import glob, os

#from config import *


def conllfile2tree(path):
	"""
	takes a path to a conll 14 file
	gives back a list of dictionaries of nodes of the form
	{1 : { "word":"blah", ... }, 2: {...}, ... }
	"""
```

```
      f=codecs.open(path,"r","utf-8")
      #string
      tree=""
      #int
      sentencecount=0
      #dic
      nodedic={}
      #lst
      nodediclist=[]

      #pr ttes les lignes dans le fichier, on enlève espaces à gauche &
à droite
      for li in f:
            li=li.strip()

            if li:     tree+=li+"\n" # si il existe une ligne on l'add à
l'arbre
            elif tree: # si ligne vide et j'ai déjà trouvé un arbre
(pour éviter des problèmes liés au doubles passage à la ligne)

                  #ttes les lignes d'un arbre = la phrase, mais une ligne
= le mot
                  for line in tree.split('\n'):
                        #print line
                        if line.strip():
                              #cellules conll du mot
                              cells = line.split('\t')
                              nrCells = len(cells)

                              if nrCells == 14:
                                    #on donne un nom aux cellules
                                    nr, word, lemma, lemma2, _, tag, _, _,
head, _, rel, _, _, _ = cells
                                    #certaines cellules sont des entiers
                                    try:
                                          nr,head = int(nr), int(head)
                                    except ValueError:
                                                pass
                                    if lemma2!="_": lemma=lemma2 # TODO:
find out what the difference between these colons is!!!!!!
                                    #on donne un nom aux cellules

                                    #get permet de retrouver la valeur de
la cle deps
                                    #deps nous permet de retrouver les
dépendants
                                    nodedic[nr]={'id':nr,'word':
word,'lemma': lemma, 'tag': tag, 'head':head, 'rel':
rel,'deps':nodedic.get(nr,{}).get('deps',[])}

                                    #la méthode get permet de retrouver la
valeur d'une clé donnée, ici on veut retrouver la valeur de la clé
"head" et si pas de clé head on met rien dans le dic
                                    hdic = nodedic.get(head,{})
                                    hdic['deps']=hdic.get('deps',[])+[nr]
                                    nodedic[head]=hdic
                                    #print hdic,nodedic
                  sentencecount+=1
```

```
                  #un nodedic représente une phrase
                  nodediclist+=[nodedic]
                  nodedic={}
                  tree=""

            if not li: sentencecount+=1
      f.close()
      return nodediclist


def getDependents(nodedict,nodenum,deplist=[],cutat=[],cutpoints=[]):
      """takes
      a node,x
      its number,
      the list of its dependents that should in any case be in the list
(usually empty)

      cutat is a list of relations that should not be followed down the
tree
      cutpoints is a list of indeces which should not be included in the
subtree

      returns deplist = list of number of dependents
         """
      #print "______",nodenum,deplist
      node = nodedict[nodenum]
      #if cutpoints: print node["rel"] in cutat , node["address"] in
cutpoints, node["address"] , cutpoints, deplist
      if node["rel"] in cutat or node["id"] in cutpoints: return deplist

      deplist+=[nodenum]
      for childnum in node["deps"]:
            getDependents(nodedict,childnum,deplist,cutat,cutpoints)
      return deplist

def leffe(list):
      verblefff=codecs.open('vlefff.txt','r','utf-8')
      ld = {}
      count=0
      for line in verblefff:
            par = line.strip().split("\t")
            if len(par)!=3:
                  pass
                  #print line
            else:
                  ld[par[0]]=(par[1],par[2])
                  count+=1

      #pasmot=re.compile("\W",re.U)
      #mots = pasmot.split(list)

      mymot=[]
      for mot in list:
            res = ld.get(mot.lower(),None)
            #print res

            if res and "2p" in res[1]:
                  mot = res[0]
```

```
            elif res and "G" in res[1]:
                  mot = res[0]
            mymot+=[mot]

      resuleffe=" ".join(mymot)
      return resuleffe

def causeaeffet(nodedict):

      resultats=[]

      nodedict2={}
      nodedict3={}
      allnodes=[]
      sentence=[]
      #pr chaque cell dans nodedict
      for address in nodedict:
            allnodes+=[nodedict[address].get("word","")]
      #on exclut d'office toutes les phrases comportant le verbe "pou-
vez" : elles ne nous intéressent pas ici
      if "pouvez" in allnodes:
            #pass
            pourdedans,lorsquededans, pouvezdedans, endedans=
False,False,False,False
            for id in nodedict:

                  numhead=nodedict[id].get("head","")
                  if nodedict[id].get("word","").lower()=="pouvez" and
nodedict[id].get("rel","") in ["dep"] and
nodedict[numhead].get("word","").lower()=="vous" and
nodedict[numhead].get("rel","") in ["suj","aff"] and
nodedict[nodedict[numhead].get("head",0)].get("rel","")=="root":
                        if "lorsque" in
[ nodedict[i].get("word","") for i in
nodedict[nodedict[numhead].get("head",0)]["deps"] ]  :

                              cas="'lorsque pouvez' faux"
                              #le noeud a la fonction "dep" et se
nomme "pouvez"
                              # son gouverneur est égal à vous et a
la fonction suj ou aff
                              # le grand-père a la fonction "root"
                              # "lorsque" est dans les dépendants de
"root"


      grandpere=nodedict[numhead].get("head",0)
                              lorsqueindex = [i  for i in
nodedict[grandpere]["deps"] if
nodedict[i].get("word","")=="lorsque" ][0]

                              causehead= grandpere # je prends le
premier dep du noeud courant qui est un verbe
                              causelist= sort-
ed(getDependents(nodedict,causehead,[],[],cutpoints=[numhead,lorsqueind
ex]))
```

```
                                effethead= [i  for i in
nodedict[grandpere]["deps"] if
nodedict[i].get("word","")=="lorsque" ][0]# je prends le premier dep du
grand-pere égal à "en"
                                effetlist= sort-
ed(getDependents(nodedict,effethead,[],[],cutpoints=[]))
                                effetlist=effetlist[2:]
                                phrase = "
".join( [  nodedict[i]["word"]  for i in range(1,len(nodedict))] )

                                resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                                #print resultats
                                break

                if nodedict[id].get("word","").lower()=="lorsque" and
nodedict[id].get("rel","")=="mod" and
nodedict[numhead].get("word","")=="pouvez" and
nodedict[numhead].get("rel","")=="root" and "V" in
[ nodedict[i].get("tag","") for i in nodedict[numhead]["deps"] ]   :
                            cas="'lorsque pouvez' correct"
                            #le noeud a la fonction "mod" et se nomme
"lorsque"
                            # son gouverneur est égal "pouvez" et il a
la fonction root
                            # il y a un dép du gouverneur "pouvez" qui
est un verbe
                            causehead= [ i  for i in
nodedict[numhead]["deps"] if nodedict[i].get("tag","")=="V" ] [0] # je
prends le premier dep du head qui est un verbe
                            causelist= sort-
ed(getDependents(nodedict,causehead,[],[]))
                            effethead= [ i  for i in
nodedict[id]["deps"] if nodedict[i].get("tag","")=="V" ] [0] # je
prends le premier dep du id qui est un verbe
                            effetlist= sort-
ed(getDependents(nodedict,effethead,[],[],cutpoints=[id+1]))
                            phrase = " ".join( [  nodedict[i]["word"]
for i in range(1,len(nodedict))] )

                            resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                            #print resultats
                            break

                if nodedict[id].get("word","").lower()=="en" and
nodedict[id].get("rel","")=="mod" and
nodedict[nodedict[id]["deps"][0]].get("word","").lower().endswith("ant"
) and nodedict[numhead].get("word","")=="pouvez" and
nodedict[numhead].get("rel","")=="root" and "V" in
[ nodedict[i].get("tag","") for i in nodedict[numhead]["deps"] ]   :


                        cas="'participepresent pouvez' correct"
                        #le noeud a la fonction "mod" et se nomme "en"
                        #le mot qui suit est un verbe au participe-
present
```

```
                        # son gouverneur est égal "pouvez" et il a la
fonction root
                        # il y a un dép du gouverneur qui est un verbe
                        mainv= [numhead]["deps"][0]
                        print mainv
                        causehead= [ i  for i in
nodedict[numhead]["deps"] if nodedict[i].get("tag","")=="V" ] [0] # je
prends le premier dep du head qui est un verbe
                        causelist= sort-
ed(getDependents(nodedict,causehead,[],[]))
                        effethead= [ i  for i in nodedict[id]["deps"] if
nodedict[i].get("tag","")=="V" ] [0] # je prends le premier dep du id
qui est un verbe
                        effetlist= sort-
ed(getDependents(nodedict,effethead,[],[],cutpoints=[id+1]))
                        phrase = " ".join( [  nodedict[i]["word"]  for i
in range(1,len(nodedict))] )

                        resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                        break

                if nodedict[id].get("word","").lower()=="pouvez" and
nodedict[id].get("rel","") in ["dep"] and
nodedict[numhead].get("word","").lower()=="vous" and
nodedict[numhead].get("rel","") in ["suj","aff"] and
nodedict[nodedict[numhead].get("head",0)].get("rel","")=="root":
                            if "en" in [ nodedict[i].get("word","") for
i in nodedict[nodedict[numhead].get("head",0)]["deps"] ]:
                                #le noeud a la fonction "dep" et se
nomme "pouvez"
                                # son gouverneur est égal à vous et a
la fonction suj ou aff
                                # le grand-père a la fonction "root"
                                # "en" est dans les dépendants de
"root"

     grandpere=nodedict[numhead].get("head",0)
                                enindex = [i  for i in
nodedict[grandpere]["deps"] if nodedict[i].get("word","")=="en" ][0]
                                antindex=nodedict[enindex]["deps"][0]
                                if "ant" in
nodedict[antindex].get("word",""):
                                    cas="'participepresent pouvez'
faux"
                                    causehead= enindex # je prends
le premier dep du noeud courant qui est un verbe
                                    causelist= sort-
ed(getDependents(nodedict,causehead,[],[],cutpoints=[]))

                                    effethead= grandpere# je prends
le premier dep du grand-pere égal à "en"
                                    effetlist= sort-
ed(getDependents(nodedict,effethead,[],cutpoints=[enindex,numhead]))
                                    phrase = "
".join( [  nodedict[i]["word"]  for i in range(1,len(nodedict))] )
```

```
                                        resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                                        #print resultats
                                        break

                if nodedict[id].get("word","").lower()=="pour" and
nodedict[id].get("rel","")=="mod" and
nodedict[numhead].get("word","")=="pouvez" and
nodedict[numhead].get("rel","")=="root" and  "V" in
[ nodedict[i].get("tag","") for i in nodedict[id]["deps"] ]  and "V" in
[ nodedict[i].get("tag","") for i in nodedict[numhead]["deps"] ]   :
                        cas="'pour pouvez' correct"
                        #le noeud a la fonction "mod" et se nomme "pour"
                        # son gouverneur est égal "pouvez" et il a la
fonction root
                        # il y a un dép de mon noeud qui est un verbe
                        # il y a un dép du gouverneur qui est un verbe


                        causehead= [ i  for i in
nodedict[numhead]["deps"] if nodedict[i].get("tag","")=="V" ] [0] # je
prends le premier dep du head qui est un verbe
                        causelist= sort-
ed(getDependents(nodedict,causehead,[],[]))
                        effethead= [ i  for i in nodedict[id]["deps"] if
nodedict[i].get("tag","")=="V" ] [0] # je prends le premier dep du id
qui est un verbe
                        effetlist= sort-
ed(getDependents(nodedict,effethead,[],[]))
                        phrase = " ".join( [  nodedict[i]["word"]  for i
in range(1,len(nodedict))] )

                        resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                        break

                if nodedict[id].get("word","").lower()=="pouvez" and
nodedict[id].get("rel","") in ["dep"] and
nodedict[numhead].get("word","").lower()=="vous" and
nodedict[numhead].get("rel","") in ["suj","aff"] and
nodedict[nodedict[numhead].get("head",0)].get("rel","")=="root":
                        if "pour" in [ nodedict[i].get("word","") for i
in nodedict[nodedict[numhead].get("head",0)]["deps"] ]  :
                                # je veux : un noeud "pouvez" avec la fonc-
tion dep,
                                # son gouverneur est égal "vous" et il a la
fonction suj ou aff
                                # nodedic[nodedict[numhead].get("head",0)]
est le grand père est il a la fonction root
                                      #et le grand père a dans ses dépen-
dants le mot "pour"
                                cas="'pour pouvez' faux"

                                grandpere=nodedict[numhead].get("head",0)
                                pourindex = [i  for i in
nodedict[grandpere]["deps"] if nodedict[i].get("word","")=="pour" ][0]
                                if "V" in [nodedict[i].get("tag","") for i
in nodedict[pourindex]["deps"]]:
```

```
                                    #print "bien"
                                    causehead= grandpere # je prends le
premier dep du head qui est un verbe

                                    causelist= sort-
ed(getDependents(nodedict,causehead,[],[],[numhead,pourindex]))
                                    effethead= [ i  for i in
nodedict[pourindex]["deps"] if nodedict[i].get("tag","")=="V" ] [0] #
je prends le premier dep du pour qui est un verbe
                                    effetlist= sort-
ed(getDependents(nodedict,effethead,[],[]))
                                    phrase = "
".join( [  nodedict[i]["word"]  for i in range(1,len(nodedict))] )
                                    #print
phrase,nodedict,causehead,[],[numhead,pourindex],causelist

                                    resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]

                        elif "soit" in [ nodedict[i].get("word","") for i
in nodedict[nodedict[numhead].get("head",0)]["deps"] ]  :
                                    #et le grand père a dans ses dépen-
dants le mot "soit"
                                cas="'pour pouvez alternative'"

                                grandpere=nodedict[numhead].get("head",0)

                                #index du 1er "soit"
                                soitindex1 = [i  for i in
nodedict[grandpere]["deps"] if nodedict[i].get("word","")=="soit" ][0]
                                #index du 2nd "soit"
                                soitindex2 = [i  for i in
nodedict[grandpere]["deps"] if nodedict[i].get("word","")=="soit" ][1]
                                #index du 1er "pour", fils de "soit"
                                pourindex1 =
nodedict[soitindex1].get("deps",0)[0]

                                #index du 2nd "pour", fils de "soit"
                                pourindex2 =
nodedict[soitindex2].get("deps",0)[0]

                                causehead= grandpere

                                #on crée une cause de laquelle on enlève
les sous-arbres soitindex1 & soitindex2 & numhead (tête du mot courant)
                                causelist= sort-
ed(getDependents(nodedict,causehead,[],[],[numhead,soitindex1,soitindex
2]))

                                #on cherche dans les deps du 1er "pour" un
verbe et si c le cas, on crée une cause et un effet
                                if "V" in [nodedict[i].get("tag","") for i
in nodedict[pourindex1]["deps"]]:

                                    effethead= [ i  for i in
nodedict[pourindex1]["deps"] if nodedict[i].get("tag","")=="V" ] [0]
                                    #ex : imprimer les lettres
```

```
                                    effetlist1=
sorted(getDependents(nodedict,effethead,[],[]))

                                #on cherche dans les deps du 2nd "pour" un
verbe et si c le cas, on crée une cause et un effet
                                if "V" in [nodedict[i].get("tag","") for i
in nodedict[pourindex2]["deps"]]:

                                    effethead= [ i  for i in
nodedict[pourindex2]["deps"] if nodedict[i].get("tag","")=="V" ] [0]
                                    #ex : les envoyer par mail
                                    effetlist2=
sorted(getDependents(nodedict,effethead,[],[]))

                                # on rassemble les 2 effets possibles en
rajoutant l'alternative
                                #effetlist="pour "+effetlist1+" ou pour
"+effetlist2

                                effetlist=effetlist1+effetlist2
                                    # je prends le premier dep du pour qui
est un verbe

                                phrase = " ".join( [  nodedict[i]["word"]
for i in range(1,len(nodedict))] )
                                    #print
phrase,nodedict,causehead,[],[numhead,pourindex],causelist

                                resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                                #print resultats
                                break




            if nodedict[id].get("word","").lower()=="pour":
                  pourdedans=True
            if nodedict[id].get("word","").lower()=="lorsque":
                  lorsquededans=True
            if nodedict[id].get("word","").lower()=="pouvez":
                  pouvezdedans=True
            if nodedict[id].get("word","").lower()=="en":
                  endedans=True
            if pourdedans and pouvezdedans:
                                phrase = " ".join( [  nodedict[i]["word"]
for i in range(1,len(nodedict))] )
                                cas="poubellepourpouvez"
                                resultats+=[ ( (0,[]),(0,[]), phrase,
cas ) ]
                                break
            elif lorsquededans and pouvezdedans:
                                phrase = " ".join( [  nodedict[i]["word"]
for i in range(1,len(nodedict))] )
                                cas="poubellelorsquepouvez"
                                resultats+=[ ( (0,[]),(0,[]), phrase,
cas ) ]
```

```
                              break
                    elif endedans and pouvezdedans:
                              phrase = " ".join( [  nodedict[i]["word"]
for i in range(1,len(nodedict))] )
                              cas="poubellepppouvez"
                              resultats+=[ ( (0,[]),(0,[]), phrase,
cas ) ]
                              break

     elif "pouvez" not in allnodes and "vous" not in allnodes and
"Vous" not in allnodes and "auquel" not in allnodes and "par" not in
allnodes and "Par" not in allnodes and "puis" not in allnodes and
"_Raccourcis_clavier" not in allnodes:

           nodedict3=nodedict
           #print nodedict3
           for id in nodedict3:

                num=range(1,len(nodedict3))
                #tête du noeud courant
                numhead=nodedict3[id].get("head","")
                #index=nodedict2[id]

                if nodedict3[id].get("rel","")=="mod" and
nodedict3[id].get("word","").lower()=="pour" and
nodedict3[numhead].get("rel","")=="root"  and  "V" in
[ nodedict3[i].get("tag","") for i in nodedict3[id]["deps"] ]:
                          # je veux : un noeud "pour" de type "mod"
                          # la tête de pour est racine
                          #pour a dans ses dépendants un verbe
                          cas="'pour' correct"
                          causehead= numhead # je prends le head, tête de
la cause
                          #création d'une causelist, en excluant le sous-
arbre "pour"
                          #on veut aussi exclure ts les ss-arbres de type
"puis", "par exemple",
                          causelist= sort-
ed(getDependents(nodedict3,causehead,[],cutpoints=[id]))
                          effethead= [ i  for i in nodedict3[id]["deps"] if
nodedict3[i].get("tag","")=="V" ] [0] # je prends le premier dep du id
qui est un verbe # je prends le head de l'effet, qui est le 1er verbe
de "pour", noeud courant
                          effetlist= sort-
ed(getDependents(nodedict3,effethead,[],[]))
                          phrase = " ".join( [  nodedict3[i]["word"]  for i
in range(1,len(nodedict3))] )

                          resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]


                          break

     elif "pouvez" not in allnodes and "Appuyez sur Échap" not in
allnodes:
           nodedict2=nodedict
           for id in nodedict2:
```

```
                num=range(1,len(nodedict2))
                ##tête du noeud courant
                numhead=nodedict2[id].get("head","")

                if nodedict2[id].get("rel","")=="mod" and
nodedict2[id].get("word","").lower()=="lorsque" and
nodedict2[numhead].get("rel","")=="root" and  "V" in
[ nodedict2[i].get("tag","") for i in nodedict2[id]["deps"] ]:
                        # je veux : un noeud "lorsque" de type "mod"
                        # la tête de "lorsque" est racine
                        #"lorsque" a dans ses dépendants un verbe
                        cas="'lorsque' correct"

                        causehead= [ i  for i in nodedict2[id]["deps"] if
nodedict2[i].get("tag","")=="V" ] [0] # je prends le premier dep du id
qui est un verbe # je prends le head de l'effet, qui est le 1er verbe
de "pour", noeud courant
                        suj= nodedict2[causehead].get("deps","")[0]
                        causelist= sort-
ed(getDependents(nodedict2,causehead,[],cutpoints=[suj]))

                        effethead= numhead # je prends le head, tête de
l'effet
                        #création d'une effetlist, en excluant le sous-
arbre "lorsque"
                        effetlist= sort-
ed(getDependents(nodedict2,effethead,[],[],cutpoints=[id]))



                        phrase = " ".join( [  nodedict2[i]["word"]  for i
in range(1,len(nodedict2))] )

                        resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]
                        #print resultats
                        break

                if nodedict2[id].get("rel","")=="mod" and
nodedict2[id].get("word","").lower()=="en" and
nodedict2[nodedict2[id]["deps"][0]].get("word","").lower().endswith("an
t") and nodedict2[nodedict2[id]["deps"][0]].get("tag","")=="V" and
nodedict2[numhead].get("rel","")=="root" :
                        # je veux : un noeud "en" de type "mod" dont le
1er dep se termine en -ant et est un verbe
                        # la tête de "lorsque" est racine
                        #"lorsque" a dans ses dépendants un verbe
                        cas="'participepresent' correct"
                        #effetlistok=[]
                        causehead= node-
dict2[nodedict2[id]["deps"][0]].get("id","") # je prends le 1er dep du
noeud courant qui est un verbe
                              # je prends le head, tête de l'effet
                        causelist= sort-
ed(getDependents(nodedict2,causehead,[],[],cutpoints=[]))

                        effethead= numhead
```

```
                        effetlist= sort-
ed(getDependents(nodedict2,effethead,[],[],cutpoints=[id]))
                        effetlistok=effetlist[1:]
                        #print effetlistok

                        phrase = " ".join( [  nodedict2[i]["word"]  for i
in range(1,len(nodedict2))] )
                        resultats+=[ ( (causehead,
causelist),(effethead,effetlist), phrase, cas ) ]

                        #print resultats
                        break
      return resultats


def nettoie(s):
      #print s
      undere=re.compile(r"(_[^ ]*?) ",re.U)
      s=undere.sub(r'"\1" ',s)
      return s.replace("  "," ").replace(" ,",",").replace("- ","-
").replace("_"," ").replace("  ","
").replace("( ","(").replace(" )",")").replace(" .",".").replace("'
","'")

def sortirCauseEffet(fichierorigine, causeeffetsetc, phrase, cas, fi-
chiersortie):
      fichiersortie.write("_________\n"+fichierorigine+"\n")
      fichiersortie.write("cas: "+cas+"\n")
      fichiersortie.write(nettoie(phrase)+"\n")
      for i,truc in enumerate(causeeffetsetc):
            fichiersortie.write(str(i)+":::"+nettoie(truc)+"\n")




if __name__ == "__main__": # what is it exactly ???
      reponct=re.compile(r",.")



      corpus="corpusparsed"

      Pour_resultList=codecs.open('causations_test/Pour_resultList.txt',
'w','utf-8')


      count=0
      for root, dirs, files in os.walk(corpus):
            for filename in files:
                  if "dependently" in filename:

                        mypath ="/".join( (root,filename) )

                        for dic in  conllfile2tree(mypath): # pour chaque
arbre
                              l=causeaeffet(dic)
                              for cause,effet,phrase,cas in l:# resu est
un quadruplet
```

```
                                    (effethead,effetlist) = effet

list=[ dic[i].get("word","").lower() for i in effetlist]
                                    if effetlist and
dic[effetlist[0]].get("word","") == ",":
                                          effetlist=effetlist[1:]
                                    effetlist= [ i for i in effetlist]


                                    (causehead, causelist) = cause
                                    if causelist and
dic[causelist[0]].get("word","") == ",":
                                          causelist=causelist[1:]
                                    causelist= [ i for i in causelist if
dic[i].get("word","") != "." ]

                                    #dans le cas "pour correct" on doit
mettre les verbes de la cause à l'infinitif
                                    if cas=="'pour' correct"  or
cas=="'lorsque pouvez' faux" or cas=="'lorsque pouvez' correct":

      cause=[ dic[i].get("word","").lower() for i in causelist]
                                          causeinf= leffe(cause)
                                          causeinfstr="".join(causeinf)


fet=[ dic[i].get("word","").lower() for i in effetlist]
                                          effetinf= leffe(effet)
                                          effetliststr="".join(effetinf)


                                    elif cas=="'participepresent' cor-
rect":

      cause=[ dic[i].get("word","").lower() for i in causelist]
                                          causeinf= leffe(cause)
                                          causeinfstr="".join(causeinf)


fet=[ dic[i].get("word","").lower() for i in effetlist]
                                          effetinf= leffe(effet)
                                          effetliststr="".join(effetinf)

                                    else:
                                          causeinfstr="
".join([ dic[i].get("word","").lower() for i in causelist])
                                          effetliststr="
".join( [ dic[i].get("word","").lower() for i in effetlist] )


                                    count+=1

                                    sortirCauseEffet(mypath, [cau-
seinfstr , unicode(effetliststr)], phrase, cas,Pour_resultList)



      print count,"résultats"
```

```
Pour_resultList.close()
```

VI. Annexe 6 – Programme permettant de générer les questions-réponses

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys,cgi,cgitb,json, codecs, re
import glob, os, random

def f6(seq):
      Set = set(seq)
      return list(Set)

fichierep=codecs.open('www/quiz/config.php','w','utf-8')
resfile=codecs.open('causations_test/Pour_resultList.txt','r','utf-8')

count=0
ld1 = []
ld2 = []
ld3 = []
ld4 = []
ldglob = []
listligne=[]
lglobal=[]

# si ligne est égale à "___________", on ne la met pas dans la
liste,sinon on met dans la liste
# si ligne est égale à "___________", on lregarde si la liste est bien
égale à 6 et si oui la ligne 2 et 4 forment le dico
for line in resfile:
      line=line.strip()
      if line == "_________":
            if len(listligne)==5:
                  lglobal+=[listligne]
            else:
                  pass
            listligne=[]
      else:
            listligne+=[line]
count=0
tst=codecs.open('qcm_tst.txt','w','utf-8')
for l in lglobal:
      #tst.write(str(l))
      #count+=1
      #tst.write("\n")
      #tst.write(str(count))

      if l[1]=="cas: 'pour' correct":

      ld1+=[(l[1],l[3].split(":::")[1].replace('\'','\\\''),l[4].split("
:::")[1].replace('\'','\\\''),u"'Que faire pour ",int(2),int(1))]
            ldglob+=ld1
            #print ld1
```

```
      elif l[1]=="cas: 'participepresent pouvez' faux" or l[1]=="cas:
'participepresent' correct":

      ld2+=[(l[1],l[3].split(":::")[1].replace('\'','\\\''),l[4].split("
:::")[1].replace('\'','\\\''),u"'Comment faire pour ",int(2),int(1))]
            ldglob+=ld2
            #print ld2
      elif l[1]=="cas: 'pour pouvez alternative'" or l[1]=="cas: 'pour
pouvez' faux" or l[1]=="cas: 'pour pouvez' correct":

      ld3+=[(l[1],l[3].split(":::")[1].replace('\'','\\\''),l[4].split("
:::")[1].replace('\'','\\\''),u"'Que faire pour ",int(2),int(1))]
            ldglob+=ld3
      elif l[1]=="cas: 'lorsque' correct":

      ld4+=[(l[1],l[3].split(":::")[1].replace('\'','\\\''),l[4].split("
:::")[1].replace('\'','\\\''),u"'Que se passe-t-il lorsque vous
",int(1),int(2))]
            ldglob+=ld4
      else:
            pass


#print ldglob

def qcmtophp(fichier):
      x=0
      fichier.write(u"<?php"+"\n")
      fichier.write(u"$questions = array("+"\n\n")
      choisisglob = [ld1,ld2,ld3,ld4]
      #print choisisglob

      while x <10:

            choisisall = random.sample(choisisglob,1)
            #print choisisall
            #break

            #on veut choisir au hasard quatre phrases dans choisisall
mais on ne veut pas de doublon
            choisis = random.sample(choisisall[0],3)

            #print choisis
            #break

            fi-
chier.write(choisis[0][3]+choisis[0][choisis[0][4]]+"?'"+u"=>array("+"\
n")

            fichier.write(u"REPONSES=>array("+"\n")

            ##génération des choix de reponse
            choix = []

            ##position de la reponse
            r=choisis[0][5]
            bonnereponse=choisis[0][r]
            ##print bonnereponse
```

```
            for c in choisis:
                  choix+=[ c[r] ]

            random.shuffle(choix)

            for i,c in enumerate(choix):
                  fichier.write("'"+c+"',"+"\n")
                  #print c
            fichier.write("),"+"\n")
            fi-
chier.write(u"BONNE_REPONSE=>"+str(choix.index(bonnereponse))+"\n")
            fichier.write("),"+"\n\n")
            x=x+1

      fichier.write(")\n")
      fichier.write("?>")



qcmtophp(fichierep)
```

VII. Annexe 7 – Extrait de l'application web mise en place

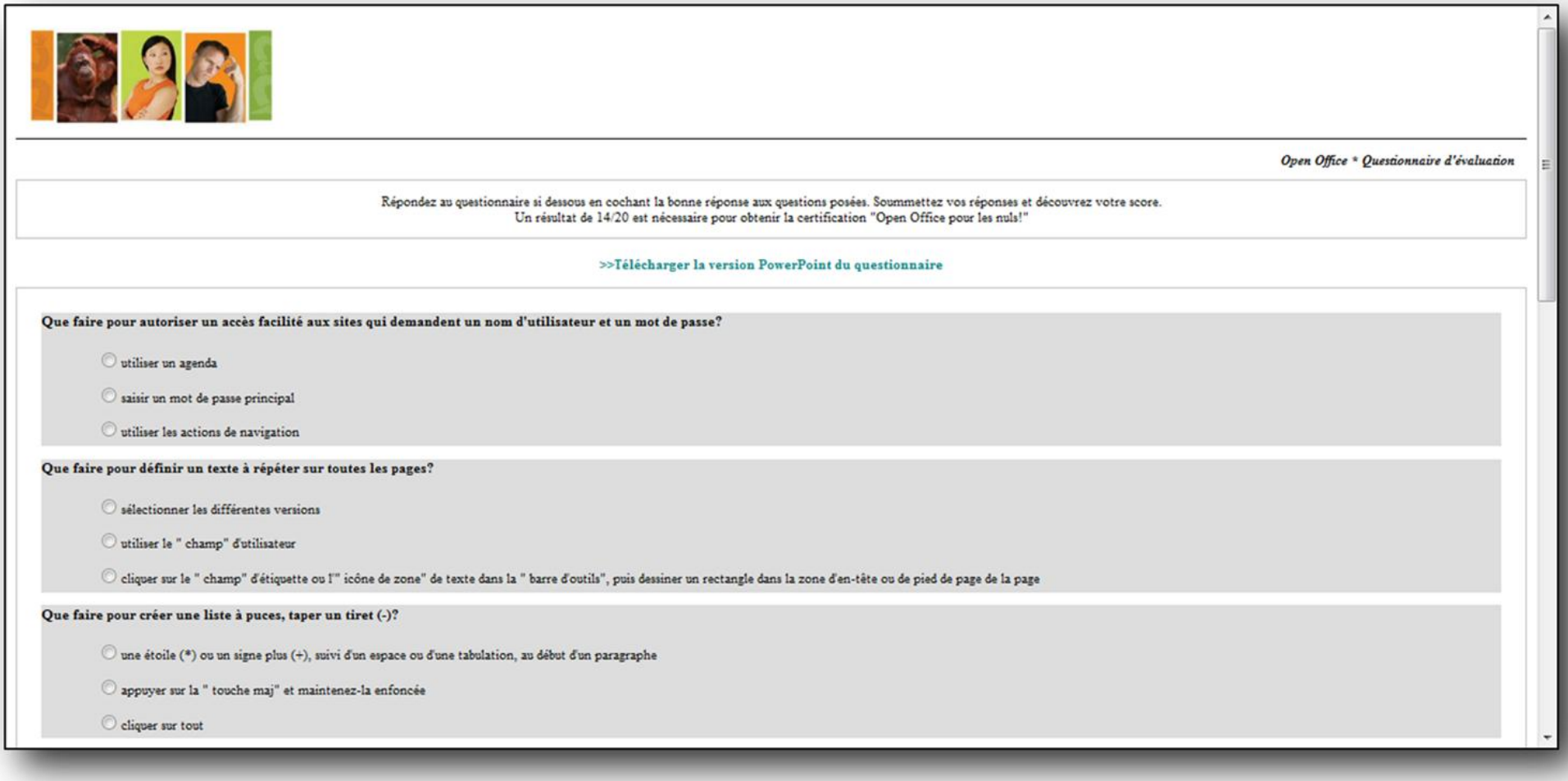


Figure 1 - L'apprenant peut tester ses connaissances et soumettre ses réponses

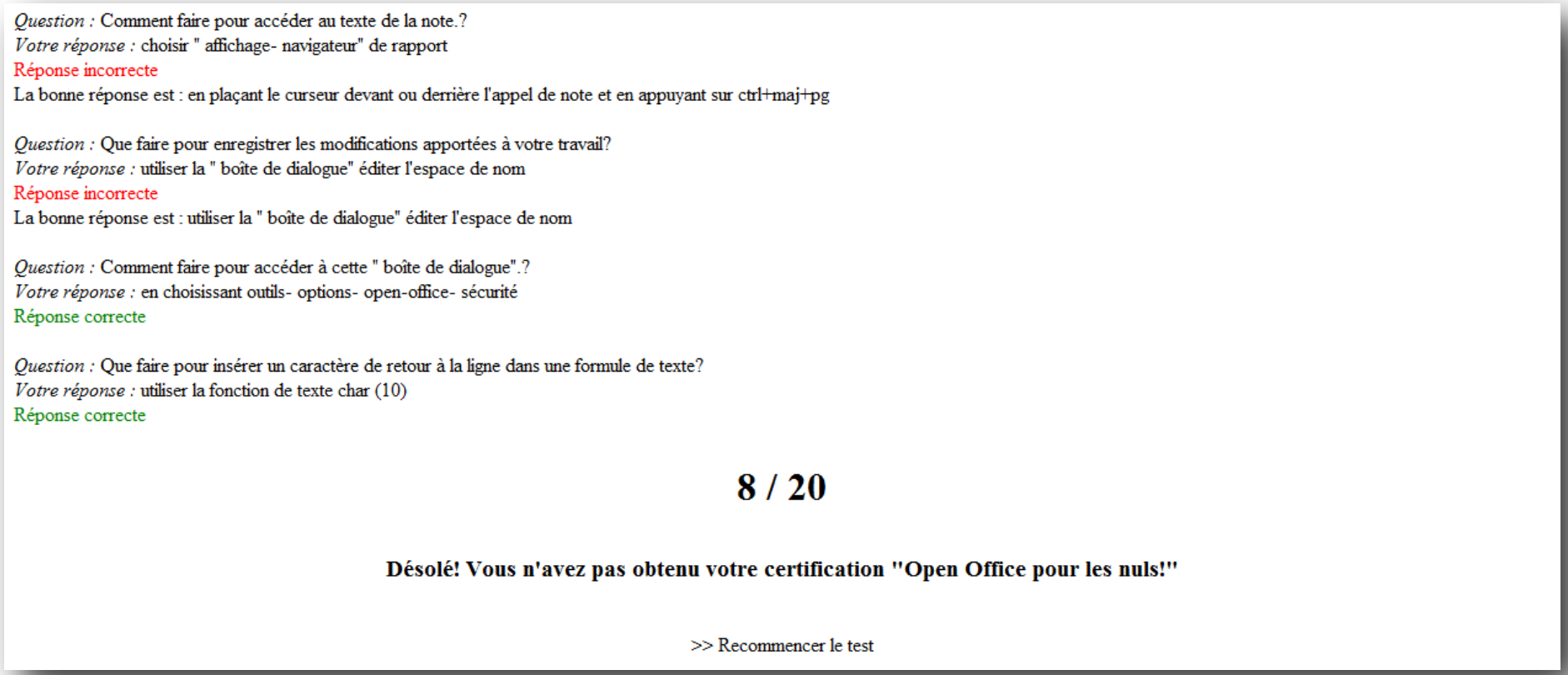


Figure 2- L'apprenant soumet son questionnaire et obtient ou non la certification

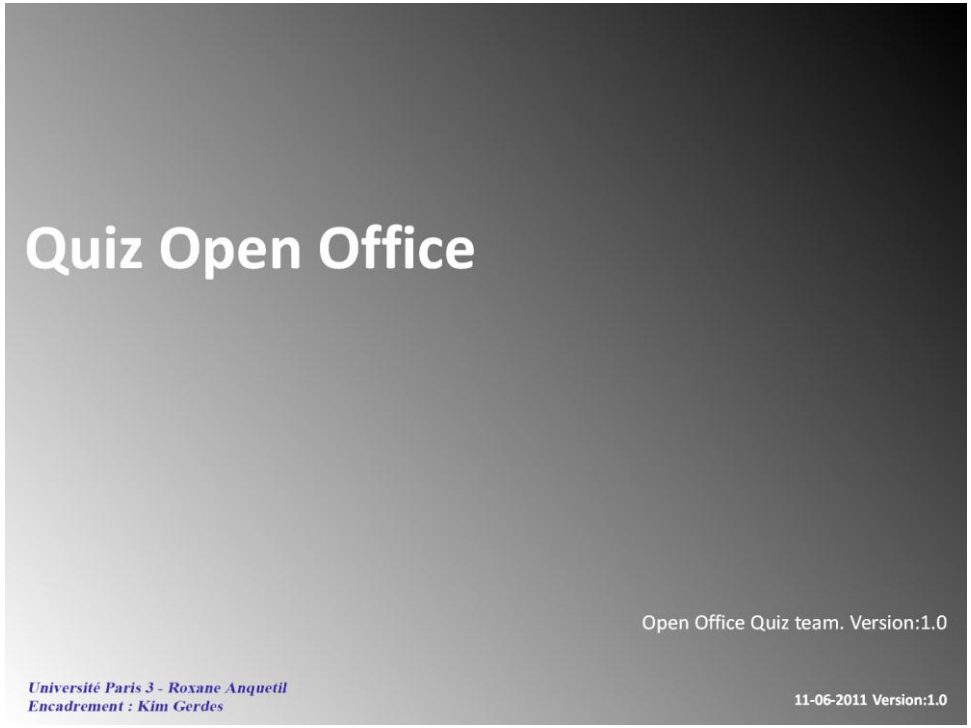


Figure 3 - Un professeur peut également générer un support Powerpoint pour évaluer ses étudiants lors d'une présentation

Bibliographie

Abeillé, A. (2007). *Les grammaires d'unification.* Paris: Lavoisier.

Corblin, F. (1995). *Les formes de reprise dans le discours.*

François-Régis Chaumartin, S. K. (s.d.). Une approche paresseuse de l'analyse sémantique ou comment construire une interface syntaxe-sémantique à partir d'exemples.

Gerdes, K. e. (2011). *Projet Rhapsody*. Récupéré sur Rhapsody Ilpga: http://rhapsodie.ilpga.fr/wiki/D%C3%A9pendances

Guillaume Bonfante, B. G. (2010, 19–23 juillet). Réécriture de graphes de dépendances pour l'interface. *TALN 2010,Montréal*.

Heinecke, J. (2006). *Génération automatique des représentations ontologiques.*

Igor, K. S. (2006). *Les sémantèmes de causation en français.* S. Hamon & M. Amy.

Jario, L. D. (2007). *Mise à jour d'ontologies basée sur les motifs fréquents.* LIRMM.

Lareau, S. K. (2005). *Grammaire d'Unification Sens-Texte : modularité et polarisation.* Dourdan.

Lison, P. (2006). *Implémentation d'une Interface Sémantique-Syntaxe.* Louvain, Université Catholique.

Marie-Hélène Candito, S. K. (s.d.). Can the TAG derivation tree represent a semantic graph ? An answer in the light of Meaning-Text Theory.

Mel'čuk, I. (1997). Vers une linguistique Sens-Texte. Leçon inaugurale. *Paris: Collège de France.*

Mounin, G. (1972). *La Sémantique* . Petite Bibliothèque Payot.

Nivre, J. (2007). *http://nextens.uvt.nl/depparse-wiki/DataFormat*. Récupéré sur Depparse Wiki: http://nextens.uvt.nl/depparse-wiki/DataFormat

Pantel, D. L. (s.d.). *Discovery of Inference Rules for Question Answering.* Department of Computing Science - University of Alberta.

Polguère, A. (1998). *Lexicologie et Sémantique lexicale* . Montréal: Les presses de l'université de Montréal.

Polguère, A. (s.d.). *Les fonctions lexicales pour tous.* Grenoble : OLST.

Ruslan Mitkov, L. A. (s.d.). *Computer-Aided Generation of Multiple-Choice Tests.* School of Humanities, Languages and Social Sciences - University of Wolverhampton.

Schwarze, C. (2001). *Introduction à la sémantiqué lexicale.* Tübingen.

Steven Bird, E. K. (s.d.). Natural Language Processing with Python . O'Reilly.

Tesnière, L. (1959). *Eléments De Syntaxe Structurale.* Editions Klincksieck.

Thompson, W. C. (Une 1987). *Rhetorical Structure Theory : A theory of text organization.* California: http://www.scribd.com/doc/7765676/MannThompson1987-Rhetorical-Structure-Theory.

Vasin Punyakanok, D. R.-t. (s.d.). *Natural Language Inference via Dependency Tree Mapping : An Application to Question Answering.* University of Illinois at Urbana-Champaign.