Université catholique de Louvain
Faculté des Sciences Appliquées
Département d'Ingénierie Informatique

Implémentation d'une Interface Sémantique-Syntaxe basée sur des Grammaires d'Unification Polarisées

Pierre Lison

Mémoire présenté en vue de l'obtention du grade d'ingénieur civil en informatique.

Promoteur: Prof. Pierre Dupont
Co-promoteur: Prof. Cédrick Fairon

Louvain-la-neuve
Année Académique 2005-2006

« *Une nouvelle période commence, où des chercheurs d'abord séparés par leur formation de départ - mathématique, psychologie, philologie - se rapprochent et contribuent à l'édification d'une linguistique unique. C'est une période où l'on réfléchit sur les difficultés rencontrées par les premières expériences sur ordinateurs, et où l'on révise et approfondit les notions de base. Il n'en apparaît que mieux à quel point Tesnière était allé droit à l'essentiel.* »

Jean Fourquet, préface à la deuxième édition des "Eléments de Syntaxe Structurale" de Lucien Tesnière, 1976.

« *Tous les moyens de l'esprit sont enfermés dans le langage ; et qui n'a point réfléchi sur le langage n'a point réfléchi du tout.* »

Alain, Propos sur l'éducation, 1932.

Abstract

This work relates to *Natural Language Processing* [NLP], a scientific research field situated at the intersection of several classical disciplines such as computer science, linguistics, mathematics, psychology, and whose object is the design of computational systems able to *process* (i.e. understand and/or generate) linguistic data, whether oral or written.

In order to achieve that goal, it is often necessary to design *formal models* able to simulate the behaviour of complex linguistic phenomena. Several theories have been elaborated to this end. Significent divergences do exist between them concerning linguistic foundations as well as grammatical formalisms and related computer tools. Nevertheless, many efforts have recently been made to bring them closer together, and two major trends clearly seem to emerge from the main contemporary theories :

- They are all built around *modular* architecture, explicitly distinguishing the semantic, syntactic, morphological and phonological representation levels ;
- They all give a central position to the *lexicon*, rightly seen as a crucial resource for the establishment of efficient and wide-coverage systems.

This study examines an essential component of all these models : the *semantics-syntax interface*, responsible for the mapping between the semantic and syntactic levels of the architecture. Indeed, many distortion phenomenas can be found in every human language between these two levels. Let us mention as examples the handling of idioms and locutions, the active/passive alternation, the so-called "extraction" phenomenas (relative subordinates, interrogative clauses), elliptic coordination, and many others.

We approach this issue in the framework of a particular linguistic theory, the *Meaning-Text Unification Grammars* [MPUG] (Kahane, 2002; Kahane et Lareau, 2005), an articulated mathematical model of language recently devised by S. Kahane, and his related description formalism, *Polarized Unification Grammars* [PUG] (Kahane, 2004).

The first part of our work deals with the general study of the role and inner workings of the semantics-syntax interface within this theory. We then propose a concrete implementation of it based on *Constraint Programming*. This implementation is grounded on an axiomatization of our initial formalism into a *Constraint Satisfaction Problem*.

Rather than developing the software entirely from scratch, we have instead chosen to reuse an existing tool, the *XDG Development Kit*, and to adapt it to our needs. It is a grammar development environment for the meta grammatical formalism of *Extensible Dependency Grammar* [XDG] (Debusmann, 2006), entirely based on Constraint Programming.

Practically, this work makes three original contributions to NLP research :

1. An *axiomatization* of MTUG/PUG into a Constraint Satisfaction Problem, enabling us to give a solid formal ground to our implementation ;
2. An *implementation* of our semantics-syntax interface by means of a compiler from MTUG/-PUG grammars to XDG grammars called `auGUSTe` as well as by the integration of eight new "principles" (i.e. constraints sets) into XDG ;
3. And finally, the *application* of our compiler to a small hand-crafted grammar centered on culinary vocabulary in order to experimentally validate our work.

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* (i.e. comprendre et/ou synthétiser) des données linguistiques écrites ou orales.

Pour ce faire, il est souvent nécessaire de conçevoir des *modélisations formelles* permettant de simuler le comportement de certains phénomènes linguistiques complexes. De nombreuses théories ont été élaborées à cet effet. Des divergences considérables apparaissent parfois entre elles, tant au au niveau des fondements linguistiques que des formalismes et des outils informatiques développés. Néammoins, la tendance actuelle est plutôt au rapprochement, et deux orientations semblent clairement se dégager au sein des principales théories contemporaines :

- Elles sont tous construites autour d'une architecture *modulaire* distinguant explicitement les niveaux de représentation sémantique, syntaxique, morphologique et phonologique ;
- Elles accordent une place essentielle au *lexique*, considéré à juste titre comme une ressource cruciale pour le développement d'outils performants et à large couverture.

Ce mémoire s'intéresse à une composante fondamentale de tous ces modèles : l'*interface sémantique-syntaxe*, chargée d'assurer la correspondance entre les niveaux sémantique et syntaxique de l'architecture. Il existe en effet dans chaque langue d'importants phénomènes de distorsion entre les deux niveaux. A titre d'exemple, mentionnons le traitement des expressions figées et des collocations, l'alternance actif/passif, les phénomènes dits "d'extraction" (relatives, interrogatives indirectes), la coordination elliptique, et bien d'autres.

Nous abordons cette question dans le cadre d'une théorie linguistique particulière, les *Grammaires d'Unification Sens-Texte* [GUST] (Kahane, 2002; Kahane et Lareau, 2005), un modèle mathématique articulé de la langue récemment développé par S. Kahane , et de son formalisme de description associé, les *Grammaires d'Unification Polarisées* [GUP] (Kahane, 2004).

La première partie de notre travail porte sur l'étude théorique du rôle et du fonctionnement de l'interface sémantique-syntaxe au sein de cette théorie. Nous proposons ensuite une implémentation concrète basée sur la *programmation par contraintes.* Cette implémentation est fondée sur une axiomatisation de notre formalisme initial en un *problème de satisfaction de contraintes.*

Plutôt que de conçevoir de bout en bout l'entièreté de notre programme, nous avons choisi de réutiliser un outil déjà existant, *XDG Development Kit*, et de l'adapter à nos besoins. Il s'agit d'une plateforme de développement de grammaires issues du formalisme meta-grammatical *Extensible Dependency Grammar* [XDG] (Debusmann, 2006), basé sur la programmation par contraintes.

En pratique, ce mémoire présente trois contributions originales à la recherche en TALN :

1. Une *axiomatisation* théorique de GUST/GUP en un problème de satisfaction de contraintes, nous permettant ainsi de donner un assise formelle solide à notre travail ;
2. Une *implémentation* de notre interface sémantique-syntaxe par le biais d'un compilateur de grammaires GUST/GUP en grammaires XDG, baptisé `auGUSTe`, ainsi que d'un ensemble de huit "principes" (i.e. ensembles de contraintes) supplémentaires intégrés à XDG ;
3. Et enfin, l'*application* de notre compilateur à une mini-grammaire construite par nos soins et axée sur le vocabulaire culinaire, afin de valider expérimentalement notre travail.

Remerciements

Le présent travail aurait difficilement pu aboutir sans le concours de nombreuses personnes que je tiens à remercier.

Je pense tout d'abord à mes deux promoteurs, Pierre Dupont et Cédrick Fairon, pour leur soutien et leurs conseils avisés tout au long de l'élaboration de ce mémoire. Ils ont parfaitement su canaliser mon enthousiasme pour le sujet en m'aidant à bien le circonscrire et distinguer l'essentiel de l'accessoire. Leur insistance à souligner l'importance des questions empiriques me fut également très profitable.

J'adresse mes plus vifs remerciements à Sylvain Kahane (Prof. à l'Université de Paris 10) pour son aide précieuse et ses éclairages toujours pertinents. Depuis notre entrevue à Paris au printemps dernier qui a permis de mettre le sujet sur les rails, il a constamment soutenu ce travail et s'est toujours montré disponible pour mes questions, parfois bien naïves.

Je remerçie également François Lareau (doctorant à Universitat Pompeu Fabra, Barcelone) pour ses remarques judicieuses concernant mon travail. J'espère que mon implémentation pourra lui être d'une certaine utilité pour l'avancement de sa thèse, qui promet d'être bien intéressante.

Je tiens à exprimer ma reconnaissance à Denys Duchier (Prof. à l'Université d'Orléans) et à Ralph Debusmann (chercheur à l'Universität des Saarlandes, Saarbrücken) pour leur aide particulièrement utile à propos de XDG.

Merci également à Piet Mertens (Prof. à la K.U.Leuven) pour notre entrevue en novembre dernier, qui m'a éclairé sur certains aspects obscurs des grammaires de dépendance.

Enfin, *last but not least*, je tiens à exprimer ma profonde gratitude à mes proches, parents et amis, qui m'ont constamment et chaleureusement soutenu durant ces années d'études, et qui m'ont permis de faire de ce passage à Louvain-la-Neuve une formidable expérience, intellectuelle bien sûr, mais aussi et surtout humaine.

Table des matières

1 Introduction 9

1.1 Le Traitement Automatique du Language Naturel (TALN) 10

1.1.1 Domaines 10

1.1.2 Approches 12

1.2 Méthodologie 13

1.3 De la modélisation en linguistique 14

1.3.1 Motivation 14

1.3.2 Application 15

1.4 Structure du mémoire 16

2 Grammaires de dépendance et théorie Sens-Texte 17

2.1 Grammaires de Dépendance 17

2.1.1 Généralités 17

2.1.2 Définitions formelles 17

2.1.3 Notion de dépendance 19

2.1.4 Syntagme *vs.* dépendance 19

2.1.5 Ordre des mots 20

2.1.6 Projectivité 21

2.1.7 Théorie de la translation de Tesnière 23

2.1.8 Fonctions syntaxiques 24

2.2 Théorie Sens-Texte 25

2.2.1 Postulats 25

2.2.2 Architecture générale 26

2.2.3 Représentations formelles 27

2.2.4 Fonctions Lexicales 29

2.2.5 Modèles Sens-Textes 33

3 Grammaire d'Unification Sens-Texte 35

3.1 Un modèle mathématique articulé de la langue 35

3.2 Théorie des signes 36

3.2.1 Signe saussurien et signe GUST 36

3.2.2 Exemple du passif 37

3.2.3 Interaction des signes 38

3.3 Représentations linguistiques de GUST 38

3.3.1 Représentation sémantique 38

3.3.2 Représentation syntaxique 39

3.3.3 Représentation morphotopologique 39

3.4 Grammaires transductives, génératives et équatives 41

3.5 Interfaces de GUST 42
3.5.1 Hiérarchisation et lexicalisation 42
3.5.2 Interface sémantique-syntaxe $\mathcal{I}_{sem-synt}$ 42
3.5.3 Interface syntaxe-morphotopologie $\mathcal{I}_{synt-morph}$ 44
3.5.4 Interface morphotopologie-phonologie $\mathcal{I}_{morph-phon}$ 44
3.5.5 Stratégies d'interaction 44
3.6 Grammaire d'Unification Polarisées 45
3.6.1 Généralités 45
3.6.2 Système de polarités 45
3.6.3 Grammaires de bonne formation *vs.* grammaires de correspondance 46
3.6.4 Procédures équative, transductive et générative 47
3.6.5 Double polarité 47
3.7 Exemple détaillé d'utilisation des GUP 48
3.7.1 Grammaire sémantique de bonne formation $\mathcal{G}_{sem}$ 48
3.7.2 Grammaire syntaxique de bonne formation $\mathcal{G}_{synt}$ 48
3.7.3 Grammaire de correspondance $\mathcal{I}_{sem-synt}$ 50
3.7.4 Exemple d'opération de synthèse sémantique 50
3.8 Possibilité de "va-et-vient" entre niveaux 55
3.9 Arbres à bulles 55
3.9.1 Motivation 55
3.9.2 Définition formelle 57
3.9.3 Grammaires à bulles 57

4 Interfaces Sémantique-Syntaxe **58**
4.1 Représentation logique et sous-spécification 58
4.1.1 Structure des représentations logiques 58
4.1.2 Grammaires $\mathcal{G}_{pred}$, $\mathcal{G}_{log}$ et $\mathcal{I}_{sem-synt}$ 59
4.1.3 Sous-spécification 61
4.1.4 Implémentation 61
4.2 Phénomènes linguistiques abordés 62
4.2.1 Sous-catégorisation 62
4.2.2 Articles définis, indéfinis et partitifs 62
4.2.3 Anaphore 62
4.2.4 Modifieurs 63
4.2.5 Copule 63
4.2.6 Conjugaison 64
4.2.7 Synonymes 64
4.2.8 Expressions figées 65
4.2.9 Collocations 65

4.2.10 Verbes supports 66
4.2.11 Verbes de montée 66
4.2.12 Verbes de contrôle 66
4.2.13 Passivation 67
4.2.14 Compléments circonstanciels 67

5 Axiomatisation de GUST/GUP 68
5.1 Programmation par contraintes 68
5.1.1 Généralités 68
5.1.2 Définitions 69
5.1.3 Types de contraintes 69
5.2 *Extensible Dependency Grammar* 70
5.2.1 Formalisation 70
5.2.2 Grammaires XDG 72
5.2.3 *XDG Grammar Development Kit* 72
5.2.4 Evolution future du formalisme 74
5.2.5 Adéquation à GUST/GUP 74
5.3 Axiomatisation de GUST/GUP en système de contraintes 75
5.3.1 Cas basiques 76
5.3.2 Règles sagittales 78
5.3.3 Règles d'accord 79
5.3.4 Règles d'interface 80
5.3.5 Classes et unités lexicales 86
5.3.6 Résumé 86

6 Implémentation de l'Interface Sémantique-Syntaxe 87
6.1 Méthodologie 87
6.2 `auGUSTe` : Compilateur XDG ⇒ GUST 88
6.2.1 Cahier des charges 88
6.2.2 Entrées et sorties 89
6.2.3 Architecture 91
6.2.4 Etapes principales 92
6.2.5 Complexité 94
6.3 Contraintes XDG 94
6.3.1 Généralités 94
6.3.2 Complexité et Performances 95
6.3.3 Algorithmes 97
6.4 Regards croisés sur GUST/GUP et XDG 98

7 Validation expérimentale 100

7.1 Méthodologie . . . 100
7.2 Grammaire “culinaire” . . . 101
7.2.1 Éléments grammaticaux . . . 101
7.2.2 Éléments lexicaux . . . 102
7.3 Corpus de validation . . . 104
7.4 Résultats . . . 105
7.4.1 Généralités . . . 105
7.4.2 Aspects quantitatifs . . . 107

8 Conclusions et Perspectives 109
8.1 Résumé . . . 109
8.2 Perspectives . . . 109
8.3 En guise de conclusion . . . 111

A Installation et mode d’emploi d’`auGUSTe` 112
A.1 Installation . . . 112
A.1.1 Installez Python : . . . 112
A.1.2 Installez Mozart/oz : . . . 112
A.1.3 Installez auGUSTe . . . 112
A.1.4 Installation de Dia et sa feuille de style . . . 113
A.2 Utilisation d’`auGUSTe` . . . 113
A.2.1 Menu principal . . . 113
A.3 Utilisation de XDK version auGUSTe . . . 114
A.4 Utilisation de Dia pour la conception de structures GUP . . . 117
A.4.1 Remarques diverses . . . 118
A.5 Exemple de grammaire . . . 118

B Résultats de la génération de 20 graphes sémantiques 120

C Extraits du code source 140
C.1 PUGParser.py . . . 140
C.2 Compiler.py . . . 143
C.3 CompileTests.py . . . 149
C.4 GUSTSemEdgeConstraints.oz . . . 159
C.5 GUSTSagittalConstraints.oz . . . 160
C.6 GUSTLinkingConstraints.oz . . . 162

D Glossaire 179

Bibliographie 183

Chapitre 1

Introduction

Le *langage humain* est un objet tout à la fois familier et mystérieux. Familier, de par l'expérience quotidienne et intime que nous en faisons : "l'homme ne peut se concevoir autrement que comme sujet parlant". Mystérieux, car la nature profonde du langage continue de rester une énigme pour la science. Certes, la recherche linguistique à l'oeuvre depuis plus d'un siècle a permis de lever un pan sur certains de ses aspects, mais la richesse de ses structures semble inépuisable, et bien des interrogations fondamentales demeurent.

Parmi celles-ci, l'une des plus importantes porte sur la possibilité de *simuler* informatiquement le fonctionnement du langage (même de façon partielle et limitée) à des fins d'automatisation. Il est pour cela nécessaire de concevoir des *modèles formels* de la langue, ainsi des *algorithmes* permettant de les manipuler et des *ressources linguistiques* (lexique, grammaire, corpus) leur donnant vie et servant de matériau de travail.

L'élaboration de tels outils a donné lieu à une nouvelle discipline scientifique, dénommée "**Traitement Automatique de la Langue Naturelle**" [dorénavant TALN]. Elle poursuit un double objectif :

- *théorique* : les modélisations formelles sont devenues un outil scientifique incontournable, et aucune science ne peut se permettre de rejeter leur utilisation ;
- *pratique* : les applications potentielles du TALN sont innombrables[1] et auront, à n'en point douter, un impact considérable sur nos vies quotidiennes.

De fait, c'est toute la question des interfaces homme-machine qui est ici en jeu, puisque l'obstacle principal à l'interaction entre l'homme et la machine est un problème de *communication* : les logiciels actuels ne comprennent pas ou peu le langage naturel, et l'utilisateur n'a de choix qu'entre des interfaces utilisateurs (textuelles ou graphiques) à l'expressivité limitée, ou l'utilisation de langages formels difficiles à maîtriser pour le non-technicien, et qui ne correspondent en aucune manière à la structuration de la pensée humaine.

Même si l'étendue du langage que la machine peut comprendre et synthésiser (son "domaine de discours") est très limitée, l'utilisation d'outils de TALN peut améliorer très significativement la convivialité et la productivité de nos systèmes informatiques. Citons, sans prétention à l'exhaustivité, quelques exemples d'applications : interrogation de base de données en langue naturelle, concordanciers, génération de textes, systèmes d'extraction et de recherche d'informations dans de larges banques de données textuelles (dont le Web), reconnaissance et synthèse vocale, dictaphones, agents conversationnels abstraits ou "incarnés" (dans un robot), etc.

De plus, le TALN peut également servir à faciliter la communication *entre* êtres humains. On pense bien sûr à la traduction automatique, véritable "Graal" du TALN puisqu'il s'agit d'une tâche fascinante mais terriblement complexe à automatiser, mais aussi, plus modestement, aux

[1]On parle d'ailleurs des *industries de la langue* pour dénoter le secteur économique qui se consacre à l'élaboration et à l'utilisation d'applications liées, de près ou de loin, aux langues naturelles.

logiciels d'assistance à la traduction, aux modules de correction orthographique et grammaticale, d'aide à la rédaction, aux programmes d'apprentissage des langues, etc.

Le décor étant ainsi planté, nous poursuivons à présent ce chapitre introductif, qui se décline en quatre sections :
- La section 1.1 nous permet de donner quelques précisions supplémentaires à propos du *cadre général* dans lequel s'inscrit ce mémoire, à savoir le TALN ;
- Ensuite, nous détaillons dans la section 1.2 la *méthodologie* que nous avons suivie pour mener à bien ce travail ;
- Nous poursuivons brièvement dans la section 1.3 par une discussion du concept de *modélisation* en linguistique, cette idée ayant un rôle central dans le travail effectué ;
- Et enfin nous résumons dans la section 1.4 en quelques paragraphes la *structure* globale du texte de ce mémoire (qui, comme nous le verrons, est directement liée à notre méthodologie).

1.1 Le Traitement Automatique du Language Naturel (TALN)

Définition. Le *Traitement Automatique du Langage Naturel*[2] est une discipline scientifique à mi-chemin entre la linguistique et l'informatique, dont l'objet porte sur les aspects *calculables* de la faculté langagière. Elle fait partie du vaste champ des sciences cognitives et s'imbrique plus particulièrement dans celui de l'Intelligence Artificielle (AI), une branche de l'informatique visant à élaborer des modèles calculables de la cognition et de l'action humaine.[3]

1.1.1 Domaines

Il est possible d'opérer une classification des différents domaines du TALN selon deux critères : (1) le *niveau* d'analyse linguistique et (2) sa *direction*.

Traditionnellement, la linguistique est divisée en différentes branches, que le TALN reprend en grande partie (même si la perspective sur ceux-ci y est parfois bien différente) :

- **Phonétique** et **Phonologie** : la phonétique étudie la production, la transmission et la réception des *sons vocaux*. On parle respectivement de phonétique articulatoire, acoustique et auditive. La phonologie est quant à elle l'étude des *phonèmes*, qui sont les unités minimales distinctives d'une langue (i.e. qui permettent de distinguer un mot d'un autre) Les deux notions sont bien différentes, comme l'a brillamment montré (de Saussure, 1916) : un mot particulier peut en effet être prononcé de diverses manières (dues à l'accent, l'intonation, la prosodie), mais il contiendra normalement la même chaîne de phonèmes.

- La **Morphologie** étudie, comme son étymologie l'indique, la *forme* ou la structure interne des mots, c'est-à-dire la façon dont ceux-ci sont construits à partir de morphèmes, qui sont les unités minimales significatives d'une langue. "`Indémontrables`" est ainsi un mot constitué de quatre morphèmes : "`in`", "`démontr`", "`able`" et "`s`".

- La **Syntaxe** étudie les règles par lesquelles les mots peuvent se *combiner* pour former des unités plus grandes ; en général des phrases. Suivant (Blanche-Benveniste, 1991), l'on peut encore distinguer une *micro-syntaxe*, constituée par les différents dispositifs de rection[4], et

[2] angl. *Natural Language Processing*. De nombreuses appelations alternatives, plus ou moins synonymes, existent également : linguistique computationnelle (angl. *computational linguistics*), linguistique informatique, informatique linguistique, linguistique calculatoire, technologies du langage, etc.

[3] Cette définition a été adaptée à partir de (Uzkoreit, 2004)

[4] La rection désigne la propriété qu'ont certains unités linguistiques d'être accompagnées d'un complément (obligatoire ou optionnel). Par exemple, un verbe transitif devra être accompagné d'un sujet et d'un complément d'objet direct, et optionnellement de compléments indirects, d'adverbes, etc.

une *macro-syntaxe*, "syntaxe de zones" traitant des éléments disloqués sans lien de dépendance avec le noyau central : apposition, détachements, énumérations, etc.

– La **Sémantique** est l'étude du *sens* des unités linguistiques et de leur organisation au sein des messages que l'on peut exprimer dans une langue.

– Enfin, la **Pragmatique** concerne l'étude du langage *en contexte*, c'est-à-dire de la manière dont les phrases sont combinées pour former des *discours*, s'insérer dans une dynamique *conversationnelle*, ainsi que la façon dont le langage est utilisé pour *référer* à la réalité extérieure (ou plus exactement, la représentation que le locuteur en a dans ses "espaces mentaux") et éventuellement la *modifier* (théorie des actes de langage : offres, demandes, ordres, promesses, attitudes liées à un comportement social).

Notons que, comme souvent en linguistique, ces définitions ne sont ni stables ni unanimement acceptées : ainsi l'utilité de notions comme le "mot" ou la "phrase" ou la possibilité même de scinder l'analyse linguistique en différents modules sont régulièrement contestées, notamment par les tenants de la linguistique cognitive. Nous n'entrerons évidemment pas dans ce débat.

Dans le cadre de ce mémoire, nous nous intéresserons principalement à la syntaxe, à la sémantique et surtout à l'interface entre ces deux niveaux. Nous n'aborderons les questions phonétiques, phonologiques, morphologiques, topologiques (i.e. qui concerne les variations dans l'ordre linéaire des mots) ou pragmatiques que de manière périphérique, lorsqu'elle ont une pertinence par rapport à notre sujet. Il est néammoins intéressant de noter que nous avons conçu l'architecture de notre interface de manière modulaire, et qu'il est donc théoriquement tout à fait possible d'étendre notre travail en rajoutant des niveaux supplémentaires.

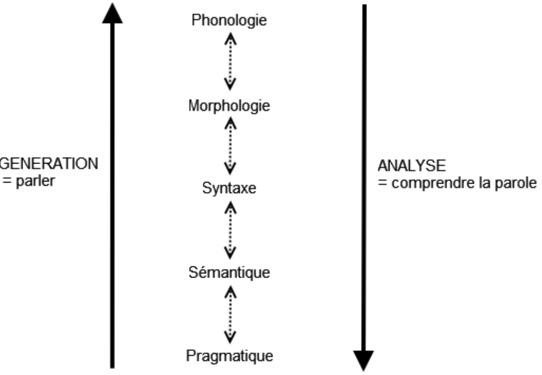


Fig. 1.1 – Niveaux d'analyse / génération utilisés en TALN

Les recherches menées en TALN diffèrent également selon que le traitement est réalisé de la parole au sens (on cherche à comprendre la signification d'un énoncé linguistique, oral ou écrit) ou du sens à la parole (l'objectif est alors de trouver une réalisation linguistique d'un sens donné). On parlera dans le premier cas d'*analyse*, et dans le second de *synthèse* ou *génération*.

Bien que le moteur d'inférence sous-jacent à notre interface soit théoriquement capable d'opérer dans les deux sens, la bidirectionnalité nous a posé de nombreux problèmes pratiques, l'opération d'analyse étant plus compliquée que la génération, de par la nécessité de désambiguïser les entrées. Or la levée d'ambiguité est un problème très complexe dépassant très largement le domaine de ce mémoire. (Mel'čuk, 1997) et (Polguère, 1998) viennent confirmer ce point de vue :

> « (..) Il convient de noter que, dans le présent article, la correspondance Sens Texte est toujours envisagée sous l'angle de la synthèse – du Sens au Texte – plutôt que sous celui de l'analyse – du Texte au Sens. La raison en est que seule la modélisa-

tion de la synthèse linguistique permet de mettre en jeu les connaissances purement linguistiques (contenues dans le dictionnaire et la grammaire de la langue). L'analyse, elle, ne peut se faire sans que l'on soit confronté au problème de la désambiguïsation, problème qui ne peut être résolu (par le locuteur ou par une modélisation formelle) sans le recours à des heuristiques basées sur des connaissances extra-linguistiques.

En résumé, la synthèse fait appel aux connaissances linguistiques du locuteur, qui doit effectuer des choix purement linguistiques entre les différentes options offertes par la langue pour l'expression d'un Sens donné. L'analyse passe par la résolution d'ambiguïtés, qui est un processus cognitif très complexe échappant au seul domaine de la linguistique. La modélisation du processus d'analyse est pour les linguistes Sens-Texte un cadre d'*application* de la linguistique ; celle du processus de synthèse est une méthode d'*expérimentation/simulation* permettant d'identifier clairement les phénomènes linguistiques. (...). » (Polguère, 1998, p. 4-5)

Dans le cadre de ce mémoire, nous avons donc choisi de nous concentrer sur la question de la génération, et notre interface n'est entièrement opérationnelle que dans cette direction-là. Nous laissons le problème de l'analyse à d'éventuels travaux ultérieurs.

1.1.2 Approches

Les recherches actuelles en TALN peuvent être, en première approximation, catégorisées en trois grandes tendances :

1. Les approches **symboliques** se basent sur des descriptions et des modélisations explicites de la langue naturelle, basées sur des ressources linguistiques élaborées "manuellement". Les grammaires d'unification, catégorielles, basées sur les contraintes, ... font ainsi partie de cette large catégorie. Elle partent d'hypothèses et de méthodologies très diverses, depuis l'utilisation de simples automates à l'implémentation de modèles sophistiqués, mais ont toutes en commun le recours à des ressources linguistiques (plus ou moins fines).

2. Les approches **statistiques** n'utilisent au contraire que peu ou pas d'informations linguistiques explicites, mais construisent leurs modèles par apprentissage automatique à partir de données contenues dans des *corpus*. Ceux-ci sont étiquetés (c'est-à-dire qu'on leur donne une catégorie grammaticale) et le système est ensuite "entraîné", de manière supervisée ou non, à analyser des textes. La désambiguation des unités lexicales s'opère en déterminant la plus probable des interprétations possibles. Ces algorithmes sont dits "robustes" car ils sont capables de fonctionner (avec plus ou moins de succès) sur n'importe quel texte.

3. Enfin, les approches **hybrides** qui ont émergé ces dernières années (Klavans et Resnik, 1996) cherchent à concilier le meilleur des deux approches en permettant d'intégrer des connaissances linguistiques à des modèles probabilistes du langage. [5]

Notre présent travail s'inscrit assez nettement dans le cadre de la première approche, mais il ne ferme pas pour autant la porte aux techniques probabilistes, comme nous l'expliquerons

[5] Tâche difficile, à la fois d'un point de vue formel - il est difficile d'encoder des formalismes linguistiques "sophistiqués" dans des modèles probabilistes souvent bien plus rudimentaires, et ce par souçi d'efficacité - et d'un point de vue pratique, les méthodes scientifiques en usage d'une part dans la communauté des ingénieurs, mathématiciens, informaticiens et d'autre part dans celle des linguistes, psychologues, philosophes et autres littéraires étant parfois à des années-lumière, ce qui rend la communication pour le moins malaisée !

Néammoins, cette approche, pour difficile qu'elle soit, me semble - c'est en tout cas ma conviction personnelle - extrémement prometteuse à long terme et enrichissante pour les deux "parties", et j'espère que ce mémoire apportera, modestement, une démonstration supplémentaire de la possibilité - et de la nécessité - d'un dialogue approfondi entre linguistes et ingénieurs pour la recherche en TALN.

brièvement dans les chapitres suivants. Théoriquement, il pourrait donc être étendu et réutilisé dans des systèmes hybrides de TALN.

1.2 Méthodologie

Le travail effectué dans le cadre de ce mémoire n'a évidemment pas été réalisé "à l'aveugle" ; nous nous sommes efforcés à suivre une méthodologie précise, planifiée et documentée[6], que l'on peut résumer succintement en cinq étapes :

1. Nous avons d'abord cherché à **comprendre** en profondeur le sujet à traiter, en rassemblant une *documentation* importante consacrée aux théories linguistiques, aux formalismes grammaticaux, aux phénomènes linguistiques situés à la frontière entre la sémantique et la syntaxe, etc. Nous nous sommes bien sûr particulièrement penchés sur les travaux décrivant le formalisme sur lequel notre travail se fondait : les *Grammaires d'Unification Sens-Texte / Grammaires d'Unification Polarisées* [GUST/GUP][7]

2. Nous sommes ensuite passés à la phase d'**analyse** proprement dite de notre problématique. Il nous a notamment fallu déterminer la nature exacte de l'algorithme à utiliser pour notre implémentation. Après avoir étudié chaque approche possible, nous avons fixé notre choix sur une approche basée sur la *programmation par contraintes*.

   De plus, nous avions eu connaissance d'un nouveau formalisme grammatical, *Extensible Dependency Grammar* [XDG], basé sur les contraintes et dont l'approche théorique était assez semblable à la nôtre. XDG possède une plateforme de développement de très bonne facture, *XDG Development Kit* [XDK], et nous avons décidé de la réutiliser dans le cadre de notre travail et de l'adapter à nos besoins.

3. Cette réutilisation du logiciel XDK pour notre interface sémantique-syntaxe nécessitait bien sûr une sorte de "compilateur" permettant de traduire des représentations de type GUST/GUP en leur équivalent en XDG. GUST/GUP étant un formalisme d'unification de structures, il ne s'est pas directement prêté à une interprétation sous forme de contraintes, et il nous a fallu procéder à un travail d'**axiomatisation** de nos grammaires.

4. Une fois cette étape terminée, nous nous sommes lancés dans l'**implémentation** proprement dite de notre compilateur, que nous avons baptisé `auGUSTe`. Nous avons également dû modifier substantiellement XDK pour l'ajuster aux particularités de notre interface, ce qui s'est notamment concrétisé par l'ajout de 8 nouveaux "principes" (i.e. des ensembles de contraintes). Au final, notre implémentation comprend plusieurs milliers de lignes de code, écrites en Python (pour le compilateur) et en Oz (pour les contraintes XDG).

5. La dernière étape fut consacrée à la **validation expérimentale** de notre travail. Nous avons ainsi élaboré une mini-grammaire, centrée sur le vocabulaire culinaire et constituée d'environ 900 règles, ainsi qu'une batterie de tests de 50 graphes sémantiques à générer.

[6]Les progrès effectués chaque mois - nonobstant les derniers qui étaient quasi exclusivement consacrés à l'implémentation - ont ainsi fait l'objet de rapports d'avancements réguliers, et d'une correspondance soutenue avec divers chercheurs.

[7]Quelques précisions terminologiques : GUST désigne une *théorie* linguistique particulière, tandis que GUP est un *formalisme* générique de description linguistique. Nous aurons bien sûr amplement l'occasion de détailler la signifiation précise de ces acronymes par la suite.

1.3 De la modélisation en linguistique

1.3.1 Motivation

Avant d'aborder le vif du sujet, il nous semble important de clarifier certaines hypothèses méthodologiques qui ont présidé à ce travail. Plus spécifiquement, les notions de *modélisation* et de *formalisation* sont au coeur de notre travail, mais qu'entendons-nous exactement par là ?

La notion de **modèle** est absolument fondamentale en sciences. Lorsqu'un scientifique ne peut accéder directement à la structure interne d'un objet, pour des raisons empiriques (donnée inobservable ou non mesurable) ou épistémologiques (domaine trop complexe ou incohérent), celui-ci a recours à un *modèle*, c'est-à-dire une *représentation* schématique, simplifiée du phénomène. Cette représentation peut être exprimée de diveres manières, qui vont du simple jargon technique à l'utilisation de structures mathématiques. Elle est alors utilisée pour concevoir une **théorie**, dont l'objectif est d'extraire un ensemble de lois, de principes, de règles gouvernant le fonctionnement du domaine étudié. La théorie ne manipule pas directement le phénomène empirique, elle ne peut en parler qu'à travers l'**interprétation** qu'en donne le modèle[8].

Muni de ce modèle et de cette théorie, le scientifique pourra alors chercher à **prédire** le comportement du phénomène en question. Le grand avantage lié à l'utilisation de modèles précis et rigoureusement spécifiés réside précisément là : un modèle informel, schématique ne pourra rendre compte de l'évolution d'un phénomène que de manière vague et ambigüe, alors qu'un modèle formel sera capable d'offrir une prédiction précise. Et ceci est d'une importance capitale du point de vue scientifique, puisque cette prédiction pourra être comparée à des mesures directes effectuées sur le phénomène, et ainsi être éventuellement *falsifiée* (au sens popperien du terme).

Il est à nos yeux *essentiel* que la linguistique s'outille d'un **appareil conceptuel formalisé**, i.e. d'une terminologie précise, rigoureuse, complétée par des modélisations formelles. Nous voyons au moins quatre arguments pour soutenir cette position :

1. L'étude des phénomènes linguistiques a pris, depuis les années 60, une telle ampleur qu'elle ne peut se baser sur la seule utilisation d'un jargon technique plus ou moins intuitif, sous peine d'imprécisions inacceptables. Trop souvent, les linguistes s'empêtrent dans des controverses stériles par manque d'une terminologie unifiée - voir par exemple (Mel'čuk, 1997, p. 35) concernant la description des déclinaisons des langues nilotiques du sud (massaï, turkana, teso, etc) réalisée par les africanistes contemporains.

2. « La linguistique est la seule science actuelle dont l'objet coïncide avec le discours qu'elle tient sur lui » (Hagège, 1986). Les linguistes se servent en effet des langues naturelles pour parler des langues naturelles. Ceci a pour conséquence d'enfler encore les problèmes terminologiques mentionnés au point précédent.

3. La plupart des objets linguistiques sont "inobservables" : seul le signal acoustique est directement accessible à l'observation, le reste est constitué de structures "invisibles" (morphologie, syntaxe, sémantique), puisqu'il nous est bien évidemment impossible "d'ouvrir les crânes" des locuteurs pour comprendre la manière dont la langue est stockée et traitée dans le cerveau ! L'intérêt et l'adéquation de ces structures n'est donc évaluable qu'indirectement.

[8] En ce qui concerne la mécanique des corps célestes par exemple, une modélisation classique consiste à représenter la position et la vitesse d'un corps quelconque par deux vecteurs au sein d'un espace Euclidien, sa masse par un nombre réel, et son mouvement par un champ vectoriel continu (un "flux"). Il s'agit donc d'une structure mathématique d'un certain type, qui assume une correspondance entre des données observables du phénomène analysé et des éléments du modèle. Un tel modèle n'a pas pour objectif d'être un *fac simile* du phénomène initial, il cherche seulement à représenter les aspects du phénomène qui sont relevants pour le physicien.

La *théorie formelle* sous-jacente à ce modèle est constituée de la logique du 1er ordre et d'un système d'équation différentielles (Hamiltoniennes) exprimant les contraintes sur les flux.

4. Enfin, l'utilisation de structures formalisées est évidemment un préalable indispensable à tout traitement automatique, et il semble aujourd'hui impensable de faire de la linguistique théorique en restant coupé des développements récents du TALN.

Nous rejetons donc vigoureusement l'idée, malheureusement encore bien présente dans certains cénacles, selon laquelle la linguistique "n'a pas besoin d'être formalisée". Lors de l'élaboration d'une théorie, il nous semble crucial de construire des modèles précis, basés sur un métalangage unifié, et spécifiant *explicitement* la nature des constructions linguistiques manipulées.

Bien sûr, la nécessité de modéliser et formaliser ne doit pas occulter le fait que ces activités n'ont de sens que si elles sont développées à partir des **données linguistiques**. Il existe en effet une autre dérive, tout aussi réelle et dangereuse, qui consiste à construire des modèles dans le principal souçi d'être "mathématiquement élégant" et de faire coller à tout prix les données de départ au formalisme, quitte à leur faire quelques infidélités.

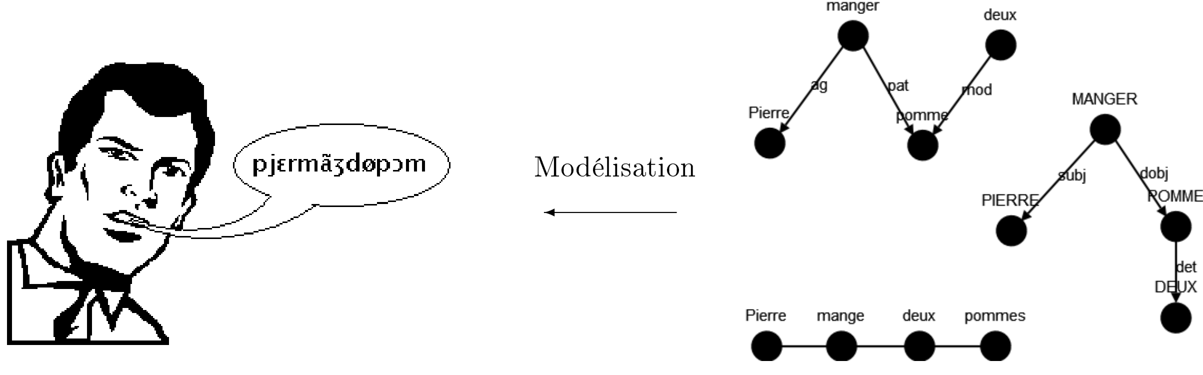
1.3.2 Application

Le travail que nous avons effectué dans le cadre de ce mémoire s'efforce de respecter au mieux ces impératifs. La figure 1.2, inspirée de (Pollard et Sag, 1994), résume notre démarche. Notre *modèle* est constitué d'un ensemble de structures polarisées dont la construction est assurée par un seul et même formalisme, les *Grammaires d'Unification Polarisées*, que nous détaillerons à la section 3.6.

La *théorie* sur laquelle nous avons travaillé s'intitule *Grammaire d'Unification Sens-Texte* (voir chapitre 3). Son l'objectif est de décrire un ensemble de phénomènes linguistiques par le biais d'un ensemble de règles contrôlant la bonne formation et la cohérence d'un ensemble de structures polarisées, représentant chacune un niveau d'analyse linguistique.

Phénomène empirique :
Enoncé linguistique quelconque

Modèle :
Structures (graphes, arbres, séquences) engendrées par des *Grammaires d'Unification Polarisées*



Théorie scientifique :
Règles issues de la *Grammaire d'Unification Sens-Texte*

FIG. 1.2 – Articulation entre phènomène, modèle et théorie

Notre choix de travailler sur cette théorie linguistique plutôt qu'une autre n'est pas anodin. En épluchant divers articles consacrés aux théories linguistiques contemporaines, nous sommes tombés par hasard sur un texte présentant les fondements théoriques de GUST et avons été frappés par la rigueur (autant linguistique que mathématique) qui se dégageait de ces travaux, et qui s'accordait parfaitement à notre vision de la recherche en TALN : une formalisation à la fois linguistiquement pertinente et mathématiquement fondée, lexicalisée, puisant son inspiration dans une théorie particulièrement riche et sophistiquée (la théorie sens-texte), et, *last but not least*, prenant en compte tous les niveaux de la langue.

Clôturons cette section par une citation de S. Kahane, le concepteur de GUST/GUP, qui résume parfaitement notre point de vue : « La formalisation n'a pas toujours bonne presse en linguistique, car elle consiste souvent en un encodage brutal et réducteur dans des formalismes complexes reposant sur des présupposés théoriques erronés. Une bonne formalisation doit au contraire être fidèle aux concepts sous-jacents et les mettre en lumière. Il faut simplement se donner les moyens (mathématiques) de décrire les choses comme on a envie de les décrire. La mathématisation ne doit pas être un appauvrissement de la pensée. » (Kahane, 2002, p. 73)

1.4 Structure du mémoire

Outre l'introduction et la conclusion, le texte de ce mémoire est divisée en six chapitres :

- Le **chapitre 2** s'attache à présenter le cadre théorique général de ce mémoire ; nous y introduisons les *grammaires de dépendance* et la *théorie Sens-Texte* [TST], théorie linguistique constituant l'inspiration principale du formalisme grammatical sur lequel notre travail se fonde.

- Le **chapitre 3** est quant à lui consacré à présenter les *Grammaires d'Unification Sens-Texte* [GUST], un récent modèle linguistique élaboré par S. Kahane, et les *Grammaires d'Unification Polarisées* [GUP], formalisme générique de description linguistique qui est actuellement utilisé pour réécrire et reformuler rigoureusement GUST.

- Nous passons ensuite au **chapitre 4**, consacré à la présentation d'un traitement possible de la *portée* des quantificateurs, ainsi qu'à détailler une série de *modélisations* originales que nous avons élaborées concernant divers phénomènes linguistiques situés à l'interface entre la sémantique et la syntaxe.

- Une fois assises les bases théoriques de notre travail, nous poursuivons notre parcours en présentant au **chapitre 5** notre *axiomatisation* de GUST/GUP en un problème de satisfaction de contraintes, à intégrer dans le logiciel XDK.

- Notre travail d'*implémentation* de l'interface sémantique-syntaxe est détaillé au **chapitre 6**. Nous expliquons le fonctionnement général de notre compilateur `auGUSTe` et des contraintes XDG que nous avons développées.

- Enfin, nous illustrons au **chapitre 7** la *validation expérimentale* (via un mini-corpus centré sur le vocabulaire culinaire et un corpus de graphes sémantiques) appliquée à notre implémentation pour en évaluer la qualité.

Bonne lecture !

Chapitre 2

Grammaires de dépendance et théorie Sens-Texte

Ce chapitre sera consacré à bâtir la “charpente linguistique” de ce mémoire, i.e. à présenter succintement les grands principes des théories linguistiques qui sont à la base des *Grammaires d'Unification Sens-Texte*, le formalisme grammatical que nous avons cherché à axiomatiser et implémenter. Vu la diversité et sophistication des théories linguistiques contemporaines, nous ne pourrons évidemment qu'effectuer un survol de cette matière ; nous nous sommes néammoins efforcés de présenter de façon claire et précise les orientations fondamentales, les hypothèses sous-jacentes, les choix de modélisation.

Nous commencons par présenter succintement les grammaires de dépendance, et poursuivons par la Théorie Sens-Texte, une théorie linguistique particulièrement riche et intéressante du point de vue du TALN, et utilisant les grammaires de dépendance comme représentation syntaxique.

2.1 Grammaires de Dépendance

2.1.1 Généralités

Si l'on se place du point de vue du type de structures manipulées, les formalismes grammaticaux peuvent être classés en deux grandes catégories :

- Les **grammaires syntagmatiques** - ou grammaires de constituants - [GS] ;
- Les **grammaires de dépendance** [GD].

Les grammaires syntagmatiques ont été popularisées par Noam Chomsky (Chomsky, 1957; Chomsky, 1965) et ont été par la suite adoptées par bien d'autres formalismes, tels *Government and Binding* (Chomsky, 1981) , *Lexical Functional Grammar* (Bresnan, 1982), *Head-Driven Phrase Structure Grammar* (Pollard et Sag, 1994) et *Tree Adjoining Grammar* (Joshi, 1987).

Typiquement, une analyse en GS divise une phrase en sous-chaînes continues appelées *syntagmes* ou *constituants*, étiquetées par des catégories syntaxiques comme *Ph* (Phrase), *SN* (syntagme nominal), *SAdj* (syntagme adjectival), etc, qui sont arrangées hiérarchiquement dans un arbre syntagmatique.

Les GD adoptent une perspective différente : une analyse syntaxique a pour but de déterminer comment la façon dont les mots, par leur présence, *dépendent* les uns des autres (d'où le nom).

2.1.2 Définitions formelles

D'un point de vue mathématique, un **arbre** peut être défini de deux manières équivalentes :

1. comme un graphe orienté ;

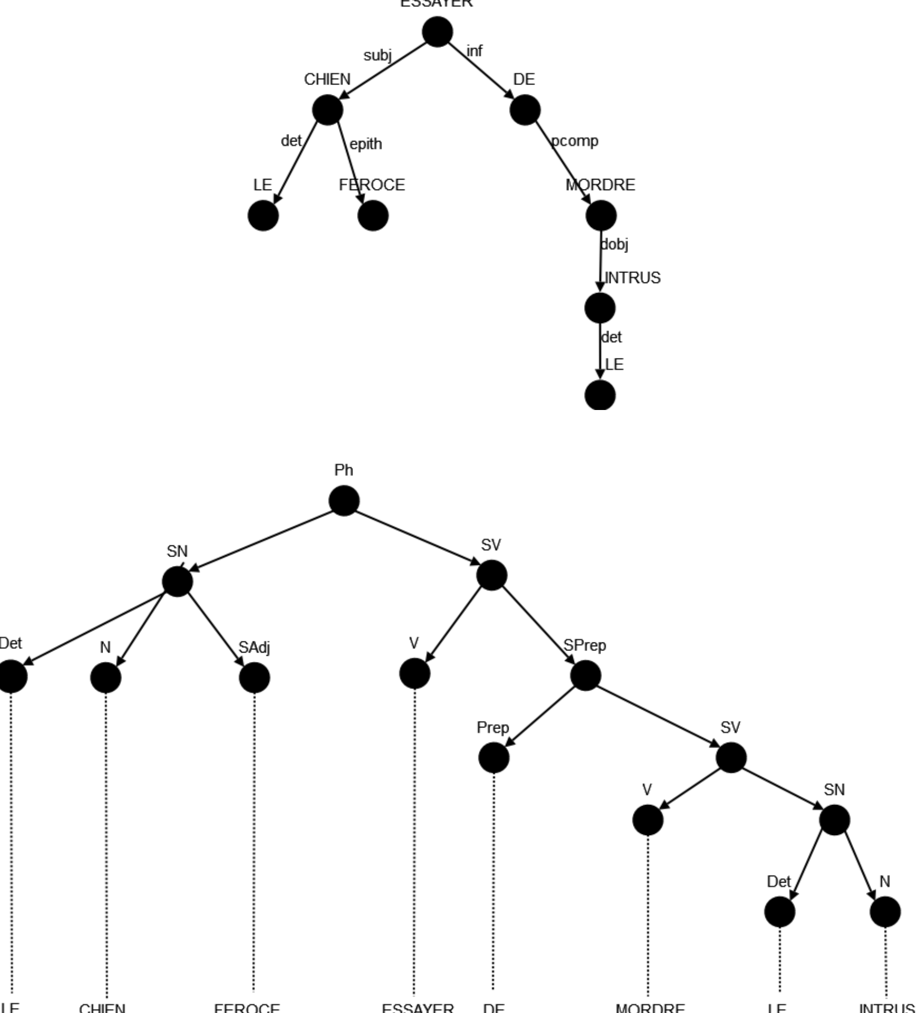


FIG. 2.1 – Comparaison entre deux analyses syntaxiques d'une même phrase, via une grammaire de dépendance (haut) et une grammaire syntagmatique (bas)

2. comme une relation binaire $\triangleleft$, appelée **relation d'arbre**. $x \triangleleft y$ si et seulement si (y, x) est un lien dans le graphe correspondant, avec x et y noeuds du graphe.

Une relation d'arbre induit une **relation de dominance** $\preceq$ définie comme suit : $x \preceq y$ si et seulement si $x = x_1 \triangleleft x_2 \triangleleft .... \triangleleft x_n = y$ $(n \geq 0)$. La **racine** de l'arbre est le seul noeud qui domine tous les autres noeuds. Un **noeud terminal** est un noeud ne possédant pas de dépendant.

Soit X un ensemble quelconque de lexèmes (unités syntaxiques), nous définissons ci-dessous les notions d'arbre syntagmatique et de dépendance :

Définition. Un **arbre syntagmatique** sur X est un quadruplet $(X, \mathfrak{B}, \phi, \triangleleft)$, où $\mathfrak{B}$ est l'ensemble des constituants (ou syntagmes), $\triangleleft$ est une relation d'arbre définie sur $\mathfrak{B}$, et ϕ est une fonction (qui décrit en fait le contenu des constituants) de $\mathfrak{B}$ vers les sous-ensembles non-nuls de X , telle que :

1. $\forall \alpha \in \mathfrak{B}, \phi(\alpha)$ est une sous-chaîne continue de X ;
2. Chaque sous-ensemble de X constitué d'un seul élément est le contenu d'un et un seul noeud terminal ;
3. Si $\alpha \triangleleft \beta$, alors $\phi(\alpha) \subseteq \phi(\beta)$

Définition. Un **arbre de dépendance** sur X est un arbre simple (angl. "*plain tree*") sur X, défini par le couple $(X, \triangleleft)$. Précisons que les noeuds de cet arbre sont étiquetés par des catégories grammaticales, et que les relations sont étiquetées par des fonctions syntaxiques.

Notons bien sûr que les relations d'arbre $\triangleleft$ n'ont pas les mêmes significations dans les deux formalismes :
- Pour les arbres syntagmatiques, $\alpha \triangleleft \beta$ représente le fait que le constituant α (par ex., $SAdj$) est inclus dans le constituant β (par ex., SN) ;
- Pour les arbres de dépendance, $\alpha \triangleleft \beta$ représente le fait que le mot α (par ex. "`féroce`") dépend du mot β (par ex. "`chien`").

2.1.3 Notion de dépendance

La notion même de dépendance en linguistique est assez ancienne - cfr. par exemple les grammaires arabes du 8e siècle (Owens, 1988) -. La première théorie linguistique moderne centrée sur la dépendance est celle de Lucien Tesnière dans (Tesnière, 1959), et de nombreuses autres l'ont suivi, principalement en Europe[1].

Les grammaires de dépendance ont toutes en commun le fait que la structure syntaxique y est déterminée par un ensemble de relations qui expriment la façon dont les mots *dépendent* les uns des autres. La dépendance est une relation orientée, où le mot "source" est appelé *tête* ou *gouverneur* et le mot "cible" *dépendant* ou *gouverné*. Les relations se situent donc uniquement entre des mots et non entre des constituants (ou syntagmes) comme dans les GS.

Bien sûr, chaque noeud de l'arbre est étiqueté par une structure de traits simples (contenant le nom de la lexie, la catégorie lexicale, etc.). Les arcs sont également étiquetés par une structure de traits indiquant la fonction syntaxique du lien, et éventuellement d'autres informations.

Par manque de place, nous ne pouvons pas discuter plus avant la caractérisation théorique de la notion de dépendance, et vous renvoyons notamment à (Mel'čuk, 1988).

2.1.4 Syntagme *vs.* dépendance

Depuis la fin des années des années 70, la plupart des modèles linguistiques issus de la mouvance chomskienne (GB, LFG, HPSG), ainsi que les grammaires catégorielles et les TAG ont introduit l'usage de la dépendance syntaxique sous des formes plus ou moins explicites (fonctions syntaxiques, constituants avec tête, cadre de sous-catégorisation, c-commande, structures élémentaires associées aux mots dans le cas des grammaires lexicalisées). D'un point de vue purement formel, les structures syntagmatiques avec tête et les structures de dépendance sont d'ailleurs mathématiquement équivalentes (Robinson, 1970) - comme nous le montrons ci-après - même si elles sont, dans la pratique linguistique, utilisées de manière différente[2].

Thèse : Tout arbre de dépendance $(X, \triangleleft_1)$ induit un arbre syntagmatique $(X, \mathfrak{B}, \phi, \triangleleft_2)$

Démonstration. Pour chaque noeud x de l'arbre de dépendance, nous créons deux constituants, notés f_x et p_x, tels que $\phi(f_x) = \{x\}$ et $\phi(p_x)$ est la projection de x, c'est-à-dire l'ensemble des

[1] Citons par exemple la *Functional Generative Grammar* de l'école de Prague (Sgall *et al.*, 1986), la *Word Grammar* de Hudson (Hudson, 1990) en Angleterre, la grammaire de l'allemand de (Engel, 1988) et enfin la théorie Sens-Texte initiée par Igor Mel'čuk (Mel'čuk, 1974; Mel'čuk, 1988) , qui constitue l'inspiration principale du formalisme grammatical sur lequel nous avons travaillé.

[2] Citons notamment le fait (sur lequel nous reviendrons) que les GD distinguent les questions portant d'une part sur la structure syntaxique ("dominance immédiate") et d'autre part sur l'ordre des mots ("précédence linéaire" ou topologie), alors que les GS analysent ces phénomènes d'une seule traite. A cela se rajoute des différences plus fondamentales liées à la conception même du langage (en particulier sur la nature des phénomènes syntaxiques) : les GS marquent ainsi un rejet très net - et à mon sens scientifiquement peu fondé - des questions de sémantique et de lexicologie - voir notamment (Gross, 1975) à propos des "irrégularités" lexicales.

noeuds y tels que $y \preceq x$ (= ensemble des noeuds dominés par x).

Bien sûr, $\mathfrak{B}$ se définira comme l'ensemble composé des f_x et p_x pour tout $x \in X$. La fonction ϕ est également déjà définie. Il reste à déterminer la relation $\lhd_2$.

Celle-ci se définit simplement comme la relation $\lhd_1$ sur les constituants "projectifs" (en d'autres termes, on a $x_1 \lhd_1 x_2 \Rightarrow p_{x_1} \lhd_2 p_{x_2}$, pour $x_1, x_2 \in X$), et de plus $f_x \lhd_2 p_x$ pour tout $x \in X$.

Nous avons alors construit un quadruplet $(X, \mathfrak{B}, \phi, \lhd_2)$, soit un arbre syntagmatique. □

Notons qu'en toute rigueur, la réciproque n'est pas vraie : pour qu'un arbre syntagmatique puisse induire un arbre de dépendance, il faut lui rajouter une information supplémentaire : les **têtes lexicales**, indiquant pour chaque constituant l'élément "central" de celui-ci (par ex. *SV* aura *V* comme tête lexicale).

On obtient alors un **arbre syntagmatique avec tête**, qui est une structure très fréquemment utilisée dans les théories linguistiques modernes (GB, LFG, HPSG), et qui se définit comme un quintuplet $(X, \mathfrak{B}, \phi, \lhd, \tau)$, où τ dénote une fonction de $\bar{\mathfrak{B}}$ (= le sous-ensemble de $\mathfrak{B}$ constitué de ses éléments non-terminaux) vers un élément de $\mathfrak{B}$, tel que $\tau(\alpha) \lhd \alpha$ pour tout $\alpha \in \bar{\mathfrak{B}}$. En d'autres termes, cette fonction τ détermine pour chaque constituant non terminal un noeud fils qui sera nommé sa tête lexicale. On peut alors montrer qu'un arbre syntagmatique avec tête induit un arbre de dépendance, voir (Robinson, 1970).

Initialement victime du succès des GS, le retour en grâce de la notion de dépendance, très marqué ces dernières années, est principalement dû à

- la prise de conscience de l'importance du **lexique** pour l'étude de la syntaxe. Les grammaires de dépendance placent en effet la *lexie* au centre de la structure syntaxique, et permettent d'exprimer de manière simple et directe des relations lexixales comme la valence ou le régime.
- L'intérêt renouvelé pour la **sémantique** : les structures de dépendance permettent de dissocier la question de l'ordre linéaire des mots et la structure syntaxique, et se rapprochent donc davantage de la représentation sémantique qu'une structure syntagmatique.

2.1.5 Ordre des mots

Les arbres de dépendance ne sont pas ordonnés. La question de l'ordre linéaire des mots (la topologie) n'est bien sûr pas oubliée, mais est traitée de manière distincte, comme un module supplémentaire intervenant après l'analyse syntaxique proprement dite. Les GD ont ainsi le grand avantage de traiter de façon beaucoup plus élégante les nombreuses langues dont l'ordre linéaire est partiellement libre, comme l'allemand ou le russe. Dans ces langues, l'ordre des mots dépend davantage de la structure communicative (angl. *information structure*) que de la syntaxe proprement dite. Ainsi, en allemand, la traduction de ‹j'ai lu un livre hier› :

"`Ich habe gestern ein Buch gelesen`" (2.1)

peut également se dire :

"`Gestern habe ich ein Buch gelesen`" (2.2)

"`Ein Buch habe ich gestern gelesen`" (2.3)

selon l'intention communicative du locuteur (veut-il mettre l'accent sur le fait qu'il a lu un livre, qu'il l'a lu hier, ou que c'est un livre qu'il a lu, et pas autre chose ?).

Notons que ce phénomène se rencontre même pour des langues à ordre plus rigide (Steele, 1978) comme l'anglais ou le français, par exemple en ce qui concerne les *wh*-questions, les topicalisations ou les clivages : la phrase "`Pierre va étudier en Allemagne l'année prochaine`" peut ainsi se reformuler :

"`C'est en Allemagne que Pierre va étudier l'année prochaine`"; (2.4)

"Pierre, il va étudier l'année prochaine en Allemagne". (2.5)

et bien d'autres expressions sont évidemment possibles. Les grammaires syntagmatiques ont de grosses difficultés à modéliser correctement ce genre de phénomènes car seule des sous-chaînes *continues* peuvent être arrangées hiérarchiquement. Ils faut alors recourir à des "astuces" plus ou moins compliquées telles que l'utilisation de traces (*GB*) ou la percolation de traits (*HPSG*).

Nul besoin de ce genre de bricolage dans les grammaires de dépendance. Soit par exemple la phrase latine "Tantae molis erat Romanam condere gentem" ‹tant était lourde la tâche de fonder la nation romaine› (Virgile, Enéide, I-33), où une discontinuité apparaît : "Romanam" et "gentem" sont bien sûr syntaxiquement liés (l'adjectif et son nom), mais sont séparés sur la chaîne linéaire (pour des raisons de métrique poétique). L'arbre de dépendance de cette phrase est illustré ci-dessous :

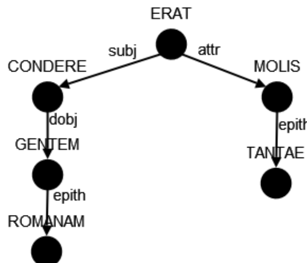


Fig. 2.2 – Analyse dépendancielle du vers "Tantae molis erat Romanam condere gentem".

2.1.6 Projectivité

Si la structure de dépendance n'encode pas explicitement l'ordre des mots, comment le modèle pourra-t-il déterminer la manière de linéariser correctement la phrase ? Cette question est étroitement liée à celle de la *projectivité* d'un arbre de dépendance. Nous commencons par définir formellement la notion de projectivité, et détaillons ensuite sa signification linguistique.

Définition formelle

Soit un **ordre hiérarchique** (en d'autres termes, la structure dépendancielle) $\prec$ entre les noeuds d'une arbre T. Soit x et y deux noeuds de l'arbre T, avec $x \prec y$; nous appelons x un **descendant** de y et y un **ancêtre** de x. Nous définissons aussi la **projection** de x comme l'ensemble des noeuds y tels que $y \preceq x$ (remarquons au passage que x fait donc partie de sa propre projection).

Nous noterons également $\vec{x}$ un arc dirigé de y vers x appartenant à l'arbre T. Le noeud y sera alors appelé le **gouverneur** et x un **dépendant**. Le noeud y pourra par ailleurs être noté x^+. Les notations $\vec{x}$ et x^+ ne sont en rien ambigues car, de par la définition d'un arbre, un noeud x ne possède qu'un seul gouverneur.

Un **arbre ordonné** est un arbre assorti d'un ordre linéaire sur l'ensemble de ses noeuds. Soit $\vec{x}$ est un arc appartenant à un arbre ordonné T, nous définissons le **support** de $\vec{x}$, $Supp(\vec{x})$ comme l'ensemble des noeuds de T situé entre les extrémités de $\vec{x}$, i.e. entre le gouverneur et le dépendant, ceux-ci inclus. Nous dirons que les éléments de $Supp(\vec{x})$ sont **couverts** par $\vec{x}$.

Définition. Un arc $\vec{x}$ est dit **projectif** si et seulement si pour chaque noeud y couvert par $\vec{x}$, on a $y \preceq x^+$. Un arbre T est dit projectif si et seulement si chaque arc de T est projectif.

Signification linguistique

De manière plus informelle, on peut vérifier si un arbre de dépendance assorti d'un ordre linéaire est projectif en plaçant les noeuds sur une ligne droite et tous les arcs dans le même demi-plan, et en vérifiant que (1) deux arcs ne se coupent jamais et que (2) aucun arc ne couvre un de ses ancêtres ou la racine de l'arbre - voir figures 2.3, 2.4, 2.5, 2.6 pour des exemples.

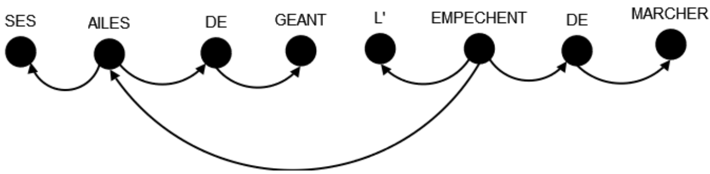


FIG. 2.3 – Arbre de dépendance parfaitement projectif

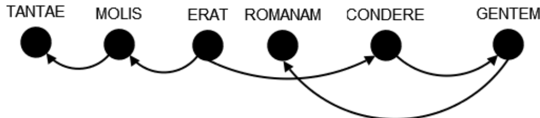


FIG. 2.4 – Arbre de dépendance *non* projectif : "`condere`" est couvert par l'arc allant de "`gentem`" à "`romanam`", mais n'est pas un dépendant de "`gentem`"

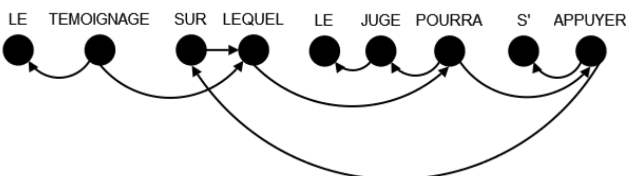


FIG. 2.5 – Arbre de dépendance *non* projectif : "`pourra`" est couvert par l'arc allant de "`appuyer`" à "`sur`", mais n'est pas un dépendant de "`appuyer`"

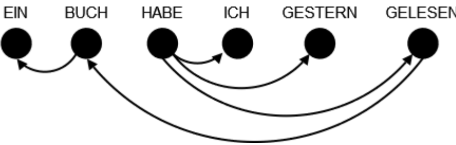


FIG. 2.6 – Arbre de dépendance *non* projectif : "`habe`" est couvert par l'arc allant de "`gelesen`" à "`buch`", mais n'est pas un dépendant de "`gelesen`"

En français, les principales sources de constructions non projectives sont les dépendances non bornées (dont les fameuses "extractions" : relativisation, interrogation directe et indirecte, etc.) et la montée des clitiques. Les langues germaniques en possèdent bien d'autres : *scrambling, VP fronting, extraposition, Oberfeldumstellung, cross serial dependencies, etc.* Il s'agit donc d'un phènomène linguistique non négligeable.

Remarquons au passage que la projectivité dans le cadre des grammaires de dépendance correspond à la *continuité des constituants* dans le cadre des grammaires syntagmatiques. La seule différence (mais elle est de taille) entre ces deux concepts réside dans le caractère *obligatoire* de cette propriété dans les GS, tandis que la projectivité est *optionnelle* dans les GD.

Au niveau des techniques d'analyse et de synthèse, la projectivité présente un intérêt évident : il suffit, pour ordonner un arbre projectif, de spécifier la position de chaque noeud par rapport à son gouverneur, ainsi que vis-à-vis de ses frères. Un arbre projectif est donc facilement linéarisable par un algorithme. En fait, on peut montrer (Gaifman, 1965) qu'une grammaire qui ne traite que de structures projectives est équivalente à une grammaire de réécriture hors-contexte, et permet donc comme elles une analyse en $O(n^3)$, où n est la longueur des phrases.

Pour traiter adéquatement les constructions non projectives et les langues dites "à ordre libre", il sera nécessaire d'enrichir le formalisme initial - nous présenterons ainsi succintement dans le chapitre suivant les "arbres à bulles" qui permettent de modéliser les phénomènes d'extraction et de coordination. En ce qui concerne les aspects computationnels, voir également (Kahane *et al.*, 1998) pour un algorithme polynomial d'analyse de structures non-projectives, basé sur la notion de "pseudo-projectivité".

2.1.7 Théorie de la translation de Tesnière

L'oeuvre de Tesnière (Tesnière, 1959) est bien connue pour avoir été la première théorie linguistique moderne centrée sur la notion de dépendance, mais sa théorie de la translation (qu'il considérait comme sa découverte principale) est souvent oubliée. Selon Tesnière, la catégorisation traditionnelle en 10 parties du discours [PdD ci-après], issue de la syntaxe latine, est bancale et inutilisable. Il propose donc une catégorisation originale, caractérisée par 4 PdD majeures : verbe, nom, adjectif, adverbe, avec des relations prototypiques entre ces PdD : les actants du verbe sont des noms et ses modifieurs des adverbes, les dépendants du nom sont des adjectifs et les dépendants de l'adjectif et de l'adverbe sont des adverbes. Néammoins, un élément de PdD X peut venir occuper une position normalement réservée à un élément de PdD Y mais dans ce cas, l'élément doit être *translaté* de la PdD X à la PdD Y par un élément morphologique ou analytique appelé un *translatif* de X en Y. Par exemple :

- Un verbe peut être l'actant d'un autre verbe (c'est-à-dire occuper une position nominale), mais il devra être à l'infinitif ou être accompagné de la conjonction de subordination *que* : "`Jean affirme sa conviction`", "`Jean affirme croire aux extra-terrestres`", "`Jean affirme qu'il croit aux extra-terrestres`". L'infinitif et la conjonction de subordination *que* sont donc des translatifs de verbe en nom ;

- Les participes passé et présent, qui permettent à un verbe de modifier un nom : "`Le poignant témoignage`", "`Le témoignage enregistré par la police`", "`Le témoignage visant le ministre`" ;

- La copule peut être à son tour vue comme un translatif d'adjectif en verbe : "`Ce témoignage est poignant`", "`Ce témoignage est enregistré par la police`" ;

- Les prépositions sont quant à elles classées comme des translatifs de nom en adjectif ou en adverbe : "`Le témoignage de cet individu`", "`Il a témoigné avec une rare assurance`".

(Explication adaptée de (Kahane, 2001))

Lors de la conception de notre grammaire, nous nous sommes inspirés - toutes proportions gardées - de cette théorie de la translation pour l'élaboration d'une liste de fonctions syntaxiques et la modélisation de certains phénomènes grammaticaux.

2.1.8 Fonctions syntaxiques

Chaque lien de dépendance sera étiqueté par une fonction syntaxique particulière. La discussion sur le nombre et la nature des fonctions syntaxiques à l'oeuvre dans chaque langue étant un sujet intéressant mais particulièrement ardu sur le plan linguistique, nous ne nous attarderons pas dans le cadre de ce mémoire. La table 2.1 donne les fonctions syntaxiques principales que nous utilisons dans notre interface sémantique-syntaxe. Notons que nous limitons notre analyse à la phrase simple (pas d'extraction, pas de coordination).

epith	**Epithète du nom** : - Adjectif (ex : "`un sujet` *`intéressant`*") - Complément non adjectival, introduit par une préposition (ex : "`le mémoire` *`de Pierre`*", "`la confiture` *`aux abricots`*")
acirc	**Complément circonstanciel de l'adjectif ou de l'adverbe** : - Adverbe (ex : "`Une` *`très`* `belle femme`") - Complément non adverbial, introduit par une préposition (ex : "`Un homme capable` *`de tout`*", "`Un plat riche` *`en miel`*")
attr	**Attribut du sujet de la copule** :[3] - Attribut adjectival (ex : "`Le tablier est` *`rouge`*") - Attribut substantival (ex : "`Les borgnes sont` *`rois`*") - Attribut adverbial (ex : "`Ils sont` *`ensemble`*") - Attribut introduit par une préposition (ex : "`Il reste` *`de marbre`*")
aux	**Complément verbal de l'auxiliaire** : (ex : "`Il a` *`englouti`* `sa soupe`")
det	**Déterminant du nom** : (ex : "*`la`* `pomme`", "*`sa`* `réserve`")
dobj	**Objet direct du verbe**, correspondant à l'accusatif : - Substantif (ex : "`Marc épluche` *`les patates`*"). - Clitique : "`me`", "`le`", "`la`", etc. (ex : "`Marc` *`les`* `épluche`")
inf	**Complément verbal à l'infinitif**, dépendant d'un verbe de contrôle ou de montée (ex : "`Pierre veut` *`manger`*", "`Pierre commence` *`à manger`*")
iobj	**Complément indirect du verbe**, correspondant au datif et généralement introduit par la préposition "`à`" : - Substantif (ex : "`Marc donne ses épinards` *`au chien`*"). - Clitique : "`me`", "`lui`", "`leur`", etc. (ex : "`Marc` *`lui`* `donne ses épinards`")
obl	**Complément oblique du verbe**, i.e. un objet introduit par une préposition : - Complément d'agent pour la diathèse passive, introduit par "`de`" ou "`par`" (ex : "`Le poisson est mangé par Pierre`", "`Bernard est aimé de Marie`") - Complèment de thème, introduit par "`de`" (ex : "`Il m'a parlé` *`de ses problèmes`*". NB : clitisation en "`en`" - Locatif "actanciel" (ex : "`Il va` *`à Paris`*"). NB : clitisation en "`y`"

[3]Il existe également une fonction d'attribut de l'objet, comme on peut l'observer dans la célèbre phrase "`Je l'aimais inconstant, qu'aurais-je fait fidèle ?`" (Racine, Andromaque), mais nous ne l'aborderons pas ici.

subj	**Sujet du verbe fini** (ex : "*Le percolateur* ne marche plus")
circ	**Complément circonstanciel du verbe** : - Adverbe (ex : "Il a *encore* raté le match") - Locatif "circonstantiel" (ex : "*A la maison*, je mange souvent du poulet") - Divers compléments circonstantiels introduits par une préposition : compléments de but (ex : "Je ferai tout *pour réussir*"), de manière (ex : "*Avec de tels atouts*, il ne peut perdre"), de temps (ex : "J'ai travaillé sur ce mémoire *durant toute l'année*"), etc. - Complément circonstantiel sans préposition (ex : "Il ira *l'année prochaine* à Saarbrücken")
pcomp	**Complément d'une préposition** : - Substantif (ex : "Après *le travail*, nous irons souper") - Verbe (ex : "Nous sommes sortis pour *prendre un verre*")

Tab. 2.1: Fonctions syntaxiques fondamentales utilisées dans l'interface sémantique-syntaxe

2.2 Théorie Sens-Texte

Les fondements de la Théorie Sens-Texte [TST, angl. *Meaning-Text Theory*] remonte aux années 60, avec les traux d'Igor Mel'čuk[4]. et ses collègues (Žolkovskij et Mel'čuk, 1965; Žolkovskij et Mel'čuk, 1967) en traduction automatique. Moins connue que le générativisme "chomskyien" et ses multiples avatars, elle jouit néammoins depuis quelques années d'un fort regain d'intérêt. Elle se distingue des autres approches formelles par sa grande richesse et sophistication, qui prend en compte tous les niveaux de fonctionnement de la langue.

Par rapport aux théories génératives, la TST se démarque notamment par l'importance qu'elle accorde au **lexique** et aux considérations sémantiques. Une autre originalité consiste en l'utilisation d'**arbres de dépendance** (voir section précédente) et non d'arbres syntagmatiques pour la représentation syntaxique. Nous détaillerons tous ces points dans la suite.

2.2.1 Postulats

Dans sa leçon inaugurale au Collège de France (Mel'čuk, 1997), Igor Mel'čuk résume les principes sous-jacents de sa théorie en 3 postulats : le 1er concerne l'*objet d'étude* de la linguistique Sens-Texte, le 2e le *résultat escompté de l'étude*, et le 3e la *nature de la description* linguistique :

Postulat 1 La langue est un système fini de règles qui spécifie une **correspondance multivoque** entre un ensemble infini dénombrable de sens et un ensemble infini dénombrable de

[4] « Igor Mel'čuk est professeur à l'Université de Montréal et directeur de l'Observatoire de Linguistique Sens-Texte. Il a débuté sa carrière à Moscou dans les années 50 et est l'un des pionniers de la traduction automatique. Eminent scientifique de l'ex-URSS, il a été contraint d'émigrer en 1978 et s'est installé au Canada où on lui a offert un poste de professeur. Il a fondé dans les années 60 la théorie Sens-Texte, qui est aujourd'hui l'une des théories de référence en linguistique formelle et en traitement automatique des langues. Il est l'auteur de nombreux ouvrages dont son "*Cours de Morphologie Générale*" en 5 volumes (Mel'čuk, 1993), les 4 volumes du "*Dictionnaire Explicatif et Combinatoire du Français Contemporain*" (Mel'čuk *et al.*, 1984), et l'ouvrage de référence en grammaire de dépendance, "*Dependency Syntax*" (Mel'čuk, 1988). Igor Mel'cuk a été professeur au Collège de France en 1997 et est membre de l'Académie des Sciences Royale du Canada. » [`http ://talana.linguist.jussieu.fr/igor.html`]

textes[5].

Postulat 2 Une correspondance sens-texte est décrite par un **système formel** simulant l'activité linguistique d'un sujet parlant.

Postulat 3 Cette correspondance est **modulaire** et présuppose au moins deux niveaux de représentation intermédiaires : syntaxique et morphologique. La phrase et le mot sont donc les unités de base de la description.

La linguistique Sens-Texte a donc pour objet la construction de **modèles formels** et **calculables** de la langue.

Concernant le 1er postulat, remarquons la différence entre cette définition de la langue (que nous appellerons *transductive*) et celle fournie par une approche générative, où la langue est vue comme un ensemble infini dénombrable d'énoncés grammaticaux, ou comme la "machine" permettant de générer cet ensemble. Néammoins, les deux définitions sont formellement similaires : il est en effet équivalent de définir une correspondance entre l'ensemble des sens et l'ensemble des textes ou de définir l'ensemble des couples formées d'un sens et d'un texte, un tel couple représentant une phrase (voir section 3.4, page 41 pour une discussion).

Notons également que la TST est universelle, en ce sens qu'elle s'efforce de dégager des principes généraux s'appliquant à toutes les langues, sans en privilégier aucune. Cette démarche contraste avec de nombreux approches actuelles portant assez clairement les "stigmates" de la langue anglaise, qui est assez particulière puisque sa morphologie est assez pauvre, et sa topologie très contrainte par la syntaxe. Le choix d'utiliser une grammaire de dépendance plutôt qu'une grammaire syntagmatique s'explique d'ailleurs en partie par cette volonté-là.

2.2.2 Architecture générale

Le modèle standard de la TST considère 7 niveaux de représentation :

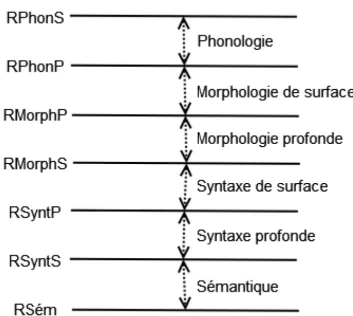


FIG. 2.7 – Niveaux de représentation de la Théorie Sens-Texte.

Dans le modèle "standard" d'Igor Mel'čuk, les différents niveaux de description sont "scindés" en une représentation profonde et une représentation de surface[6]. La raison en est que le passage d'un niveau à l'autre implique deux changements :

- la **hiérarchisation** : Passage d'un structure multidimensionnelle (graphe sémantique) à un structure bidimensionnelle (arbre syntaxique) pour $\mathcal{I}_{sem-synt}$, et d'une structure bidimensionnelle à une structure linéaire (chaîne de morphèmes) pour $\mathcal{I}_{synt-morph}$.

[5]La notion de "texte" est ici à prendre dans un sens très général, qui ne suppose en rien un support écrit.

[6]Dans GUST, cette scission n'est plus représentée de manière explicite, comme nous le verrons au chapitre suivant.

– la **lexicalisation** : Passage des sémantèmes (unités élémentaires “de sens” dans le graphe sémantique) aux lexies (unités lexicales) ou aux grammènes (unités grammaticales) pour $\mathcal{I}_{sem-synt}$, et de ces derniers aux morphèmes pour $\mathcal{I}_{synt-morph}$.

A chaque “module” de la TST correspond évidemment un type de représentation linguistique ; la table 2.2 en donne un exemple.

2.2.3 Représentations formelles

Représentation sémantique

Le sens est défini par la TST comme un “invariant de paraphrase”, c'est-à-dire comme ce qui est commun à toutes les phrases qui ont le même sens.

La représentation sémantique *Rsem* est composée de trois structures :

1. La *structure sémantique*, qui reflète le sens propositionnel de l'énoncé, et constitue le noyau de *Rsem*. Il s'agit d'un graphe orienté dont les noeuds sont étiquetés par des *sémantèmes*. On peut distinguer à l'intérieur de ceux-ci les sémantèmes lexicaux, qui reflètent le sens d'une lexie particulière, dont le signifiant peut-être un mot ou une ensemble de mots (locutions), et les sémantèmes grammaticaux, correspondant à des morphèmes flexionnels (‹singulier›, ‹pluriel›, ‹passif›, ‹futur›, etc.) ;
2. La *structure communicative* (angl. *information structure*) spécifie la façon dont le locuteur veut présenter l'information qu'il communique. Concrètement, la structure communicative subdivise *RSem* en sous-réseaux identifiants les regroupements communicatifs de sens[7].
3. Enfin, la *structure rhétorique* reflète les intentions artistiques du locuteur (ironie, pathétique, niveaux de langage différents,etc.)

La structure sémantique, qui est la seule structure nous intéressant réellement dans le cadre de notre travail, est une structure de type prédicat-argument. Ainsi, ‘aimer’ est un prédicat à deux arguments : ‹X aime Y›. Il est possible de passer de cette structure à une formule de la logique du 1er ordre à condition de connaître également les relations de portée. Nous reviendrons plus tard sur cette question.

Remarque importante : la représentation sémantique de la TST ne se situe pas à un niveau aussi “profond” que celles généralement utilisées en sémantique formelle, issues de la logique Frégéenne (la DRT, notamment), où l'on cherche à indiquer l'*état du monde* dénoté par le sens de la phrase. Ici, on cherche à représenter le sens lui-même (sous forme prédicative) qui est exprimé par les signes lexicaux et grammaticaux, sans chercher à *référer* à l'état du monde[8].

Représentation syntaxique

La représentation syntaxique de la TST fait usage des grammaires de dépendance. Nous avons déjà discuté de celles-ci dans la section précédente, et nous n'y reviendrons donc pas. Mentionnons simplement la distinction, dans le cadre de la TST standard, entre la *RSyntP* ne contenant que des lexies pleines, et la *RSyntS*, contenant également les lexies vides telles les prépositions régies. La *RSyntP* est donc en quelque sorte 'universelle', tandis que la *RSyntS* est propre à chaque langue.

[7]La structure rhème-thème est par exemple encodée par la partition des sémantèmes en deux groupes, et l'utilisation de deux prédicats $theme(x)$ et $rheme(y)$ pointant sur les noeuds dominants x et y de ces deux partitions.

[8]Notons qu'il est théoriquement possible d'ajouter une structure référentielle “au dessus” de la structure prédicative pour effectuer ce lien, mais celui-ci est inutile dans la plupart des applications d'intérêt.

Nom	Exemple de structure	Explication
RSem	'de géant' 'empêcher' 1 1 2 3 'ailes' albatros 'marcher' 1 1	Réseau sémantique : les noeuds représentent des sens et les arcs des relations prédicat-argument.
RSyntP	EMPECHER I II III AILE ALBATROS MARCHER ATTR I I DE GEANT ALBATROS ALBATROS	Arbre de dépendance non linéairement ordonné : les noeuds représentent des lexies pleines et les arcs des dépendances syntaxiques profondes (universelles).
RSyntS	EMPECHER subj dobj inf AILE LE DE det epith pcomp SON DE MARCHER pcomp GEANT	Arbre de dépendance non linéairement ordonné : les noeuds représentent des lexies (pleines ou vides) et les arcs des dépendances syntaxiques de surface (liées à la langue en question).
RMorphP	SON AILE DE GEANT LE EMPECHER DE MARCHER pl pl fem nbre:sg masc masc ind. présent 3ème pl inf	Chaîne de lexies marquées morphologiquement.
RMorphS	s+es+aile+s+de+géant+l'+empêch+ent+de+marcher	Chaîne de morphèmes.
RPhonP	[sɛ ɛl də ʒeɑ̃ lɑ̃pɛʃ də marʃe]	Chaîne de phonèmes.
RPhonS		Chaîne de phones.

TAB. 2.2 – Représentations linguistiques aux différents niveaux de la TST

Une autre particularité du passage entre *RSyntP* et *RSyntS* est l'application des *fonctions lexicales*, que nous discutons ci-dessous.

Représentation morphologique

La représentation morphologique est constituée d'une *suite linéaire* des mots de la phrase. Elle est constituée d'une chaîne morphologique, c'est-à-dire d'une suite de lexies accompagnées d'une liste de grammèmes (genre, nombre, cas,..), et (éventuellement) d'une chaîne prosodique (regroupement de mots en groupes prosodiques et accompagné de marques).

2.2.4 Fonctions Lexicales

Les fonctions lexicales [FL] sont un outil formel - relativement peu connu mais très intéressant - proposé dans le cadre de la TST pour modéliser les relations entre lexies. Cette section est un résumé de l'excellente introduction aux FL dans (Polguère, 1998).

Définition. Une **Fonction lexicale** f décrit une relation entre une lexie L - l'argument de f - et un ensemble de lexies ou d'expressions figées appelé la valeur de l'application de f à la lexie L. La fonction lexicale f est telle que :

1. l'expression $f(L)$ représente l'application de f à la lexie L ;
2. chaque élément de la valeur de $f(L)$ est lié à L de la même façon. Il existe autant de fonctions lexicales qu'il existe de type de liens lexicaux et chaque fonction lexicale est identifiée par un nom particulier.

On distingue encore les FL paradigmatiques et les FL syntagmatiques, comme expliqué ci-dessous.

Fonctions lexicales paradigmatiques

Nous commencerons par la modélisation des liens paradigmatiques[9] :

TAB. 2.3: Fonctions lexicales paradigmatiques

Syn	FL qui associe une lexie à ses synonymes exacts ou approximatifs. Ex : Syn("`voiture`") = "`automobile`" ;
$Syn_{\subset}$	FL qui associe une lexie à ses hyperonymes ;
$Syn_{\supset}$	FL qui associe une lexie à ses hyponymes ;
$Syn_{\cap}$	FL qui associe une lexie à l'ensemble des lexies possédent une intersection de sens avec celle-ci ;
$Anti$	FL qui associe une lexie à ses antonymes ;
S_0	FL qui associe une lexie verbale, adjectivale ou adverbiale, à sa contre-partie nominale ; Ex : S_0("`dormir`") = "`sommeil`" ;
V_0	FL qui associe une lexie nominale, adjectivale ou adverbiale, à sa contre-partie verbale ;
S_i	FL qui lie une lexie prédicative au nom standard de son i^e^ argument. Ex pour ‹X voler Y à Z› : S_1("`voler`") = "`voleur`", S_2("`voler`") = "`butin`", S_3"("voler) = "`victime`" ;

Les FL paradigmatiques encodent donc l'ensemble des *dérivations sémantiques* possibles à l'intérieur du lexique.

[9]En linguistique, le terme "paradigmatique", introduit par (de Saussure, 1916), désigne une connexion entre unités linguistiques à l'**intérieur du lexique** sur base d'un rapport sémantique (synonymie, antonymie, etc.). Par ex., "`livre`" est lié paradigmatiquement à "`lecture`", "`lire`", "`bouquin`", "`librairie`" ,etc.

Fonctions lexicales syntagmatiques

Nous nous tournons à présent vers les FL syntagmatiques[10], permettant de caractériser la combinatoire des lexies, et plus particulièrement des collocations. Nous commencons par définir ce que nous entendons par collocation, et présentons ensuite en quoi les FL permettent de les modéliser.

Définition. L'expression "`AB`" (ou "`BA`"), formée des lexies "`A`" et "`B`", est une **collocation** si, pour produire cette expression, le locuteur sélectionne "`A`" (appelé base de la collocation) librement d'après son sens ⟨A⟩, alors qu'il sélectionne "`B`" (appelé collocatif) pour exprimer un sens ⟨C⟩ en fonction de "`A`". Exemple : "`pleuvoir`$_{(=A)}$ `des cordes`$_{(=B)}$".

Il s'agit donc d'expressions *semi-idiomatiques* : contrairement à des expressions complètement idiomatiques comme "`prendre le taureau par les cornes`" (où le sens de l'expression est indépendant du sens des lexies le constituant), la base de la collocation conserve son sens initial, mais non le collocatif. Ainsi, "`pleuvoir des cordes`", cela reste "`pleuvoir`", mais "`des cordes`" a un sens particulier (signifiant grosso modo ⟨beaucoup⟩) qui ne fonctionne qu'adjoint à la base "`pleuvoir`". Mais le collocatif n'est pas transposable tel quel à une autre base : "`* manger des cordes`", ce n'est pas manger beaucoup !

Les collocations sont un phènomène linguistique très important. Elles sont universellement présentes dans toutes les langues, tant à l'oral qu'à l'écrit. Elles possèdent toutes un caractère arbitraire, ne peuvent pas se traduire mot à mot d'une langue à l'autre et sont donc très difficiles à acquérir.

A titre d'exemple, nous présentons à la table 2.5 une application des FL pour modéliser des collocations en allemand, russe, hongrois, arabe et chinois. On y voit par exemple que des applaudissements ⟨intenses⟩ sont dits ⟨mugissants⟩ en allemands, ⟨tempétueux⟩ en russe, ⟨tourbillonnants⟩ en hongrois, ⟨chauds⟩ en arabe et ⟨de tonnerre⟩ en chinois... et en français, on dit qu'ils sont ⟨nourris⟩ ! De même, l'idée de ⟨partir⟩ en voyage est exprimée par le verbe ⟨faire⟩ en allemand, ⟨accomplir⟩ en russe, ⟨se lever⟩ en arabe, ⟨marcher sur⟩ en chinois.

Modélisation par FL :

Tab. 2.4: Fonctions lexicales syntagmatiques

$Magn$	FL qui associe une lexie à l'ensemble des lexies ou expressions linguistiques qui expriment auprès d'elle (en tant que modificateurs de la base) l'idée d'intensification (⟨intense⟩, ⟨très⟩, ⟨beaucoup⟩, etc.). Exemple : $Magn$("`fumeur`") = "`gros`" (mais "`heavy smoker`", ⟨lourd⟩ en anglais), $Magn$("`courir`") = "`vite`" $<$ "`à fond de train`", "`à perdre haleine`".
Bon	FL qui associe une lexie à l'ensemble des lexies qui expriment auprès d'elle le sens général ⟨bon⟩, ⟨bien⟩... c'est-à-dire marquent l'évaluation positive ou l'approbation du locuteur. Exemple : Bon("`colère`") = "`saine, sainte`".
$AntiBon$	FL qui associe une lexie à l'ensemble des lexies ou expressions linguistiques qui expriment auprès d'elle (en tant que modificateurs de la base) l'idée de ⟨mal⟩, ⟨mauvais⟩, etc.
$Oper_i$	FL qui associe à une lexie prédicative nominale L l'ensemble des verbes supports[11] qui prennent l'expression du i^{e} argument de L comme sujet et prennent L comme complément d'objet direct ou indirect.

[10]Le terme "syntagmatique" désigne une connexion entre unités linguistiques à l'**intérieur de la phrase**, par des relations de combinatoire.

[11]Un verbe support est un collocatif verbal sémantiquement vide dont la fonction linguistique est de "verbaliser" une base nominale, i.e. de la faire fonctionner dans la phrase comme si elle était elle-même un verbe. Ex : "`éprouver`" avec "`regret`".

Prenons l'exemple de la lexie "`coup`". Il s'agit d'un prédicat à deux arguments : ‹`coup de X sur Y`›. Nous avons alors $Oper_1$("`coup`") = "`administrer`", "`asséner`", "`donner`", fam. "`flanquer`", "`porter`". Ces derniers verbes sont en effet bien des verbes supports qui prennent X (1er argument de ‹`coup`›) comme sujet et prennent L comme objet direct.
On a également $Oper_2$("`coup`") = fam. "`encaisser`", fam. "`manger`", "`recevoir`", puisque ces verbes prennent Y comme sujet et prennent L comme *dobj*.
Une présentation graphique de cette FL est donnée à la figure 2.8 (La signification précise du formalisme sera donnée à la section 3.6).

$Func_i$ FL qui associe à une lexie prédicative nominale L l'ensemble des verbes supports qui prennent L comme sujet grammatical et prennent l'expression du i^e argument de L comme complément d'objet direct ou indirect. Lorsque le verbe concerné est intransitif, la fonction lexicale est appelée $Func_0$.
Prenons l'exemple de la lexie "`accusation`". C'est un prédicat à deux arguments : ‹`accusation de X envers Y`›. Nous avons alors $Func_1$("`accusation`") = "`être lancée par`" et $Func_2$("`accusation`") = "`peser`". Autre exemple, permettant d'illustrer l'usage de verbes intransitifs : $Func_0$("`cri`") = "`retentir`".
Une présentation graphique de cette FL est donnée à la figure 2.9.

Il existe au total une cinquantaine de fonctions lexicales, il nous est donc impossible de les parcourir en détail. L'essentiel est d'avoir perçu l'intérêt de cet outil de modélisation linguistique, en particulier en ce qui concerne l'élaboration de lexiques pour le TALN.

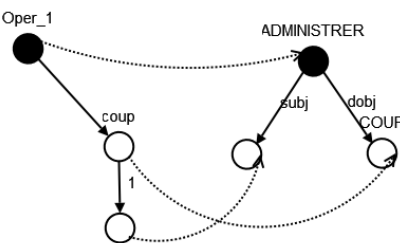


Fig. 2.8 – Présentation graphique de la fonction lexicale $Oper_1$ sur la lexie "`coup`".

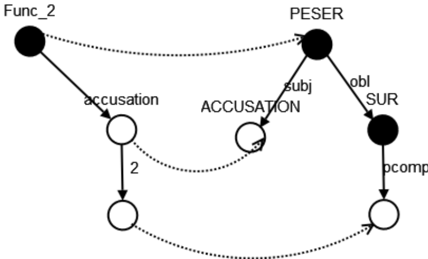


Fig. 2.9 – Présentation graphique de la fonction lexicale $Func_2$ sur la lexie "`accusation`".

TAB. 2.5: Application des fonctions lexicales pour modéliser des collocations en allemand, russe, hongrois, arabe et chinois

	Allemand	Russe	Hongrois	Arabe	Chinois
Pluie = [1]	*Magn* (“Regen” ’[1]’) = “starker” ‹fort›, “Platz”- ‹éclatant›	*Magn* (“dožd” ’[1]’) = “sil’nyj” ‹fort›, “prolivnoj” ‹d’averse›	*Magn* (“eső” ’[1]’) = “zuhogó” ‹torrentiel›	*Magn* (“matar” ’[1]’) = “γazīr” ‹abondant›, “qawijj” ‹fort›	*Magn* (“ŭ” ’[1]’) = “dà” ‹grand›
Argument = [2]	*Magn* (“Argument” ’[2]’) = “gewicthiges” ‹de poids›, “schlagendes” ‹frappant›, “unschlagbares” ‹imbattable›, “unwiderlegbares” ‹irrefutable›	*Magn* (“dovod” ’[2]’) = “veskij” ‹de poids›, “ubeditel’nyj” ‹convaincant›	*Magn* (“érv” ’[2]’) = “komoly” ‹sérieux›	*Magn* (“ḥuǯǯa” ’[2]’) = “dāmiγa” ‹frappant›, “qawijja” ‹fort›	*Magn* (“lùnjŭ” ’[2]’) = “yŏulí-de” ‹de force›
Applaudissements = [3]	*Magn* (“Applaus” ’[3]’) = “tosender” ‹mugissant›	*Magn* (“aplodismenty” ’[3]’) = “burnye” ‹tempétueux›, “gromovye” ‹de tonnerre›	*Magn* (“taps” ’[3]’) = “viharos” ‹tourbillonnant›, “vas”- ‹de fer›	*Magn* (“taṣfīq” ’[3]’) = “ḥarr” ‹chaud›	*Magn* (“zhăngshēng” ’[3]’) = “léidòng” ‹de tonnerre›
Voyage = [4]	$Oper_1$ (“Reise” ’[4]’) = [ART ~$_{acc}$] “machen” ‹faire›	$Oper_1$ (“putešestvie” ’[4]’) = “soveršit” ‹accomplir› [~e]	$Oper_1$ (“utazàs” ’[4]’) = [~t] “tenni” ‹faire›	$Oper_1$ (“safar” ’[4]’) = “qama” [bi ~] ‹se lever, partir en›	$Oper_1$ (“lǚtú” ’[4]’) = “tàshang” [~] ‹marcher sur›
Accord = [5]	$Oper_1$ (“Übereinkunft” ’[5]’) = [“über” ART ~$_{acc}$] “erzielen” ‹obtenir›	$Oper_1$ (“soglašenie” ’[5]’) = “pridti” [“k” ~“ju”] ‹venir à›	$Oper_1$ (“megegyezés” ’[5]’) = [~re] “jutni” ‹arriver à›	$Oper_1$ (“ʔittifāq” ’[5]’) = “tawaṣṣala” [“ʔila” ~] ‹arriver à, obtenir›	$Oper_1$ (“xiéyí” ’[5]’) = “dáchéng” [~] ‹arriver à›
Résistance = [6]	$Oper_1$ (“Widerstand” ’[6]’) = [N_{dat} ART ~$_{acc}$] “leisten” ‹livrer›	$Oper_1$ (“sprotivlenie” ’[6]’) = “okazat’” [un verbe vide] ‹manifester› [N_{dat} ~e]	$Oper_1$ (“ellenàllàs” ’[6]’) = [~t] “kifejteni” ‹développer›, “tanusítani” ‹démontrer›	$Oper_1$ (“muqāwamat” ’[6]’) = “qama” [“bi” ~] ‹se lever, partir en›	
Pardon = [7]	$Oper_1$ (“Entschuldigung” ’[7]’) = [N_{acc} um ~$_{acc}$ “bitten”] ‹prier›	$Oper_1$ (“izvinenija” ‹excuses›) = “prinesti” ‹apporter› [N_{dat} (“svoi”)~“ja”]	$Oper_1$ (“boczánat” ’[7]’) = [N_{abl} ~“ot”] “kérni” ‹demander›	$Oper_1$ (“ʔiʕdarāt” ‹excuses›) = “qaddama” [ART ~] ‹avancer [trans.]›	$Oper_1$ (“qiàn” ‹excuses›) = “dào” [“yige” ‹une› ~] ‹dire›

Source : (Mel’čuk, 1997)

2.2.5 Modèles Sens-Textes

On appelle *modèle Sens-Texte* [MST] d'une langue le modèle de cette langue dans le cadre de la TST. Ce modèle est constitué d'une *grammaire* et d'un *lexique*.

Grammaire d'un Modèle Sens-Texte

La grammaire d'un MST est divisée en modules. Chaque module assure la correspondance entre deux niveaux adjacents (il y en a donc 6) :
- Le module sémantique assure la correspondance entre le niveau sémantique *RSem* et le niveau syntaxique profond *RSyntP* ;
- Le module syntaxique profond assure la correspondance entre le niveau syntaxique profond *RSyntP* et le niveau syntaxique de surface *RSyntS* ;
- etc.

Les règles contenues dans ces modules sont des *règles de correspondance* entre les deux niveaux adjacents. Elles ont la forme $A \Leftrightarrow B|C$, où A et B sont des fragments de structure de deux niveaux adjacents, et C un ensemble de conditions. On peut distinguer deux types de règles : les règles *nodales* sont les règles où la portion de A manipulée par la règle est un noeud, et les règles *sagittales* sont les règles où la portion de A est un arc orienté (prédicat sémantique, dépendance syntaxique, ou relation d'ordre).

Nous présentons ici deux règles de deux modules : syntaxique profond et syntaxique de surface.

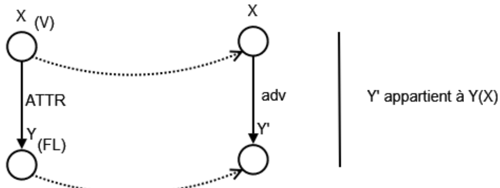


FIG. 2.10 – Règle syntaxique profonde (sagittale) pour le français

Module syntaxique profond La règle 2.10 indique que le dépendant "ATTR d'un verbe X qui est une fonction lexicale Y se réalise au niveau syntaxique de surface par une construction adverbiale, qui est une des valeurs possibles de la fonction lexicale $Y(X)$.

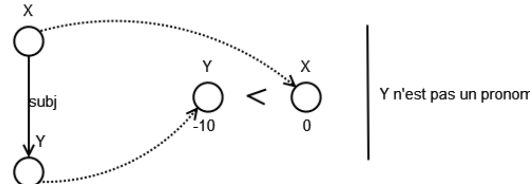


FIG. 2.11 – Règle syntaxique de surface (sagittale) pour le français

Module syntaxique de surface La règle 2.11 indique une linéarisation possible de la fonction syntaxique *subj* : si le sujet du verbe n'est pas un pronom, celui-ci se place avant le verbe. Les positions respectives des deux morphèmes sont indiquées par une marque de position indiquant la *distance* d'un dépendant vis-à-vis de son gouverneur.

Lexique d'un Modèle Sens-Texte

La Théorie Sens-Texte accorde une place fondamentale au Lexique. Celui-ci est définit au sein d'un Dictionnaire Explicatif et Combinatoire [DEC]. Chaque article du DEC définit une lexie, et est divisé en trois zones :
- Une *zone sémantique* donnant la définition lexicographique de la lexie
- Une *zone syntaxique* donnant le tableau de régime (= sous-catégorisation)
- Une zone de *cooccurrence lexicale* donnant, par l'intermédiaire de fonctions lexicales, l'ensemble des collocations formées avec cette lexie, et ses dérivations sémantiques.

Tab. 2.6 – Article de dictionnaire : BLESSUREI.2

Définition lexicographique :
'blessureI.2 de X à Y par Z = 'lésion à la partie Y du corps de X qui est causée par Z et peut causer (1) une ouverture de la peau de Y, (2) un saignement de Y, (3) une douleur de X ou (4) la mort de X.

Régime :

$X = 1$	$Y = 2$	$Z = 3$
1. de N 2. A_{poss}	1. à N	1. à N 2. par N

Contrainte sur 3.1 : N désigne une arme blanche
Contrainte sur 3.2 : N = balle, ...

Exemples :

1 :	"`la blessure de Jean / du soldat / du cheval ; sa blessure` "
2 :	"`une blessure à l'épaule / au coeur / à l'abdomen ; des blessures au corps`"
3 :	"`une blessure à l'arme blanche / au couteau ; une blessure par balle` "
1 + 2 :	"`les blessures de l'enfant aux bras ; sa blessure au poignet droit`"
1 + 2 + 3 :	"`sa blessure par balle à la jambe`"

Fonctions lexicales :

$Syn_{\subset}$:	lésion
$Syn_{\supset}$:	coupure, écorchure, égratinure, morsure, brûlure, écchymose, déchirure, fracture, entorse
$Syn_{\cap}$:	plaie, bobo "fam"
$personne - S_1$:	blessé
$A_{1/2}$:	// blessé
$A_{1/2} + Magn$:	couvert, criblé [de ˜s]
$Magn$:	grave, majeure, sérieuse
$AntiMagn$:	légère, mineure, superficielle // égratignure
$AntiBon$:	mauvaise, vilaine
$IncepMinusBon$:	s'aggraver, s'enflammer, s'envenimer, s'infecter
$Oper_1$:	avoir [ART ˜], porter [ART ˜], souffrir [de ART ˜]
$FinOper_1$:	se remettre, se rétablir [de ART ˜]
$Caus_1Oper_1$:	se faire [ART ˜]
$LiquOper_1$:	guérir [N de ART ˜]
$FinFunc_0$:	se cicatriser, (se) guérir, se refermer
essayer de $LiquFunc_0$:	soigner, traiter [ART ˜], bander, panser [ART ˜]
$CausFunc_1$:	faire [ART ˜] [à N], infliger [ART ˜] [à N] // blesser [N] [avec N= Z]
$Caus_1Func_1$:	se faire [ART ˜], se blesser [avec N=Z]
$Real_1$:	(2) souffrir [de ART ˜] (4) succomber [à ART ˜], mourrir [de ART ˜]
$AntiReal_1$:	(4) réchapper [de ART ˜]
$Fact_0$:	(1) s'ouvrir, se rouvrir, (2) saigner
$Fact_1$:	(4) emporter, tuer [N]
$Able_1Fact_1$ ($\cong Magn$) :	(1) ouverte < profonde < béante, (3) cuisante, douloureuse, (4) fatale, mortelle, qui ne pardonne pas
$AntiAble_1Fact_1$ ($\cong AntiMagn$) :	bénigne "spéc", sans conséquence

Source : (Kahane, 2001; Mel'čuk *et al.*, 1984; Mel'čuk *et al.*, 1988; Mel'čuk *et al.*, 1992)

Chapitre 3

Grammaire d'Unification Sens-Texte

Le chapitre précédent a été consacré à la présentation des grammaires de dépendance et de la théorie Sens-Texte, qui constituent l'inspiration principale des Grammaires d'Unification Sens-Texte, le formalisme grammatical sur lequel nous avons construit notre implémentation. Nous passons à présent à la discussion proprement dite de ce formalisme.

3.1 Un modèle mathématique articulé de la langue

De la plume de son concepteur, Sylvain Kahane[1], GUST se définit comme une « **nouvelle architecture pour la modélisation des langues naturelles**, (...) qui repose sur une synthèse de divers courants en modélisation des langues actuellement très actifs : HPSG (*Head-driven Phrase structure Grammar*), TAG (*Tree Adjoining Grammar*), LFG (*Lexical Functional Grammar*), les grammaires catégorielles, et, avant tout, les grammaires de dépendance et la théorie Sens-Texte. » (Kahane, 2002, p. 1)

GUST est basée, comme son nom l'indique, sur l'*unification*, La structure d'une phrase est obtenue par la combinaison de structures élémentaires associées aux signes linguistiques de la phrase. Deux structures se combinent par unification si elles peuvent être en partie superposées l'une à l'autre ; les deux fragments superposés sont identifiés et le tout donne une nouvelle structure (Kay, 1979).

GUST est un modèle fortement articulé ; il repose, comme nous le verrons, sur une *quadruple articulation du signe*. Tout comme dans la TST, différents niveaux de représentation sont postulés : sémantique, syntaxe, morphotopologie et phonologie. Le passage d'un niveau à un autre se réalise également par le biais de règles d'interface.

Mais le langage sous-jacent de ces règles a été substantiellement simplifié et harmonisé : en effet, toutes les règles GUST peuvent s'exprimer via un même outil mathématique : les **Grammaires d'Unification Polarisées** [dorénavant GUP], formalisme de description linguistique qui permet de manipuler aisément différents types de structures (graphe, arbre, arbre ordonné, chaîne) et de les apparier. Les GUP contrôle la saturation des structures qu'il combine par une *polarisation* de leurs objets.

GUP est un formalisme *générique*, qui permet de simuler la plupart des grammaires basées sur la combinaison de structures (Kahane, 2004). Dans le cadre de ce mémoire, nous ne nous intéresserons évidemment aux GUP que dans l'optique de les utiliser pour offrir une base formelle solide et unifiée aux modèles GUST, et en proposer une implémentation.

Les lecteurs désirant en savoir davantage sur GUST/GUP sont invités à consulter les sources

[1] Sylvain Kahane est Professeur de linguistique à l'Université de Paris 10, et chercheur affilié aux laboratoires Modyco (Modélisation, Dynamique, Corpus) et Lattice (Langues, textes, traitements informatiques, cognition). Il est mathématicien de formation, titulaire d'un doctorat en mathématiques pures de l'Université de Paris 6.

suivantes : (Kahane, 2002, Habilitation à Diriger des Recherches, 82p.) d'abord - tous les fondements linguistiques y sont détaillés -, mais aussi (Kahane, 2001; Kahane, 2004; Kahane et Lareau, 2005).

Dans les sections suivantes, nous détaillerons :
- La **théorie du signe** sous-jacente à GUST (section 3.2) Le *signe* est en effet un concept central de tout modèle linguistique puisqu'il s'agit des “briques” fondamentales, à partir desquelles l'architecture sera fondée ;
- Les **représentations linguistiques** utilisées au sein des niveaux sémantique, syntaxique et morphotopologique (section 3.3) ;
- Le lien mathématique existant entre **trois formalisations** possibles de la notion de grammaire, respectivement appelées transductive, générative et équative (section 3.4) ;
- Les modules d'**interface** entre les différentes représentations : interfaces sémantique-syntaxe $\mathcal{I}_{sem-synt}$, syntaxe-morphotopologie $\mathcal{I}_{synt-morph}$ et morphotopologie-phonologie $\mathcal{I}_{morph-phon}$, ainsi que leur interaction (section 3.5) ;
- La **définition des Grammaires d'Unification Polarisées**, qui permettent de formaliser de façon unifiée l'ensemble des règles contenues dans la grammaire (section 3.6) ;
- Un **exemple concret et détaillé** d'utilisation des GUP en vue d'analyser la représentation sémantique d'une phrase et en dériver sa syntaxe (section 3.7) ;
- Et enfin, une introduction aux **arbres à bulles**, structure mathématique proposée par S. Kahane pour modéliser les phénomènes d'extraction (section 3.9).

3.2 Théorie des signes

3.2.1 Signe saussurien et signe GUST

« Le signe linguistique unit non une chose et un nom, mais un concept et une image acoustique. Cette dernière n'est pas le son matériel, chose purement physique, mais l'empreinte psychique de ce son, la représentation que nous en donne le témoignage de nos sens (...) Le signe linguistique est donc une entité psychique à deux faces. » (de Saussure, 1916, p. 98-99). Ces deux faces sont respectivement appelées *signifié* et *signifiant*. On peut encore dire des signes linguistiques qu'ils sont les entités “irréductibles” de l'analyse compositionnelle d'un énoncé linguistique.

La théorie du signe saussurien repose donc implicitement sur une correspondance directe, “un à un”, entre un sémantème et une chaîne de phonèmes. L'architecture de GUST remet en cause ce postulat classique de la linguistique, en mettant en exergue la nécessité de considérer au moins deux niveaux intermédiaires d'articulation entre le niveau sémantique et le niveau phonologique : le niveau syntaxique et le niveau morphologique :

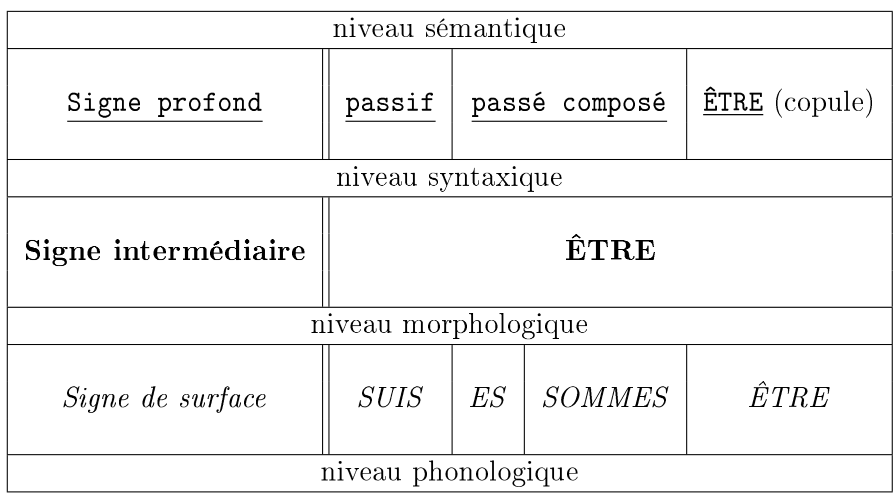
> « En effet, d'une part un lexème peut exprimer une combinaison de sémantèmes et être décomposé en autant d'unités signifiantes (les morphèmes), et d'autre part, un sémantème peut être exprimé par une combinaison de lexèmes, comme c'est le cas pour les locutions, les prépositions régimes, les auxiliaires, etc. Il faut donc pouvoir traiter séparément la correspondance entre sémantèmes et mots-formes, la correspondance entre mots-formes et morphèmes et la correspondance entre morphèmes et phonèmes. En ajoutant la correspondance entre phonèmes et son réels, ceci nous conduit à postuler une quadruple articulation de la langue, que nous prenons pour base de la modélisation. » (Kahane, 2002, p. 74)

GUST envisage donc systématiquement les signes linguistiques comme des **éléments d'interface** entre des niveaux de représentation adjacents. Le signifiant d'un signe “profond” peut donc devenir le signifié d'un signe situé plus en surface, et ainsi de suite jusqu'à arriver à la chaîne phonologique.

3.2.2 Exemple du passif

Nous reprenons ici une discussion provenant de (Kahane, 2002, p. 16-17) sur la modélisation du passif via la théorie des signes retenue pour GUST : [2]

> « Du point de vue du sens, le passif modifie la diathèse d'un verbe et donc la saillance de ses actants par rapport au fait qu'il exprime. Du point de vue de la forme, le passif s'exprime par le verbe **ÊTRE** suivi du **participe passé** du verbe avec promotion de l'objet initial en sujet et rétrogradation du sujet initial en complément d'agent. On voit bien qu'on ne souhaite pas parler de l'expression du passif en termes phonologiques. Du coup, on lui nie généralement le statut de signe (il est révélateur de voir à quel point le passif a pu reçevoir de formalisations différentes, des transformations aux règles lexicales de LFG et G/HPSG).
>
> Or, on peut très bien considérer le `passif` comme un *signe* que nous appellerons *profond* dont le signifiant n'appartiendrait pas au niveau phonologique, mais à un niveau plus profond, que nous appellerons le *niveau syntaxique*.
>
> De même, les éléments qui composent le signifiant du `passif`, **ÊTRE** et la flexion **participe passé**, peuvent être vus comme des *signes* que nous appelerons *intermédiaires*. Ces signes n'ont toujours pas leur signifiant au niveau phonologique, mais à un niveau intermédiaire entre le niveau syntaxique et le niveau phonologique, que nous appellerons le *niveau morphologique*.
>
> Le signe intermédiaire **ÊTRE** donne lieu, selon les signes intermédiaires grammaticaux avec lesquels il se combine, à une série de *signes de surface* qui composent les formes de ce verbe : *SUIS*, *ES*, *EST*, *SOMMES*, etc. Alors que le signe intermédiaire **ÊTRE** désigne la lexie, le signe de surface *ÊTRE* désigne la forme infinitive de cette lexie. Ce dernier signe a pour signifié le signifiant de **ÊTRE** ⊕ **infinitif** et pour signifié la chaîne phonologique /ɛtrə/. »



niveau sémantique				
`Signe profond`	`passif`	`passé composé`		`ÊTRE` (copule)
niveau syntaxique				
Signe intermédiaire	**ÊTRE**			
niveau morphologique				
Signe de surface	*SUIS*	*ES*	*SOMMES*	*ÊTRE*
niveau phonologique				

TAB. 3.1 – Structuration des signes linguistiques concernant le verbe "`être`"

[2] Concernant la notation utilisée dans les paragraphes suivants :
- Les signes profonds sont `soulignés` ;
- Les signes intermédiaires sont écrits en **gras** ;
- Les signes de surface sont écrit en *italique* ;
- Parmi ces trois types de signes, , les signes lexicaux sont signalés en MAJUSCULE et les signes grammaticaux en minuscule.

3.2.3 Interaction des signes

La correspondance entre les représentations sémantique et phonologique est assurée par la combinaison des différents signes de différents modules. Ainsi, le mot-forme "`mange`" dans "`Pierre mange deux pommes`" peut être considéré comme la combinaisons de douze signes :

- Trois signes de surface : *MANG*, -∅- (indicatif-présent) et -*E* (1re personne du pluriel), ayant pour signifiants /mɑ̃ʒ/, /∅/ et /ə/ ;
- Cinq signes intermédiaires : le lexe **MANGER**, les grammes intermédiaires de l'**indicatif** et du **présent** et les grammes d'accord de la **1re personne** et du **pluriel** ;
- Quatre signes profonds : la lexie `MANGER` et les grammies de l'`indicatif`, du `présent` et de l'`actif`.

En résumé, la description d'une phrase dans GUST est obtenue par la *combinaison* de signes lexicaux et grammaticaux de différents modules (profonds, intermédiaires et de surface)

Ces signes représenteront nos règles d'interface entre niveaux adjacents : l'ensemble des *signes profonds* constituent l'*interface sémantique-syntaxe*, l'ensemble des *signes intermédiaires* constitue l'*interface syntaxe-morphotopologie*, et l'ensemble des *signes de surface*, l'*interface morphotopologie-phonologie*.

L'existence d'une quatrième interface, composée de *signes phoniques* et réalisant l'interface entre le niveau phonologique et le signal acoustique, est également envisageable, mais nous n'en parlerons pas dans le cadre de ce travail.

Notons enfin qu'il n'y a plus à proprement parler de distinction entre la grammaire et le lexique : tous les signes, lexicaux ou grammaticaux, sont exprimés au sein d'un formalisme unique.

3.3 Représentations linguistiques de GUST

Cette section est consacrée à présenter les représentations linguistiques utilisées à chaque "étage" de l'architecture GUST. Ces représentations étant globalement similaires à celles déjà présentées dans la section 2.2.3 à propos de la TST, nous serons donc brefs. Le tableau 3.2 présente la catégorisation des signes linguistiques dans GUST et de leurs signifiés et signifiants respectifs. Cette taxinomie n'a pas une importance capitale pour la bonne compréhension du modèle et est, pour dire vrai, plutôt alambiquée, mais nous signalons néammoins son existence.

Nous n'abordons pas ici la représentation phonologique, étant donné qu'elle n'a pour le moment pas été réellement étudiée au sein du GUST. Signalons simplement qu'elle se présente sous la forme d'une chaîne linéraire de phonèmes.

3.3.1 Représentation sémantique

L'objet de ce niveau est de représenter le *sens linguistique* d'un énoncé, ou en d'autres termes l'organisation des signifiés des signes profonds de l'énoncé. Ces signifiés sont appelés *sémantèmes*. Les sémantèmes sont reliés entre eux par des relations de type *prédicat-arguments*, qui sont représentés par des arcs orientés au sein d'un graphe.

Signalons, comme nous l'avons déjà fait à la section 2.2.3, que la notion de "sens" utilisée ici dénote le sens *linguistique*, sans chercher pas à référer à l'état du monde comme le font la plupart des théories sémantiques issues des la logique frégéenne.

Dans le cadre de ce travail, la représentation sémantique se résume à graphe prédicatif, sans envisager de modéliser la structure communicative, la hiérarchie logique (nous en dirons néammoins quelques mots à la section 4.1), les aspects liés à la rhétorique ou à la référence.

Un exemple de représentation sémantique GUST est illustré à la figure 3.6, p. 49.

sémantème				
⇕	sémante (lexie, grammie)	=	signe profond	⇕
lexème, grammène, syntaxème				
⇕	lexe, gramme syntagme	=	signe intermédiaire	⇕
morphème				
⇕	morphe	=	signe de surface	⇕
phonème				
⇕	phone	=	signe phonique	⇕
son				

Tableau repris de (Kahane, 2002)

TAB. 3.2 – Catégorisation des signes linguistiques et de leurs signifiés/signifiants respectifs

3.3.2 Représentation syntaxique

La représentation syntaxique est, comme de bien entendu, un arbre de dépendance. Ses noeuds sont étiquetés par des *lexèmes* éventuellement accompagnés d'une suite de *grammènes*. Les lexèmes et les grammènes sont ici définis comme des unités "monoplanes" à l'interface des signes profonds et intermédiaires (signifiants de signes profonds et signifiés de signes intermédiaires).

Les dépendances syntaxiques sont étiquetées par des fonctions syntaxiques, appelées *syntaxèmes*, et qui peuvent vus comme des signifiés/signifiants de signes (profonds et intérmédiaires, respectivement). Ainsi, la fonction *sujet* pourra être vue comme le signifiant d'un signe profond (par exemple, l'actant d'un verbe à la voix active qui se traduit par une fonction sujet) et le signifié d'un signe intermédiaire (la relation d'ordre entre le verbe et son sujet).

Un exemple de représentation sémantique GUST est illustré à la figure 3.8, p. 51.

3.3.3 Représentation morphotopologique

Morphologie

La représentation morphologique de la phrase est la suite des représentations morphologiques des mots de la phrase. Dans l'état actuel de GUST, le traitement morphologique est réduit à la portion congrue : par souci de simplicité et d'efficacité, le modèle associe directement les structures morphologiques des mots à leur forme phonologique ou graphique (comme cela se fait, du reste, dans la plupart de systèmes de TALN).

Donnons néammoins un petit exemple d'une structure morphologique (ou plutôt de deux : morphologie profonde et de surface), basée sur la phrase "`J'ai parlé de mon passionnant mémoire avec enthousiasme au promoteur`" :

JE $AVOIR_{ind+present+3+sg}$ $PARLER_{partPasse+masc+sg}$ DE $MON_{masc+sg}$ $PASSIONNANT_{masc+sg}$ $MEMOIRE_{sg}$ $AVEC$ $ENTHOUSIASME_{sg}$ A $LE_{masc+sg}$ $PROMOTEUR_{sg}$

$\{JE\}$ $\{A\text{–}\}.\{IND.PRE\}.\{1SG\}$ $\{PARL\text{–}\}.\{PART.PASSE\}.\{MASC\}.\{SG\}$ $\{DE\}$ $\{MON\}.\{MASC\}.\{SG\}$ $\{PASSIONNANT\}.\{MASC\}.\{SG\}$ $\{MEMOIRE\}.\{SG\}$ $\{AVEC\}$ $\{ENTHOUSIASME\}.\{SG\}$ $\{A\}$ $\{LE\}.\{MASC\}.\{SG\}$ $\{PROMOTEUR\}.\{SG\}$

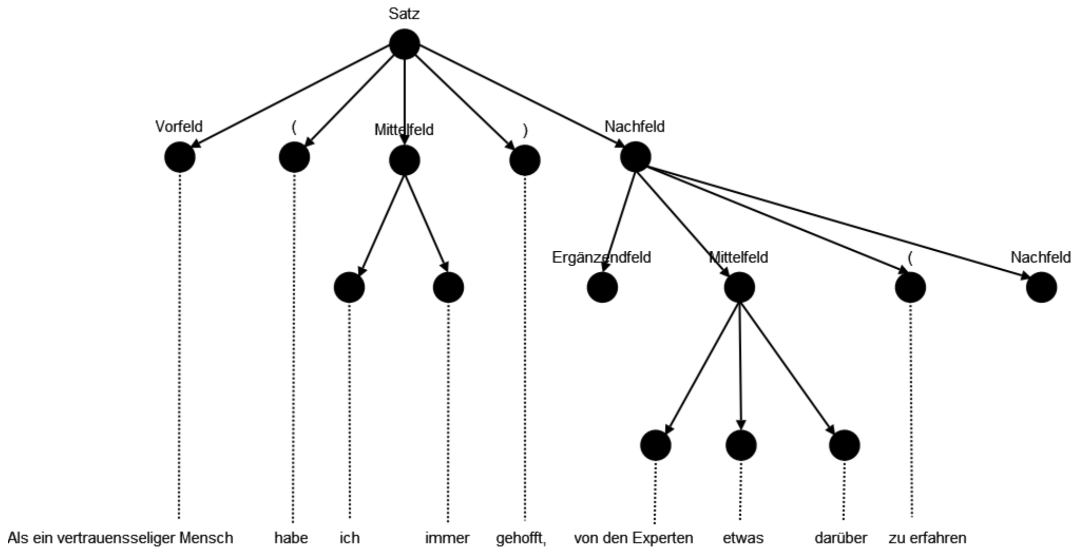
Topologie

Quant à la structure topologique, nous adoptons ici la représentation présentée dans (Kahane et Lareau, 2005), qui formalise la structure topologique à l'aide d'une **grammaire de réécriture hors-contexte**[3]. Nous ne détaillons pas ici les principes sous-jacents à cette modélisation, centrée en français sur la notion d'*amas verbal*, et vous renvoyons à (Gerdes et Kahane, 2001; Gerdes et Kahane, 2004; Gerdes et Kahane, 2006) pour une formalisation détaillée.

Précisons néammoins que les constituants déterminés par cette grammaire ne sont pas des constituants *syntaxiques* comme dans l'approche chomskyienne (=projections syntaxiques maximales des têtes lexicales), mais des constituants *topologiques* (=groupes linéairement ordonnées apparaissant dans la configuration de l'ordre des mots de la phrase, motivés par la structure syntaxique **et** par la structure communicative[4]).

Pour prendre l'exemple de l'allemand, le modèle topologique, inspiré des théories topologiques classiques comme (Drach, 1937; Bech, 1955), est ainsi basé sur une hiérarchie de domaines divisé en cinq champs principaux, appelés *Vorfeld*, *parenthèse gauche*, *Mittelfeld*, *parenthèse droite* et *Nachfeld*, ce qui donne un genre d'analyse présenté à la figure 3.1.

La topologie ne constituant pas le sujet central de ce mémoire, nous nous arrêtons là. Notons pour terminer que la structure topologique (et l'interface entre cette dernière et la structure syntaxique) peuvent toutes deux être facilement formalisées via les GUP, puisque celles-ci possèdent la capacité générative faible et peuvent simuler toute grammaire de réécriture hors-contexte - voir (Kahane, 2004; Kahane et Lareau, 2005).



⟨Etant une personne crédule, j'ai toujours espéré apprendre des experts quelque chose sur ce sujet⟩ (Paul Flora, "Was ist schön ?")

FIG. 3.1 – Exemple d'analyse topologique d'une phrase allemande

[3]Une *grammaire de réécriture hors-contexte* est, pour rappel, un quaduplet (V, Σ, R, S) tel que
- V est un alphabet (ensemble fini de symboles) ;
- $\Sigma \subseteq V$ est l'ensemble de symboles terminaux (symboles faisant partie de l'alphabet sur lequel le langage généré est défini). Bien sûr, $V - \Sigma$ est alors l'ensemble des symboles non terminaux ;
- $R \subseteq V^+ \times V^*$ est l'ensemble des règles de production. Dans le cas des grammaires hors-contexte, on exige de plus que ces règles soient toutes de la forme $A \rightarrow \beta$, avec A symbole non terminal ;
- $S \in V - \Sigma$ est le symbole de départ .

[4]En russe par exemple, c'est la structure communicative qui "guide" l'ordre linéaire des mots, et la syntaxe n'y a qu'un rôle mineur.

3.4 Grammaires transductives, génératives et équatives

Cette section, qui résume une discussion issue de (Kahane, 2002, p. 27-33), est consacrée à une question d'ordre mathématique sur la notion de *grammaire*, question qui nous éclairera sur le lien formel existant entre des grammaires génératives, des grammaires basées sur des contraintes (angl. "*constraint-based grammars*"), et des grammaires de correspondance comme la TST.

Soit deux ensembles S et S' de structures (graphes, arbres, séquences linéaire, etc.). Nous appelerons *grammaire transductive* une grammaire $\mathcal{G}$ qui met en correspondance des éléments de S et de S', par un ensemble fini de règles définissant chacune une *supercorrespondance*, c'est-à-dire un triplet $(s, s', \phi_{(s,s')})$ où s et s' sont des éléments de S et S' mis en correspondance et $\phi_{(s,s')}$ est la fonction associant les partitions de s et de s' définies par la mise en correspondance de s et s'. Une telle supercorrespondance peut être vue comme un cas particulier de *structure produit*, c'est-à-dire une structure obtenue par l'enchevêtrement des structures s et s'.

Une grammaire transductive peut être simulée par une grammaire *générative* qui génère l'ensemble des triplets $(s, s', \phi_{(s,s')})$ décrit par $\mathcal{G}$. Les règles de correspondance sont alors vues comme des règles générant des fragments de structure produit. Elle peut également être simulée par une grammaire *équative* (utilisée par les grammaires basées sur les contraintes) vérifiant si chaque couple de structure s et s' donné en entrée se correspond bien. La grammaire est alors utilisée comme un filtre ou un ensemble de contraintes à satisfaire.

Pour résumer, une même grammaire peut être écrite de 3 manières différentes :

1. Présentation **transductive** : on se donne une structure de l'un des deux ensenbles, et l'on montre comment la grammaire permet de lui associer une structure correspondante ;
2. présentation **générative** : on ne se donne rien au départ, et l'on génère simultanément deux structures en correspondance ;
3. Présentation **équative** : on se donne deux structures, une de chaque ensemble, et l'on utilise les règles de la grammaire pour vérifier si ces deux structures se correspondent.

Cette explication nous permet d'éclairer un point important de notre travail, et ce sous deux aspects. Tout d'abord, la possibilité de définir une équivalence mathématique entre grammaires transductives et grammaires génératives (dont les grammaires d'unification font partie) est évidemment essentielle en ce qui concerne GUST ; car elle permet de traduire les règles grammaticales de la TST en règles d'unification.

Ensuite, l'implémentation de GUST que nous proposons dans ce travail est basée, comme nous le verrons au chapitre 5, sur la programmation par contraintes. Ce qui signifie donc qu'elle fonctionne en fait selon le modèle "équatif" ! La possibilité de compiler une grammaire transductive en une grammaire équative se trouve donc ici mathématiquement fondée.

Terminons cette section par la mention d'une propriété importante de GUST : l'**associativité** : étant donné trois règles de correspondance A, B et C, nous avons $(A \oplus B) \oplus C = A \oplus (B \oplus C)$. Cette propriété assure que la grammaire est fondamentalement *déclarative*[5] (par opposition à "procédurale"), et que toutes les stratégies imaginables de dérivation sont compatibles avec la grammaire. Bien plus, cela signifie que la grammaire est (théoriquement) complètement réver-

[5] Cette assertion doit probablement être nuancée. En effet, GUST reste, suivant la terminologie introduite par (Pullum et Scholz, 2001), un formalisme *génératif-énumératif*, c'est-à-dire qu'il adopte une perspective "proof-theoretic" sur la grammaire : on dérive une analyse en combinant un ensemble d'unités par l'application de règles de production. Une expression E sera alors considérée comme grammaticale selon G si E est *dérivable* dans G.

Une autre perspective possible est la perspective "model-theoretic", où une grammaire est définie comme une *description* logique de modèles bien formés, et où une expression E est considérée grammaticale selon G ssi E est un *modèle* de G - la notion de modèle étant ici prise dans son sens mathématique : m est un modèle d'une formule α ssi la formule α est vrai dans le modèle m.

La perspective "model-theoretic" est clairement plus déclarative que son pendant "proof-theoretic", car elle se détache complètement des mécanismes procéduraux pour ne se centrer que sur la description linguistique.

sible[6] et peut être utilisée directement dans le sens de la synthèse comme de l'analyse.

3.5 Interfaces de GUST

Nous discuterons dans cette section des modules d'interface de GUST, c'est-à-dire de trois interfaces particulières : $\mathcal{I}_{sem-synt}$, $\mathcal{I}_{synt-morph}$ et $\mathcal{I}_{synt-morph}$, et des stratégies d'interaction entre celles-ci.

L'architecture n'est pas très éloignée de l'architecture parallèle défendue par (Jackendoff, 2002)[7], à une différence notable : les interfaces de GUST ne fonctionnent que pour des niveaux adjacents, alors que celles de Jackendoff relient en parallèle tous les niveaux.

Mentionnons également que ces interfaces sont *relationnelles*, c'est-à-dire que la correspondance entre deux niveaux adjacents est définie comme une relation "m-to-n", où une même structure d'entrée peut avoir plusieurs réalisations possibles en sortie, et vice-versa. Ceci contraste avec l'approche *fonctionnelle* - souvent utilisée en sémantique formelle classique, cfr. (Montague, 1974) - où la représentation de sortie est calculée déterministiquement à partir des entrées.

3.5.1 Hiérarchisation et lexicalisation

Nous l'avons dit, l'architecture de GUST est composée de trois modules d'interfaces. Contrairement à la TST, GUST ne sépare les opérations de hiérarchisation et de lexicalisation, et ne distingue donc pas de niveau "profond" et "de surface". Ces deux opérations sont réalisées en parallèle, ce qui a l'avantage de simplifier le modèle.

Néammoins, comme l'expliquent (Kahane, 2002, p. 38-39) et (Kahane, 2003a), la structure profonde est en réalité sous-jacente à la *structure de dérivation*[8] de la représentation d'entrée. La supression de ce niveau ne signifie donc aucunement une "perte" de généralité du modèle.

3.5.2 Interface sémantique-syntaxe $\mathcal{I}_{sem-synt}$

$\mathcal{I}_{sem-synt}$ est constituée d'un ensemble de signes profonds (=règles d'interface sémantique-syntaxe) assurant la correspondance entre, d'une part, des unités *sémantiques* (les sémantèmes), et d'autre part des unités *syntaxiques* (lexèmes, grammènes, syntaxèmes).

La notation utilisée est tirée de (Kahane et Lareau, 2005). Notons au passage la similitude entre celle-ci et les travaux de (Bohnet *et al.*, 2000), qui ont également cherché à formaliser la TST à l'aide de règles de combinaison de graphes.

Donnons un premier exemple concret de signe profond : la figure 3.2 présente la diathèse[9] du verbe "`parler`". Les notations graphiques utilisées dans cette figure seront détaillées à la section 3.7.3, page 50, mais nous pouvons d'ores et déjà en saisir l'intuition générale.

Nous observons que le sémantème ‹parler› exige trois arguments : ‹X parle de Y à Z›. Ceux-ci sont réalisés au niveau syntaxique par une fonction sujet et deux compléments indirects régis par les prépositions "`de`" et "`à`".

Analysons à présent une règle grammaticale, illustrée à la figure 3.3. Elle exprime l'idée que

[6]Notons que notre implémentation GUST n'est entièrement opérationnelle que dans le sens de la synthèse, mais cette restriction n'est pas due à des problèmes fondamentaux, mais à des limitations d'ordre pratique.

[7]En quelques mots, l'architecture parallèle de Jackendoff postule l'existence de différents modules autonomes (représentant les différents niveaux linguistiques) contraints par des règles de bonne formation et reliés par des interfaces relationnelles. Elle s'oppose donc nettement avec les architectures syntactico-centriques utilisées par la plupart des formalismes.

[8]La structure de dérivation, notion bien connue en TALN (cfr. les structures TAG), est en quelque sorte le "témoin" de la dérivation : elle décrit comment des règles se sont combinées pour dériver une structure.

[9]Une diathèse se définit comme la correspondance entre arguments sémantiques et actant syntaxiques.

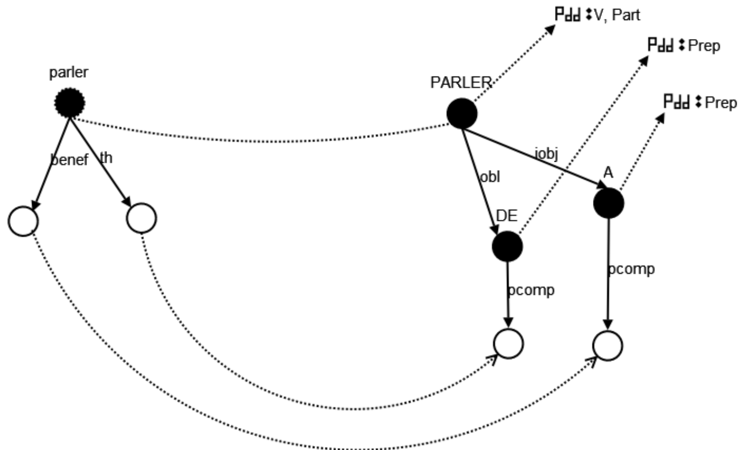


FIG. 3.2 – Diathèse du verbe "parler"

le sens ‹période : passé› attaché à un procès (qui dénote en général le fait que l'action se déroule antérieurement à la situation d'énonciation) peut se traduire en français par l'utilisation de l'imparfait. Remarquons au passage que la règle exige, pour être appliquable, que le mode du verbe soit l'indicatif et que sa voix soit active.

Naturellement, une même entité peut être traduire de plusieurs manières, ainsi ‹période : passé› peut également s'exprimer par le passé composé, comme l'indique la figure 3.4.

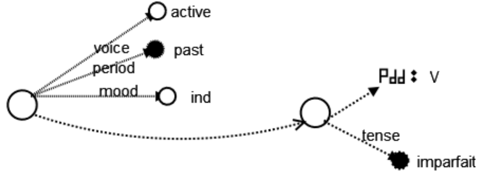


FIG. 3.3 – Règle grammaticale exprimant la réalisation de l'imparfait

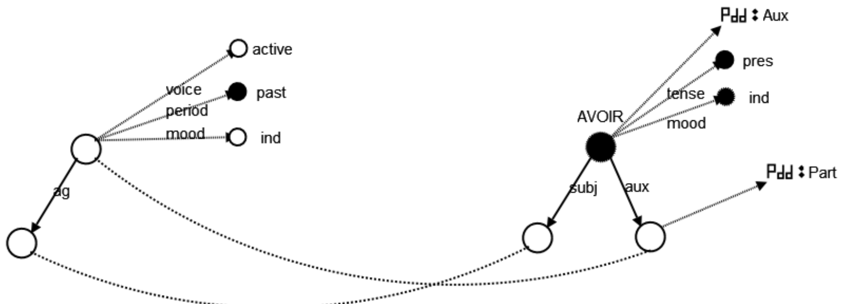


FIG. 3.4 – Règle grammaticale exprimant la réalisation du passé composé

Nous clôturons ici cette première approche de l'interface $\mathcal{I}_{sem-synt}$ de GUST. La section 3.7.3 donne un exemple complet d'un fragment de grammaire, entièrement écrit via le formalisme des GUP. Le chapitre 4 est consacré à la question des modélisations de différents phénomènes linguistiques à traduire dans cette interface, le chapitre 5 à son axiomatisation via la programmation par contraintes, et le chapitre 6 à son implémentation.

3.5.3 Interface syntaxe-morphotopologie $\mathcal{I}_{synt-morph}$

Dans (Kahane, 2002, p. 43), $\mathcal{I}_{synt-morph}$est décrit comme un module traitant essentiellement de trois questions : l'*ordre des mots*, l'*accord* et le *régime*. Elle est composée d'un ensemble de signes intermédiaires (lexes, grammes et syntagmes).

Dans (Kahane et Lareau, 2005), les règles concernant l'accord et de régime ont été déplacées et ne font plus partie de $\mathcal{I}_{synt-morph}$, mais de la grammaire de bonne formation de $\mathcal{G}_{synt}$.

En l'absence d'un réel traitement de la morphologie, $\mathcal{I}_{synt-morph}$ se résume donc à traiter de l'ordre des mots.

Comme nous l'avons indiqué à la section 3.3.3, la structure topologique est contrôlée par une grammaire de réécriture hors-contexte (ou, plus précisément, une GUP simulant une telle grammaire). L'interface $\mathcal{I}_{synt-morph}$ contient donc des règles permettant de passer d'une structure syntaxique, représentée par un arbre de dépendance, à une structure topologique représentée par un arbre de type syntagmatique.

Par manque de place, nous ne détaillerons pas plus avant le contenu de $\mathcal{I}_{synt-morph}$, le lecteur intéressé pourra se référer à (Kahane, 2004; Kahane et Lareau, 2005).

3.5.4 Interface morphotopologie-phonologie $\mathcal{I}_{morph-phon}$

GUST ne traite pas à l'heure actuelle les questions morphologiques et phonologiques, cette interface assigne donc pour l'instant directement une représentation phonique ou graphique à une suite de morphèmes.

3.5.5 Stratégies d'interaction

Terminons cette section par une discussion sur les différentes *stratégies* d'interaction entre les interfaces, pour des tâches d'analyse ou de synthèse. En effet, l'ordre dans lequel les signes vont être déclenchés déterminera une stratégie particulière.

Ainsi, la stratégie **horizontale** consiste à déclencher les signes étage après étage. Pour une tâche d'analyse, l'on commencera donc par activer les différents signes de surface (= lemmatisation), ensuite les signes intermédiaires ("analyse morphosyntaxique"), et enfin les signes profonds ("analyse sémantique").

A l'opposé, la stratégie **verticale** déclenche, pour chaque forme de surface (un "mot"), l'ensemble des signes des différents étages qui y correspondent, sans attendre une analyse complète de l'énoncé à un niveau donné. Cette stratégie, si elle est peu utilisée en TALN, est néammoins beaucoup plus proche du fonctionnement cognitif du traitement linguistique par un locuteur humain.

Ces deux stratégies sont évidemment les extrémités d'un continuum à l'intérieur duquel bien d'autres stratégies sont possibles. La synthèse **incrémentale**, par exemple, cherche à produire l'arbre syntaxique de haut en bas et les mots-formes dans l'ordre linéaire.

Dans notre implémentation de GUST, ce choix entre les différentes stratégies pourra être simulé en déterminant un ensemble de *priorités* associées aux contraintes, et qui déterminent l'ordre dans lequel les différentes contraintes linguistiques seront appliquées.

Terminons en mentionnant que GUST permet, de par son architecture, d'écrire à la fois des grammaires complètement **lexicalisées** (i.e. où toute l'information est attachée aux unités lexicales, comme en TAG) ou des grammaires fortement **articulées**, comme les métagrammaires (Duchier *et al.*, 2004). De plus, il est possible de passer d'une grammaire articulée à une grammaire lexicalisée en *précompilant* la grammaire, et ainsi améliorer les performances du sytème.

3.6 Grammaire d'Unification Polarisées

3.6.1 Généralités

Les GUP sont, avant tout, des grammaires d'*unification*, c'est-à-dire des grammaires fonctionnant par la *superposition* de structures. Il est par là possible de construire des unités complexes (typiquement, une phrase) en combinant des unités élémentaires. Si le premier formalisme basé sur l'unification provient de (Kay, 1979), l'intuition de base se retrouve déjà dans (Jespersen, 1924; Tesnière, 1934; Ajdukiewicz, 1935). La phrase peut à cet égard être comparée - (Tesnière, 1959, p. 16) le fait d'ailleurs explicitement - à une molécule chimique, composée de différents "atomes" représentant les mots-formes. Chacun de ces "atomes" possède une valence (c'est le terme même utilisé en linguistique, par analogie à la chimie) qui lui permet de spécifier ses possibilités de combinatoire avec d'autres mots-formes.

Néammoins, la plupart des grammaires d'unification actuelles ne permettent pas d'indiquer de manière explicite la nature *saturée* ou *insaturée* de certaines unités, et ce faisant la stabilité de la structure globale. Ainsi, un adverbe n'a en général de sens que relié à un verbe ou un adjectif[10] , un nom commun exige un déterminant, et un verbe requiert la présence de certains actants. Dans la plupart des formalismes, le respect de ce type de propriété est assuré via des règles distinctes de l'opération d'unification.

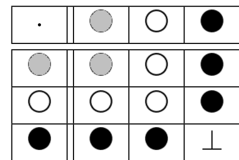
L'avantage des GUP est de pouvoir contrôler *explicitement* la saturation des structures, et ce par l'attribution à chaque objet d'une *polarité*, que nous définissons ci-dessous. Notons que le processus d'analyse ou de synthèse d'une structure équivaut donc à une tentative de saturation de l'ensemble de la structure, opération qui permettra donc de guider la composition des unités élémentaires en unités plus grandes.

Pour terminer cette introduction par un historique de ces grammaires, signalons que la première mention de cette idée de "saturation" de structures linguistiques provient de (Nasr, 1995), que (Duchier et Thater, 1999) est le premier à introduire l'idée de polarité, que (Perrier, 1999) propose un formalisme complet basé sur ce concept de polarisation (dans le cadre des grammaires catégorielles), et enfin que la présentation des GUP que nous adoptons ici provient essentiellement de (Kahane, 2004; Kahane et Lareau, 2005), ainsi que d'une entrevue et de quelques communications personnelles avec son concepteur, Sylvain Kahane.

3.6.2 Système de polarités

> « Les grammaires d'unification polarisées sont des grammaires permettant de générer des ensembles de structures finies. Une structure repose sur des *objets*. Par exemple, pour un graphe (orienté), les objets sont des noeuds et des arcs. Chaque arc est lié à deux noeuds par les fonctions *source* et *cible*. Ce sont ces fonctions qui fournissent la structure proprement dite.
>
> Une *structure polarisée* est une structure dont les objets sont polarisés, c'est-à-dire associés par une fonction à une valeur appartenant à un ensemble fini P de *polarités*. L'ensemble P est muni d'une opération commutative et associative notée "$\cdot$", appelée *produit*. Un sous-ensemble N de P contient les polarités dites *neutres*. Une structure polarisée est dite *neutre* si tous les objets de cette structure sont neutres. Nous utiliserons ici un système de polarités P = {◍,○,●} (que nous appellerons ainsi : ● = noire = saturée, ○ = blanche = contexte obligatoire et ◍ = grise = neutre absolu), avec N = {◍,●}, et un produit défini par le tableau suivant (où $\perp$ représente l'impossibilité de se combiner) :

[10]Bien sûr, dans ce cas particulier, le verbe en question peut être sous-entendu : "`(fais) vite !`"



·	◍	○	●
◍	◍	○	●
○	○	○	●
●	●	●	⊥

TAB. 3.3 – Tableau des polarités

Les structures peuvent être combinées par *unification*. L'unification de deux structures A et B donne une nouvelle structure $A \oplus B$ obtenue en "collant" ensemble ces structures par l'identification d'une partie des objets de A avec une partie de ceux de B. Lorsque A et B sont unifiées, la polarité d'un objet de $A \oplus B$ obtenue par identification d'un objet de A et d'un objet de B est le produit de leurs polarités. Toutes les fonctions associées aux objets unifiés sont nécessairement identifiées (comme le sont les traits quand on unifie deux structures de traits).

Une *grammaire d'unification polarisée* (GUP) est définie par une famille finie T de types d'objets (avec des fonctions associées aux différents types d'objets), un système $(P, \cdot)$ de polarités, un sous-ensemble $N \in P$ de polarités neutres, et un ensemble fini de structures élémentaires polarisées, dont les objets sont décrits par T et dont une peut être marquée comme la structure initiale. Les structures *générées* par la grammaire sont les structures neutres obtenues par combinaison de la structure initiale et/ou d'un nombre fini de structures élémentaires. Rappelons que le formalisme est monotone (avec l'ordre ◍< ○< ●) et que les structures peuvent être combinées dans n'importe quel ordre. » (Kahane et Lareau, 2005, p. 3)

3.6.3 Grammaires de bonne formation *vs.* grammaires de correspondance

La formalisation de GUST via les GUP fait un grand usage de la notion de polarisation pour contrôler la construction des objets, et ce au sein de *deux classes* de "grammaires" :

- Chaque niveau de représentation possède sa *grammaire de bonne formation* assurant, comme son nom l'indique, que la structure d'un niveau donné est "bien formée". Nous verrons aux sections 3.7.1 et 3.7.2 des exemples concrets de ces grammaires ;
- De plus, il existe bien sûr des *grammaires de correspondance* (les "signes" où règles d'interface de GUST) permettant aux structures d'être traduites d'un niveau à l'autre. Pour un exemple concret, voir section 3.7.3.

A chacune de ces grammaires sera associée une polarité, qu'il s'agira bien sûr de neutraliser. GUST/GUP ayant la propriété d'associativité, l'ordre dans lequel cette saturation s'effectuera n'aura pas d'influence sur le résultat final, mais bien sur le chemin suivi pour y arriver :

> « L'ordre dans lequel nous neutraliserons ces différentes polarités va décider d'une procédure particulière en analyse ou en génération. Les procédures en largeur résultent de la neutralisation successive de toutes les polarités propres à un module (neutralisation de toute la structure sémantique, puis syntaxique, etc.) tandis que les procédures en profondeur résultent d'une neutralisation en cascade de toutes les polarités introduites par un objet, c'est-à-dire que dès qu'un objet est construit à un niveau donné, on cherche à neutraliser les polarités des autres niveaux associées à cet objet plutôt que de construire d'autres objets du même niveau. Malgré une architecture stratifiée (séparation des niveaux et des interfaces), notre modèle peut donc tout à fait gérer une interaction complexe entre les différents modules et simuler une analyse ou une génération incrémentale. » (Kahane et Lareau, 2005, p. 2)

3.6.4 Procédures équative, transductive et générative

A la section 3.4, nous avons établi une distinction entre grammaires équatives, transductives et génératives. Voyons à présent comment cette distinction est envisagée sous l'angle des GUP.

Soit deux ensembles A et B appartenant respectivement à des ensembles $\mathfrak{A}$ et $\mathfrak{B}$ quelconques, dont la bonne formation est assurée par les grammaires $\mathcal{G}_{\mathfrak{A}}$ et $\mathcal{G}_{\mathfrak{B}}$, respectivement. Soit également une grammaire $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ mettant en correspondance des structures élémentaires composant les éléments de $\mathfrak{A}$ et $\mathfrak{B}$. $\mathcal{G}_{\mathfrak{A}}$, $\mathcal{G}_{\mathfrak{B}}$ possèdent chacune leur propre polarité propre.

Voyons à présent comment cette correspondance va opérer, selon que $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ est équative, transductive ou générative[11] :

Grammaire équative : Dans cette optique, deux structures sont fournies en entrée et le rôle de $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ est d'assurer que celles-ci sont bien en correspondance. Il est donc nécessaire de partitionner les deux structures en un même nombre de fragments qui se correspondent deux à deux, et d'assigner des polarités $p_{\mathcal{I}}$ blanches aux objets des deux structures, qui devront être neutralisées. Les règles de $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ contiendront quant à elles des objets (des niveaux de représentation de $\mathfrak{A}$ et de $\mathfrak{B}$) de polarité $p_{\mathcal{I}}$ noire mis en correspondance. De plus, tous les objets de A devront être neutralisés par la grammaire de $\mathfrak{A}$, et les objets de B par la grammaire de $\mathfrak{B}$, pour s'assurer de leur bonne formation.

Grammaire transductive : On fournit à $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ une structure A de $\mathfrak{A}$ dont tous les objets portent une polarité $p_{\mathcal{I}}$ blanche. On déclenche alors un jeu de règles afin de neutraliser A, ce qui nous construit une structure B synchronisée avec $\mathfrak{A}$. Une grammaire de bonne formation des structures de $\mathfrak{B}$ doit maintenant vérifier que B appartient bien à $\mathfrak{B}$. La structure B, tout en étant neutre pour $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ (tous les objets portent une polarité $p_{\mathcal{I}}$ noire), doit donc déclencher la grammaire de $\mathfrak{B}$. Elle doit pour cela être entièrement blanche en polarité $p_{\mathcal{B}}$.

Grammaire générative : Ici, aucune entrée n'est fournie à la grammaire, son rôle étant de générer un couple de structures en correspondance. Les couples (A, B) générés par $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ reçoivent une polarité $p_{\mathcal{I}}$ noire, étant donné que ceux-ci sont, *per se*, en correspondance. A et B doivent ensuite être saturés par leurs grammaires respectives $\mathcal{G}_{\mathfrak{A}}$ et $\mathcal{G}_{\mathfrak{B}}$, et la génération est effectuée.

3.6.5 Double polarité

> « En somme, chaque module, qu'il soit une grammaire de bonne formation ou une grammaire d'interface, possède une polarité propre contrôlant la construction des objets par cette grammaire, mais pour assurer l'appel des modules adjacents, les objets des règles de $\mathfrak{A}$ et de $\mathfrak{B}$ reçoivent, en plus de leurs polarités respectives $p_{\mathfrak{A}}$ et $p_{\mathfrak{B}}$ blanche, tandis qu'il faut ajouter aux règles de $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ une polarité blanche $p_{\mathfrak{A}}$ pour les éléments du niveau de représentation de $\mathfrak{A}$ et $p_{\mathfrak{B}}$ pour les éléments du niveau de $\mathfrak{B}$. Un objet construit par $\mathfrak{A}$ aura donc une double polarisation $p_{\mathfrak{A}} - p_{\mathcal{I}}$ (●, ○) tandis que l'élément correspondant construit par $\mathcal{I}_{\mathfrak{A}-\mathfrak{B}}$ aura une double polarisation $p_{\mathfrak{A}} - p_{\mathcal{I}}$ (○, ●). Ceci nous donne un système à quatre polarités {(○, ○), (○, ●), (●, ○), (●, ●)}, équivalent au système (○, −, +, ●) de (Bonfante *et al.*, 2004) et (Kahane, 2004). Ces couples de polarités sont les *polarités d'articulation.* » (Kahane et Lareau, 2005, p. 5)

[11]Les procédures concernant les grammaires équative et transductive sont issues de (Kahane et Lareau, 2005) (et ici légérement adaptées), celle concernant la grammaire générative est originale.

3.7 Exemple détaillé d'utilisation des GUP

Discutons à présent d'un exemple concret qui nous permettra de mieux appréhender le fonctionnement de notre grammaire. Notre exemple est basé sur la phrase suivante : "`J'ai parlé avec enthousiasme de mon passionnant mémoire au promoteur`"[12].

3.7.1 Grammaire sémantique de bonne formation $\mathcal{G}_{sem}$

Nos représentations sémantiques sont basées sur un graphe de relations *prédicat-argument* entre sémantèmes. Les noeuds d'un graphe sémantique représentent les sémantèmes et les arcs représentent des relations prédicat-argument, étiquetées par des *rôles thématiques*[13].

Nous avons également ajouté à notre formalisme un ensemble d'*attributs sémantiques*[14] attachés aux sémantèmes, et qui spécifient des informations supplémentaires telles que le nombre (pour un sémantème associé à un lexème nominal), la période, la voix et le mode (pour les sémantèmes associé à un lexème verbal), etc. Ils sont représentés par des arcs en pointillés.

Notre grammaire $\mathcal{G}_{sem}$ est donc une grammaire de graphe dont les objets portent chacun une polarité, notée p_{sem} dans la suite, qui indique quel est le noeud construit par la règle (polarité noire), et quels sont les noeuds constituant la valence sémantique du prédicat (polarité blanche).

Nous présentons aux figures 3.5 et 3.6 un fragment de $\mathcal{G}_{sem}$ et le graphe à saturer.

La table 3.4 donne quelques indications sur la signification des abbréviations utilisées dans notre grammaire sémantique.

3.7.2 Grammaire syntaxique de bonne formation $\mathcal{G}_{synt}$

Passons à présent à l'arbre syntaxique et à la grammaire qui en assure la bonne formation. L'arbre syntaxique correspondant au graphe sémantique présenté à la section précédente est illustré à la figure 3.8, et le fragment de grammaire qui en assure la bonne formation est présenté à la figure 3.7.

La représentation syntaxique est bien évidemment un arbre de dépendance dont les noeuds sont étiquetés par des représentations canoniques, lemmatisées de mots. Le noeud syntaxique est étiqueté par le nom d'un *lexème* et des *fonctions grammaticales* lient ce noeud à d'autres objets représentant les *grammènes*. Enfin, les arcs de l'arbre sont étiquetés par des *fonctions syntaxiques*. Les objets de $\mathcal{G}_{synt}$ possèdent une polarité p_{synt} indiquant quels sont les objets construits par la règle, et sous quelles conditions la règle peut être appliquée.

[12]Nous avons choisi de travailler sur un exemple non trivial pour montrer toutes les possibilités du formalisme. Voir également (Kahane et Lareau, 2005) pour une présentation du même type, sur un exemple un peu plus simple.

[13]Notons que ces rôles thématiques sont absents du formalisme initialement présenté dans (Kahane, 2004), (Kahane et Lareau, 2005), il s'agit donc d'une ajout de notre part au formalisme, dont la principale motivation est de permettre de distinguer les différents prédicats associés à un même lexème. Ainsi, ‹mémoire› peut être un prédicat unaire dont l'argument spécifie la personne qui l'a rédigée, mais l'argument peut tout aussi bien désigner, par exemple, le sujet de ce mémoire. Ces deux prédicats sont réalisés différemment d'un point de vue syntaxique : le premier donnera une structure de type : "`Mémoire de X`", tandis que le second produira "`Mémoire sur X`". L'utilisation d'étiquettes comme les rôles thématiques permet de distinguer clairement les deux constructions, en utilisant tout simplement deux rôles différents pour ces deux constructions. Nous laissons de côté la question - controversée, voir (Dowty, 1989) - de l'adéquation linguistique de ces rôles thématiques.

[14]Notons que l'enrichissement du graphe par l'utilisation d'attributs - idée qui ne figurait pas non plus dans la présentation initiale du formalisme dans (Kahane et Lareau, 2005) - n'altère en rien l'expressivité de celui-ci ; il eût en effet été parfaitement possible de représenter ces informations par des sémantèmes "normaux", mais il facilite grandement le traitement automatique en limitant le nombre d'unités composant le graphe.

Le formalisme initial, qui ne contenait que des sémantèmes et des relations prédicats-arguments, nous a en effet posé de sérieux problèmes de traitement puisqu'il exigeait de traiter des graphes contenant parfois un grand nombre de noeuds. L'ajout des *attributs sémantiques*, objets polarisés et étiquetés rattachés à un noeud du graphe, nous a ainsi permis de réduire substantiellement la taille des graphes à analyser, et donc la complexité du traitement.

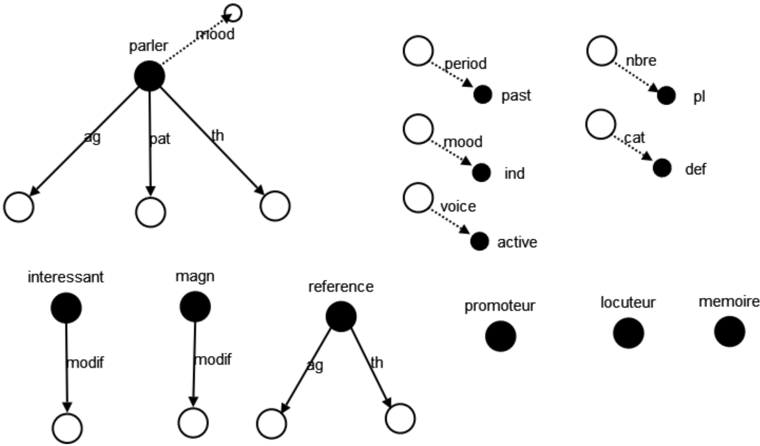


FIG. 3.5 – Extrait de la grammaire sémantique de bonne formation $\mathcal{G}_{sem}$

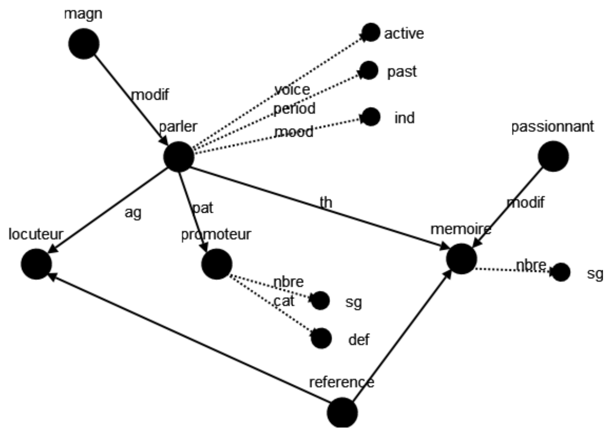


FIG. 3.6 – Graphe sémantique initial

On peut distinguer dans $\mathcal{G}_{synt}$ trois grands types de règles :

1. Les règles lexicales (2 premières lignes dans la figure 3.7) : elles permettent la saturation des lexèmes, indique leur *pdd* et les grammènes qui leur sont nécessaires ;
2. Les règles sagittales (3ᵉ ligne) : elles décrivent les différentes relations syntaxiques possibles ;
3. Les règles lexicales (2 premières lignes dans la figure 3.7) : elles permettent la saturation des lexèmes, indique leur *pdd* et les grammènes qui leur sont nécessaires ;
4. Règles grammaticales (4ᵉ ligne) : règles d'accord, de conjugaison, etc.

Contrairement à la structure sémantique où nous avons opéré quelques modifications sur le formalisme initial, la structure syntaxique que nous avons adoptée est, à quelques détails près, identique à celle présentée dans (Kahane et Lareau, 2005). La seule différence notable porte sur l'utilisation d'une polarité $p_{synt-gouv}$, proposée pour vérifier que la structure est bien un arbre (i.e. chaque noeud est gouverné une et une seule fois, à l'exception du sommet). Nous n'avons pas utilisé cette polarité dans notre implémentation étant donné l'existence dans le système XDG d'une contrainte globale spécialement conçue à cet effet et bien plus efficace que l'utilisation d'une polarité supplémentaire.

La table 3.4 donne quelques indications sur la signification des abbréviations utilisées dans notre grammaire sémantique.

ag	Agent d'un procès
$benef$	Bénéficiaire d'un procès
cat	Catégorie ‹définie› ou ‹indéfinie› d'une entité (qui s'exprimera syntaxiquement par l'utilisation d'un article défini ou indéfini)
$entityType$	Type d'entité (par ex. ‹pays›, ‹nourriture›, ‹être vivant›)
$goal$	But d'un procès
loc	Cadre spatial d'un procès
$manner$	Manière d'un procès
$modif$	Modifieur d'une entité
$mood$	Mode d'un procès (‹indicatif›, ‹infinitif›, ‹impératif›, etc.)
$nbre$	Nombre (‹singulier›, ‹pluriel›, ‹partitif›)
pat	Patient d'un procès
$period$	Période d'un procès, exprimée relativement à la situation d'énonciation (‹présent›, ‹passé›, ‹futur›)
th	Thème d'un procès
$time$	Cadre temporel d'un procès
$voice$	Voix d'un procès (‹actif›, ‹passif›)

Tab. 3.4 – Abréviations utilisées dans la grammaire sémantique

3.7.3 Grammaire de correspondance $\mathcal{I}_{sem-synt}$

L'interface sémantique-syntaxe $\mathcal{I}_{sem-synt}$ est une grammaire de correspondance entre l'ensemble des graphes sémantiques décrit par $\mathcal{G}_{sem}$ et l'ensemble des arbres syntaxiques décrit par $\mathcal{G}_{synt}$.

Les objets des règles de $\mathcal{I}_{sem-synt}$ portent tous une polarité $p_{sem-synt}$ (blanche ou noire, selon qu'ils sont ou non construits par la règle en question). Au niveau des conventions graphiques, nous avons noté les liens de correspondances par des courbes orientées en pointillés. Le reste est identique aux éléments déjà discutés dans les sections précédentes.

Un fragment de grammaire est présenté à la figure 3.9.

3.7.4 Exemple d'opération de synthèse sémantique

Nous illustrons à présent le fonctionnement complet du système pour une opération de synthèse sémantique. Nous procédons en trois étapes successives, qui sont détaillées à la figure 3.10 (nous avons retiré des structures tous les attributs, grammènes et parties du discours, afin de ne pas surcharger l'image de détails) :

1re étape : Nous commencons par un graphe sémantique muni de la double polarisation ($p_{sem-synt}$, p_{sem}). Nous observons que toutes les polarisations p_{sem} sont saturées, ce qui est logique puisque ce graphe respecte les règles de bonne formation générées par la grammaire $\mathcal{G}_{sem}$. Par contre, les polarisations $p_{sem-synt}$ sont évidemment insaturées puisque nous n'avons pas encore procédé à la neutralisation de celles-ci par les règles de correspondance de $\mathcal{I}_{sem-synt}$.

2e étape : L'image présente la structure résultant de la neutralisation par $\mathcal{I}_{sem-synt}$ de toutes les polarités $p_{sem-synt}$. La structure de gauche est donc entièrement saturée. Quant à la structure de droite, ses noeuds possèdent une double polarité (p_{synt}, $p_{sem-synt}$), et l'on observe donc que les polarités $p_{sem-synt}$ sont saturées, mais pas les polarités p_{synt}.

3e étape : Il nous reste donc à saturer les polarités p_{synt} de la structure de droite, en appliquant les règles de $\mathcal{G}_{synt}$. La troisième image présente donc le résultat final, c'est-à-dire une structure entièrement saturée, qui signifie donc que la synthèse s'est terminée avec succès.

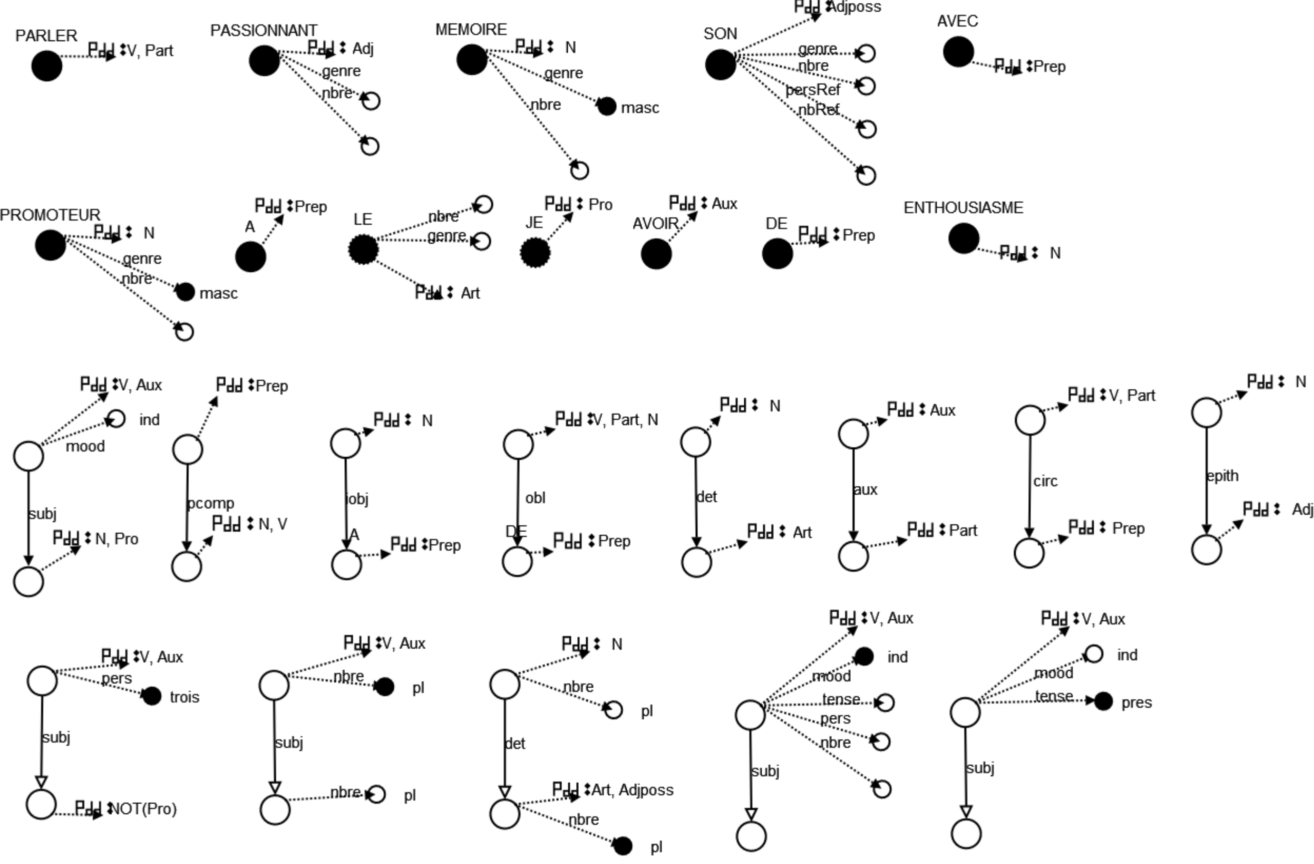


Fig. 3.7 – Extrait de la grammaire syntaxique de bonne formation $\mathcal{G}_{synt}$

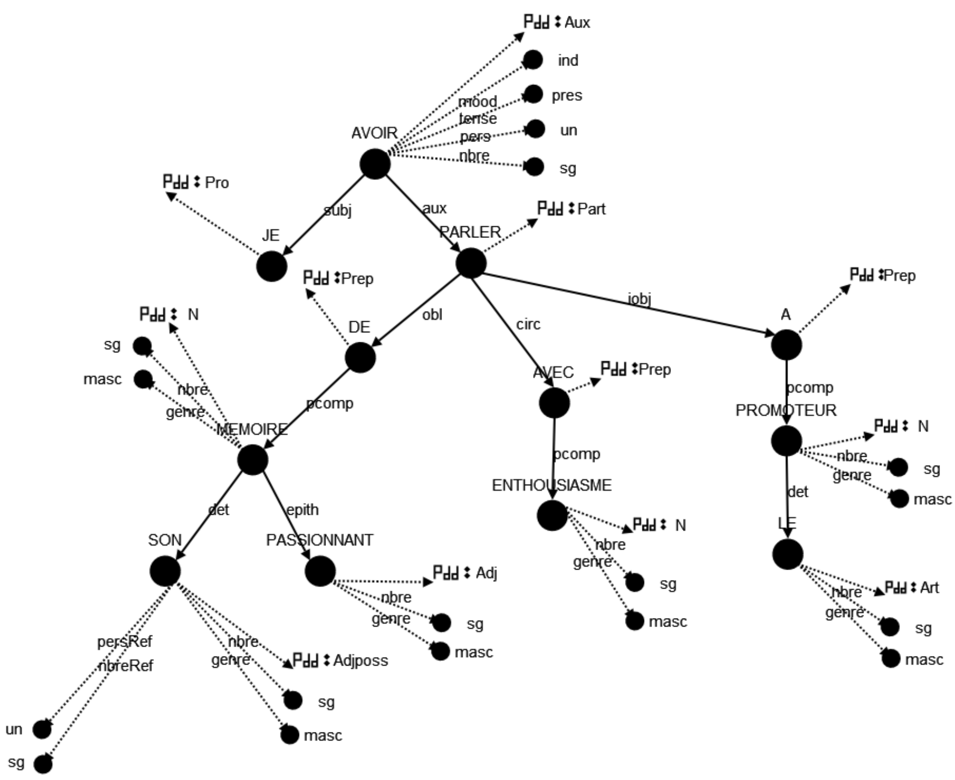


Fig. 3.8 – Arbre syntaxique généré par la grammaire

acirc	Complément circonstanciel de l'adjectif ou de l'adverbe
attr	Attribut du sujet de la copule
Adj	Adjectif
AdjPoss	Adjectif possessif
Adv	Adverbe
Art	Article
Aux	Auxiliaire
aux	Complément verbal de l'auxiliaire
circ	Complément circonstanciel du verbe
det	Déterminant d'un nom
dobj	Complément direct du verbe
epith	Epithète du nom
genre	Genre d'un nom, d'un adjectif, d'un déterminant
inf	complément à l'infinitif d'un verbe
iobj	Complément indirect du verbe
N	Nom
nameType	Type particulier d'un nom (par ex. le fait qu'il accepte un déterminant partitif)
nbre	Nombre d'un nom, d'un verbe, d'un adjectif, d'un déterminant
nbreref	Nombre référentiel d'un adjectif possessif (donne "`mon`"-"`notre`", "`son`"-"`leur`", etc.)
Num	Adjectif Numéral
obl	Objet oblique du verbe
Part	Participe
pcomp	Complément prépositionnel
pdd	Partie du discours
persref	Personne d'un adjectif possessif (qui donne "`mon`"-"`ton`"-"`son`", etc.)
Prep	Préposition
Pro	Pronom
Quantif	Quantifieur (par ex. "`beaucoup de`")
obl	Oblique du verbe
subj	Sujet du verbe
tense	Temps d'un verbe
V	Verbe
verbclass	Classe particulière d'un verbe (par ex. transitif)

TAB. 3.5 – Abbréviations utilisées dans la grammaire syntaxique

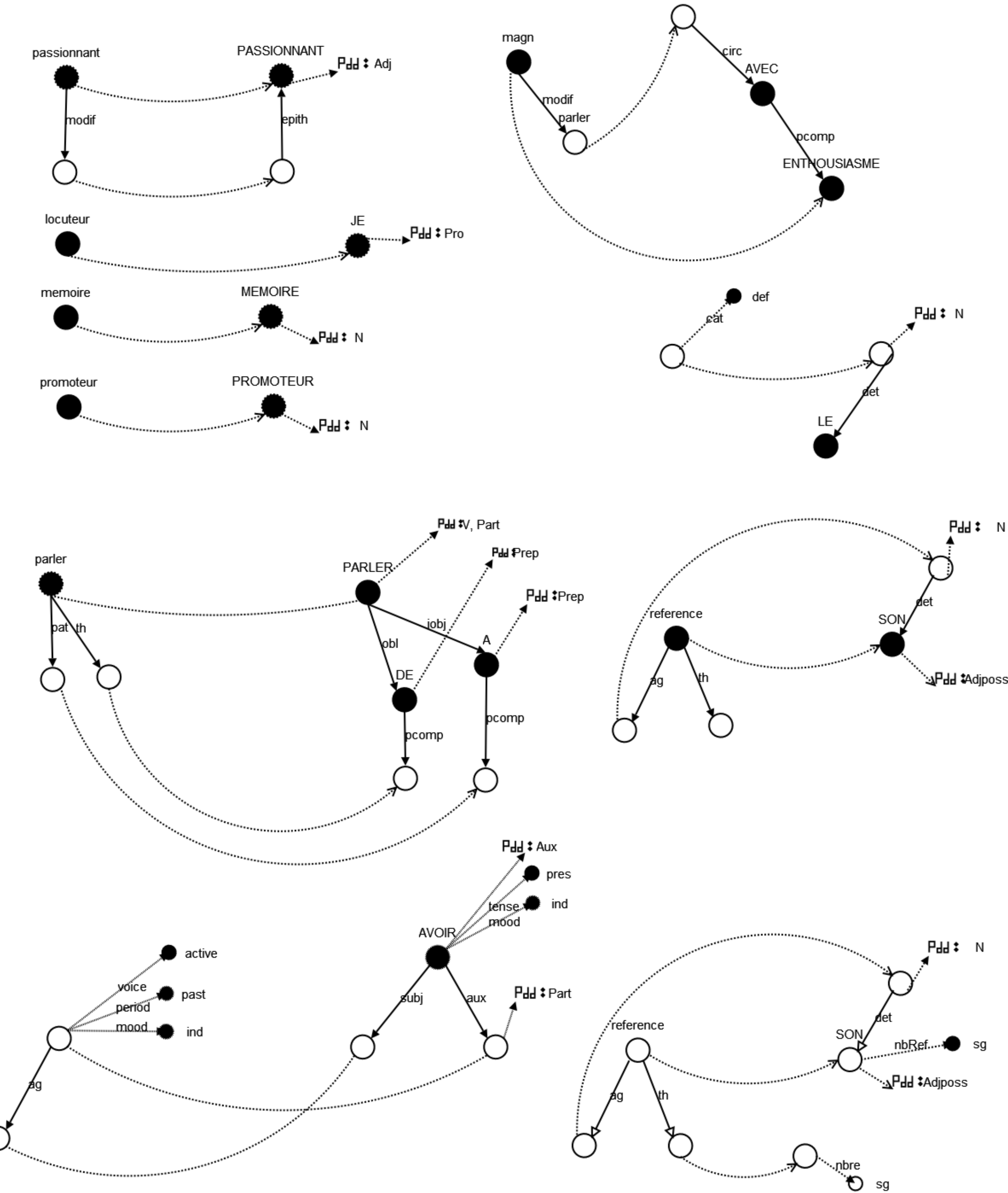
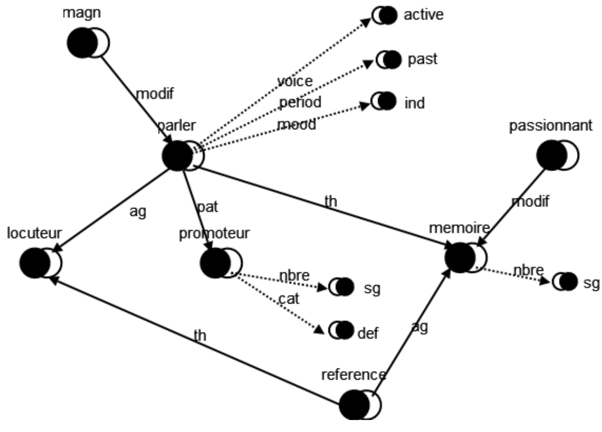
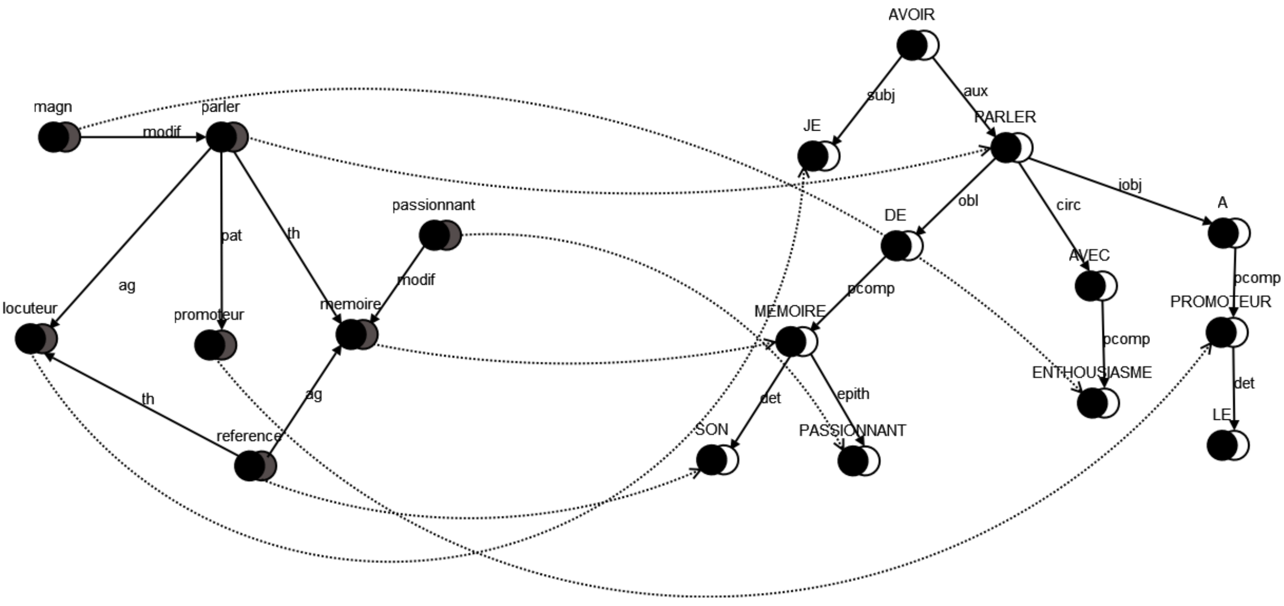


Fig. 3.9 – Extrait de la grammaire de correspondance $\mathcal{I}_{sem-synt}$





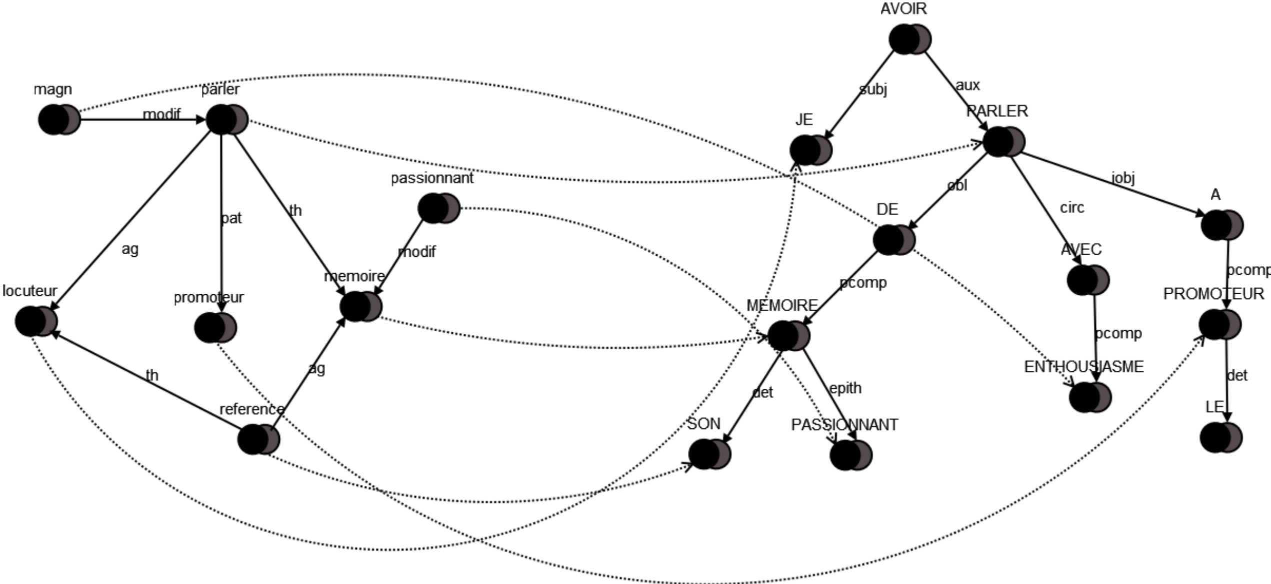


FIG. 3.10 – Exemple d'opération de synthèse sémantique ⇒ syntaxe, en trois étapes

3.8 Possibilité de “va-et-vient” entre niveaux

L'article (Kahane et Lareau, 2005, p. 7) insiste sur la possibilité, via des GUP, d'opérer des *va-et-vient* entre niveaux lorsque la représentation donnée en entrée n'est pas suffisante pour saturer entièrement la structure de sortie. Expliquons par un exemple ce que ceci signifie :

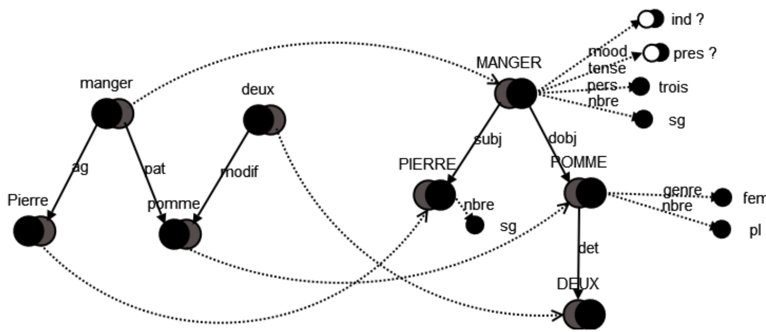


FIG. 3.11 – Exemple de structure insaturée

Dans la figure 3.11, l'on remarque que les grammènes de mode (“`mood`”) et de temps (“`tense`”) sont insaturés, car rien dans la structure d'entrée ne permet de fixer leur valeur. (Kahane et Lareau, 2005) explique qu'il est possible, de par la monotonicité du formalisme GUP, de revenir à la structure d'entrée et modifier celle-ci “on the fly” pour lui ajouter les informations manquantes.

De par l'utilisation de la programmation par contraintes, notre implémentation peut simuler ce va-et-vient. En effet, les contraintes opèrent en parallèle sur toutes les structures, et les contraintes de liaison permettent aux structures de chaque niveau de se contraindre mutuellement.

La possibilité d'aller-retours entre niveaux est donc formellement et techniquement envisageable. Mais à notre sens, la principale question est ailleurs : sur quelles bases le système pourra-t-il “deviner” quelles sont les informations à ajouter à la structure initiale ? Sur ce point, (Kahane et Lareau, 2005) reste particulièrement flou. A titre personnel, nous sommes plutôt sceptiques quant à l'intérêt et la faisabilité pratique d'un tel système[15].

3.9 Arbres à bulles

Terminons ce chapitre par l'étude d'une structure mathématique particulière, l'*arbre à bulles*, qui permet de modéliser élégamment les phénomènes réputés “difficiles” que sont l'extraction et la coordination. La formalisation présentée ici est essentiellement issue de (Kahane, 1997).

3.9.1 Motivation

Coordination

Si les arbres de dépendance sont parfaitement adaptés à la représentation des liens de *subordination*, ils ont beaucoup plus de difficultés à représenter une opération orthogonale comme la coordination, où plusieurs éléments sont regroupés pour occuper une seule position syntaxique : “`Pierre parle à Jean et Marie`”.

Une solution, déjà mentionnée par (Tesnière, 1959), consiste à regrouper les éléments coordonnés à l'intérieur d'une **bulle** de coordination.

[15]Notons qu'une fonctionnalité intégrée à XDG (cfr. chap 5) se rapproche un peu de ce système : il s'agit de l'utilisation d'un *Oracle* permettant de guider la recherche de solutions, par exemple via une guidance statistique. Mais les *Oracles* de XDG ne font qu'orienter la recherche, ils ne modifient en rien la structure de départ.

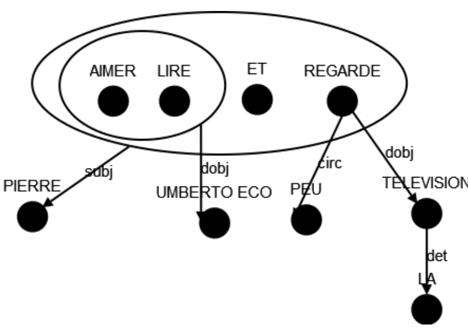


Fig. 3.12 – Arbre à bulles exemplifiant la coordination : “`Pierre aime lire Umberto Eco et regarde peu la télévision`”

Les éléments coordonnés à l'intérieur d'une bulle partagent obligatoirement leur gouverneur, mais le partage de leurs dépendants est optionnel. Ainsi, à la figure 3.12, “`Pierre`” est partagé par les éléments “`aime lire`” et “`regarde`”, mais “`Umberto Eco`” est le dépendant du seul “`aime lire`”.

Evidemment, les choses se compliquent lorsque certains éléments sont sous-entendus, comme dans les *coordinations elliptiques* : “`Le livre de Lionel s'est bien vendu, et celui d'Alfred aussi`”.

La coordination (et en particulier la coordination elliptique) est un phénomène linguistique aussi répandu que difficile à traiter, et constitue, avec l'extraction que nous abordons ci-dessous, une des principales pierres d'achoppement des formalismes linguistiques utilisés en TALN.

Extraction

L'extraction est un phènomène linguistique où l'on observe qu'un élément dépendant d'un noyau verbal est *extrait* de sa position linéaire usuelle pour venir occuper une position à l'extérieur de la proposition. Les topicalisations (“`Ce café, il ne me goûte vraiment pas`”), les interrogations indirectes (“`Qu'as-tu donc bu à ce banquet ?`”), et bien sûr les relatives (“`la pomme que j'avais envie de manger`”) en font partie. L'élément extrait et le noyau verbal se trouvant à une distance potentiellement illimitée, on parle donc également de *dépendances non bornées* à leur propos. Il s'agit bien sûr de structures *non projectives* (voir section 2.1.6).

Observons d'abord que le mot “`qu-`” occupe deux fonctions syntaxiques distinctes : dans une relative par exemple, il est la tête de la relative, mais il est également subordonné au noyau verbal dans lequel il occupe la fonction de sujet (“`qui`”), d'objet (“`que`”), de complément prépositionnel “`avec qui`”, etc. Dans la figure 3.13, nous représentons cet état de fait par l'utilisation de deux demi-noeuds.

La formalisation de l'extraction que nous esquissons à présent emprunte à (Tesnière, 1959) la notion de *nucléus*, et particulièrement de nucleus verbal et de nucleus nominal. Nous définissons le *nucleus verbal* comme une entité linguistique composée d'un verbe ou d'une entité complexe “assimilée”[16] Le *nucléus nominal* est un nom ou un pronom, ou une entité “assimilée”[17].

En regroupant les nucléus nominaux et verbaux d'une construction comportant une extraction, nous observons que le *çaractère non borné* de l'extraction qui était à l'origine de notre problème, s'évanouit de lui-même.

[16]Par exemple, un couple auxiliaire-participe (“`a parlé`”), un couple verbe-infinitif (“`veut manger`”), un triplet verbe-conjonction-verbe (“`crois que dévorer`”), un couple verbe-préposition (“`téléphone à`”), et des unités construites par transitivité à partir de ces dernières (“`crois qu'il faut téléphoner à`”).

[17]Couple déterminant-nom (“`quelle voiture`”), nom-nom-complément (*angl.* “`the daughter of which man`”), etc.

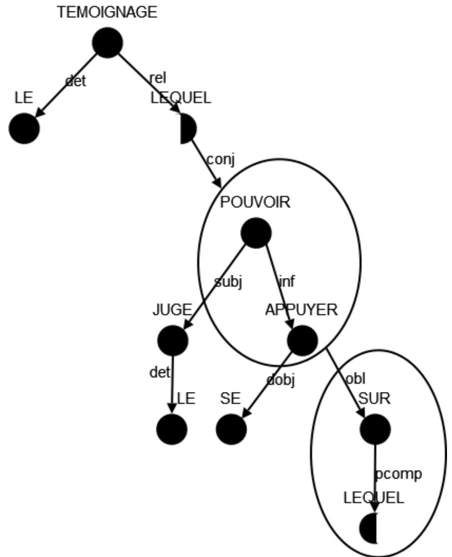


FIG. 3.13 – Arbre à bulles du syntagme nominal : “`Le témoignage sur lequel le juge pourra s’appuyer`”

3.9.2 Définition formelle

Définition. Un arbre à bulles est un quadruplet $(X, \mathfrak{B}, \phi, \triangleleft)$ où X est l’ensemble des noeuds de base, $\mathfrak{B}$ est l’ensemble des bulles, ϕ est un fonction de $\mathfrak{B}$ vers les sous-ensembles non-nuls de X (qui décrit donc le contenu des bulles) et $\triangleleft$ est une relation sur $\mathfrak{B}$ vérifiant :

1. Si $\alpha, \beta \in \mathfrak{B}$, alors $\phi(\alpha) \cap \phi(\beta) = \emptyset$ ou $\phi(\alpha) \subseteq \phi(\beta)$ ou $\phi(\beta) \subseteq \phi(\alpha)$;
2. Si $\phi(\alpha) \subset \phi(\beta)$, alors $\alpha \prec \beta$. Si $\phi(\alpha) = \phi(\beta)$, alors $\alpha \preceq \beta$ ou $\beta \preceq \alpha$.

La relation $\triangleleft$ est appelée **relation de dépendance-inclusion**. Deux sous-relations de $\triangleleft$ sont considérées :

1. La relation de **dépendance** $\lll$ définie comme suit : $\alpha \lll \beta$ si $\alpha \triangleleft \beta$ et $\phi(\alpha) \cap \phi(\beta) = \emptyset$. Nous dirons que α **dépend** de β si $\alpha \lll \beta$;
2. La relation d’**inclusion** $\odot$, définie comme suit : $\alpha \odot \beta$ si $\alpha \triangleleft \beta$ et $\alpha \subseteq \beta$. Nous dirons que α est **directement inclus** dans β si $\alpha \odot \beta$

Notons enfin que la **projection** d’une bulle α est définie comme l’union des contenus de toutes les bulles dominées par α, α incluse.

Graphiquement, nous représenterons la relation de dépendance $\lll$ par des arcs orientés entre noeuds (comme précédemment) , et la relation d’inclusion $\odot$ par l’inclusion dans une bulle.

3.9.3 Grammaires à bulles

(Kahane, 2002, p. 54) indique qu’il est possible d’étendre une grammaire de dépendance comme GUST pour manipuler des arbres à bulles, pour donner des *grammaires à bulles*, et donne l’intuition générale sous-tendant cette extension. Néammoins, à l’heure actuelle, il n’existe pas à ma connaissance de modélisation complète et entièrement formalisée de la coordination et de l’extraction sous GUST/GUP.

Notre implémentation ne contient donc pas pour l’heure de traitement des arbres à bulles, et nous laissons ce problème intéressant mais difficile pour d’éventuels travaux ultérieurs.

Chapitre 4

Interfaces Sémantique-Syntaxe

Cette section poursuit un double objectif :

- Présenter brièvement un traitement possible de la *portée* des quantificateurs grâce aux grammaires d'unification polarisées, par l'utilisation d'une structure hiérarchique logique couplée à la structure prédicative classique (section 4.1) ;
- Détailler une série de *modélisations* originales que nous avons élaborées concernant divers phénomènes linguistiques situés à l'interface entre la sémantique et la syntaxe, dont certains sont réputés "difficiles" car introduisant des distorsions importantes entre les deux structures (section 4.2).

4.1 Représentation logique et sous-spécification

La présente discussion est un résumé de l'article (Kahane, 2005). Nous y reprenons l'exemple suivant, qui nous servira de fil rouge :

$$\text{“}\texttt{Tout homme aime une femme}\text{''} \tag{4.1}$$

4.1.1 Structure des représentations logiques

Les deux représentation logiques usuelles de 4.1 sont

$$\forall x \;\; [homme(x) \rightarrow \exists y \;\; [femme(y) \wedge aimer(x,y)]] \tag{4.2}$$

$$\exists y \, [femme(y) \wedge \forall x \, [homme(x) \rightarrow aimer(x,y)]] \tag{4.3}$$

Nous choisissons ici l'interprétation 4.2. En s'autorisant la confusion entre un prédicat et son extension, on peut également utiliser la représentation suivante :

$$\forall x \in homme, [\exists y \in femme, \;\; aimer(x,y)] \tag{4.4}$$

On observe que l'équation 4.4 lie un quantificateur à trois objets : une *variable*, sa *restriction*, et sa *portée*. Ainsi, on peut, comme le fait par exemple (Woods, 1975), représenter une telle formule par un réseau sémantique tel que présenté à la figure 4.1.

Comme le montre cette dernière figure, les variables x et y ont pour seule utilité de *lier* les différentes positions de la formule. On peut donc supprimer celles-ci et les remplacer par des arcs orientés, ce qui nous donne un dag, comme l'illustre la figure 4.2. Les quantificateurs y sont représentés comme des opérateurs à deux arguments, leur restriction et leur portée.

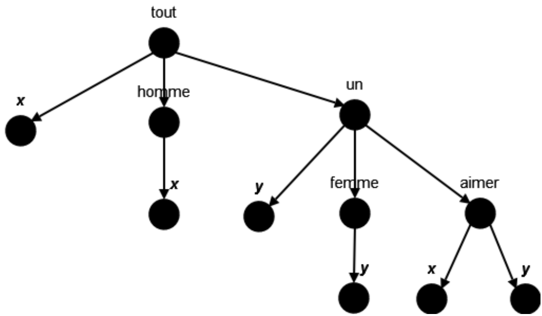


FIG. 4.1 – Réseau sémantique “à la Woods” de 4.1

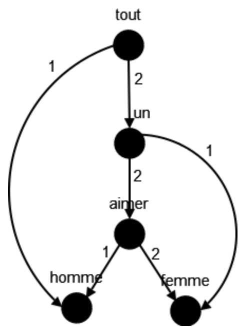


FIG. 4.2 – Graphe orienté acyclique représentant 4.1

Cette représentation a l'avantage d'être assez intuitive, et on peut démontrer qu'elle est formellement équivalente à une formule logique du 1er ordre - il suffit, pour assurer le passage de l'un à l'autre, d'opérer une réification sur les objets de la structure. Elle constituera notre *représentation sémantique*. Cette représentation est en fait la *superposition* de deux structures, comme nous le montre la figure 4.3 ; une structure *prédicative* - que nous avons déjà présentée à maintes reprises - et une *hiérarchie logique*.

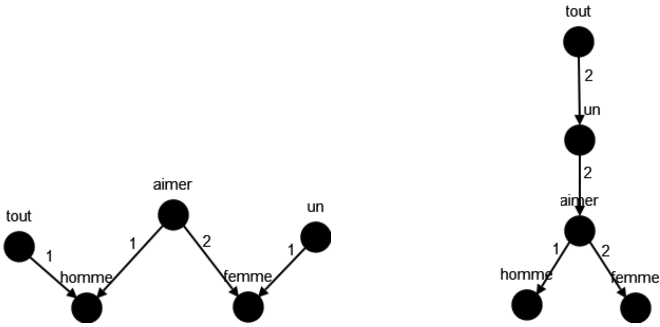


FIG. 4.3 – Structure prédicative (gauche) et hiérarchie logique (droite) composant la figure 4.2

4.1.2 Grammaires $\mathcal{G}_{pred}$, $\mathcal{G}_{log}$ et $\mathcal{I}_{sem-synt}$

Nous indiquons ici succintement la manière dont des GUP peuvent construire et apparier des structures prédicatives et logiques.

La figure 4.4 illustre un fragment de grammaire prédicative $\mathcal{G}_{pred}$ permettant de générer le graphe prédicatif de la figure 4.3, à gauche. La polarité d attachée à certains noeuds indique que ceux-ci sont indéterminés, si la polarité est ○, et déterminés si la polarité est ●.

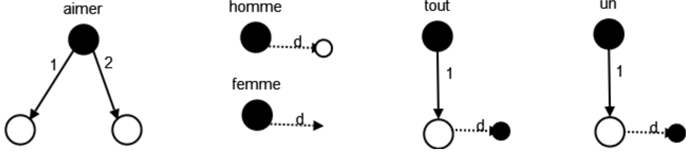


FIG. 4.4 – Grammaire prédicative $\mathcal{G}_{pred}$

La figure 4.5 illustre, elle, la grammaire logique $\mathcal{G}_{log}$, qui génère la hiérarchie logique de la figure 4.3, à droite. On observe que les noeuds sont *typés*.

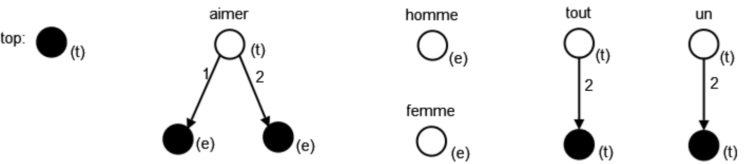


FIG. 4.5 – Grammaire logique $\mathcal{G}_{log}$

En peut alors construire, en superposant les deux grammaires, obtenir une grammaire sémantique $\mathcal{G}_{sem} = \mathcal{G}_{pred} \times \mathcal{G}_{log}$, qui possède évidemment une double polarité (p_{sem}, p_{log}). Lorsqu'un noeud n'apparaît pas, nous lui donnons une polarité neutre ● au niveau où il n'apparaît pas. Cette grammaire[1] est illustrée à la figure 4.6.

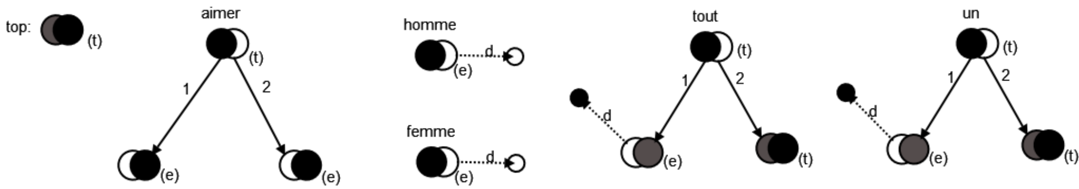


FIG. 4.6 – Grammaire sémantique $\mathcal{G}_{sem} = \mathcal{G}_{pred} \times \mathcal{G}_{log}$

L'interface sémantique-syntaxe $\mathcal{I}_{sem-synt}$ est similaire à celle que nous avions présentée aux sections 3.5.2 et 3.7.3. Seule la structure prédicative est prise en compte dans cette interface, les questions de portée n'ayant pas d'influence sur la structure syntaxique en tant que telle[2]. Nous illustrons un fragment à la figure 4.7.

[1]En toute rigueur, cette grammaire doit être enrichie pour assurer que la restriction d'un quantificateur est dans sa portée, voir (Kahane, 2005)

[2]Il existe par contre probablement un lien - mais qui n'a jamais, à notre connaissance, été formalisé dans le cadre de la TST - entre hiérarchie logique et structure communicative. Cette dernière ayant elle-même un lien évident avec les configurations topologiques et prosodiques possibles (les marques intonatives et l'ordre linéaire permettant au locuteur d'accentuer ou diminuer la saillance de tel ou tel objet linguistique), la portée pourra donc avoir une influence sur les formes "de surface".

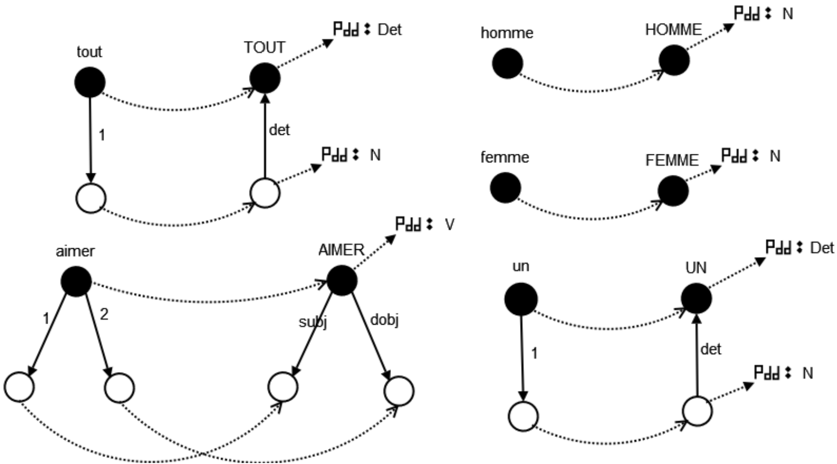


FIG. 4.7 – Interface sémantique-syntaxe $\mathcal{I}_{sem-synt}$

4.1.3 Sous-spécification

Lorsque nous utilisons notre interface dans le sens de l'analyse (de la syntaxe au sens), nous sommes évidemment confrontés au problème de l'ambiguité concernant la portée respective des quantificateurs (voir formules 4.2 et 4.3) Il s'agit d'une ambiguité qu'il est impossible de résoudre sans faire appel à des ressources de niveaux extérieurs - la prosodie, le contexte, etc. Il est donc préférable d'utiliser une représentation sémantique où la hiérarchie logique est sous-spécifiée. La figure 4.8 en donne un exemple (les noeuds qui pourront se superposer sont placés en vis-à-vis).

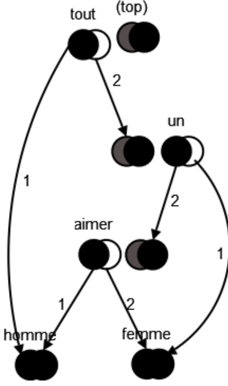


FIG. 4.8 – Représentation sous-spécifiée de 4.1

4.1.4 Implémentation

Nous l'avons dit, les phénomènes de portée n'ont normalement aucune influence sur la structure syntaxique générée (mais bien sur la topologie et la prosodie). Au vu de cette observation, nous avons donc décidé de ne *pas* implémenter de représentation logique dans notre interface, puisque celle-ci n'aurait aucune utilité dans le sens de la génération - et resterait sous-spécifiée dans le sens de l'analyse.

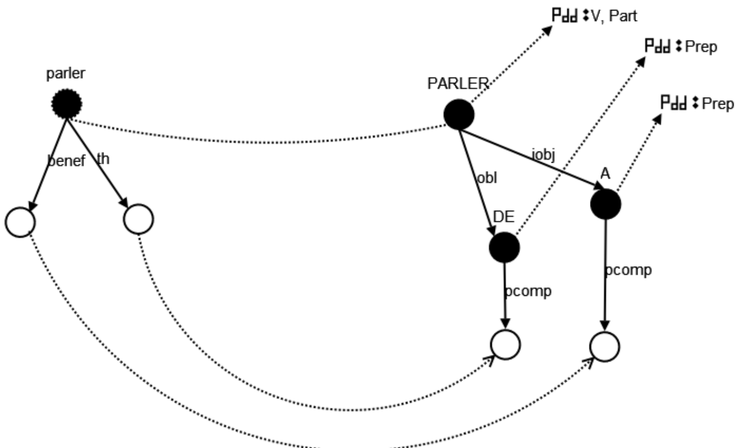
4.2 Phénomènes linguistiques abordés

Nous présentons ici, sans aucune exhaustivité, quelques phénomènes linguistiques (à l'interface entre la sémantique et la syntaxe) dignes d'intérêt que nous avons modélisés dans le cadre de ce travail.

4.2.1 Sous-catégorisation

Le *cadre de sous-catégorisation* (angl. *subcategorization frame*), aussi appelé le *régime* d'une unité linguistique est le nombre et le type des arguments syntaxiques avec lesquels il co-occure.

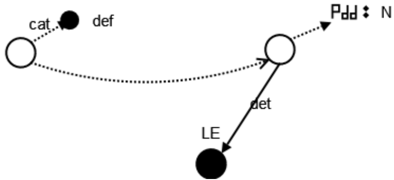
Ainsi, le cadre de sous-catégorisation du verbe "parler" est constitué, outre du sujet, de deux compléments indirects optionnels, respectivement régis par les prépositions "à" et "de".



Notons que la fonction *sujet* est absente dans cette règle ; nous avons en effet préféré l'exprimer dans une règle séparée[3]. Nonobstant cela, la règle illustrée à la figure ci-dessus permet de rendre compte de la sous-catégorisation complète du verbe. Nous n'avons pas considéré dans cette règle le caractère *optionnel* des compléments introduits par "de" et "à". Cet aspect peut être aisément pris en compte en ajoutant de nouvelles règles traitant des différentes constructions.

4.2.2 Articles définis, indéfinis et partitifs

La figure ci-dessous illustre la réalisation du trait grammatical ‹cat : def› ("catégorie définie") par l'article défini "LE". ‹cat› peut prendre 3 valeurs distinctes : ‹def›, ‹undef› et ‹partitif›.

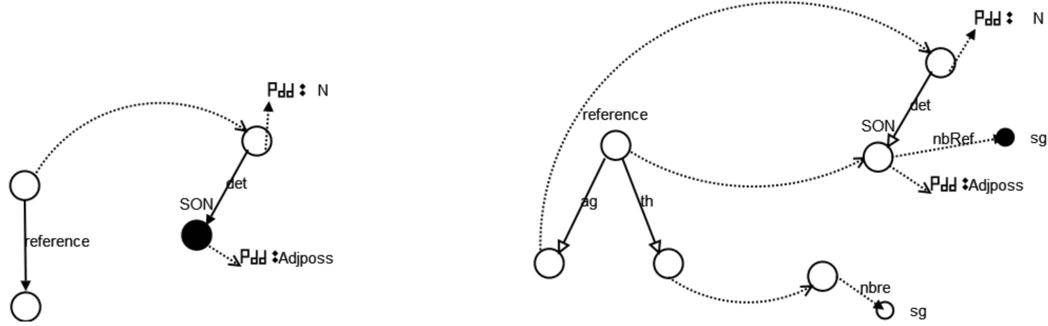


4.2.3 Anaphore

Les deux figures ci-dessous illustrent la réalisation d'une référence anaphorique par un adjectif possessif. A gauche, nous observons la génération de l'adjectif proprement dite, et à droite, une

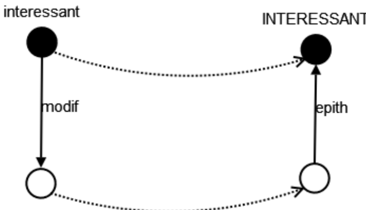
[3]En effet, la présence d'une fonction sujet n'est pas spécifique au verbe parler. Il s'agit, en français, d'une obligation pour tous les verbes à l'indicatif. Ce n'est pas le cas de l'italien ou de l'espagnol, où le sujet peut être omis : "Habla español", ‹Il/Elle parle espagnol›.

règle grammaticale. La morphologie de ce dernier est déterminée par 4 grammènes : le "`genre`" et le "`nombre`" du nom auquel il est subordonné, et la "`personne`" et le "`nombre`" de sa référence.



4.2.4 Modifieurs

Les modifieurs tels que les adjectifs (modifieurs du nom) et les adverbes (modifieurs du verbe) ont la particularité d'inverser le sens de la relation qu'ils entretiennent avec leur objet. Ainsi, pour la figure ci-dessous, ‹`interessant`› est au niveau sémantique un prédicat à un argument, mais au niveau syntaxique, "`INTERESSANT`" est subordonné au nom auquel il se rattache.

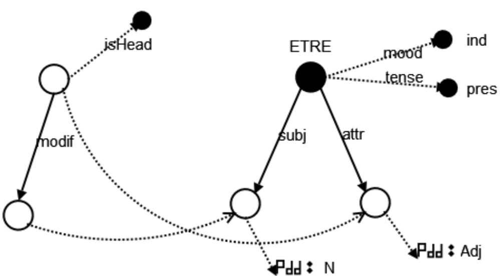


4.2.5 Copule

Selon la théorie de la translation de Tesnière (voir section 2.1.7), la copule - dont l'exemple prototypique est le verbe "`être`" - est un translatif permettant de modifier un adjectif en verbe.

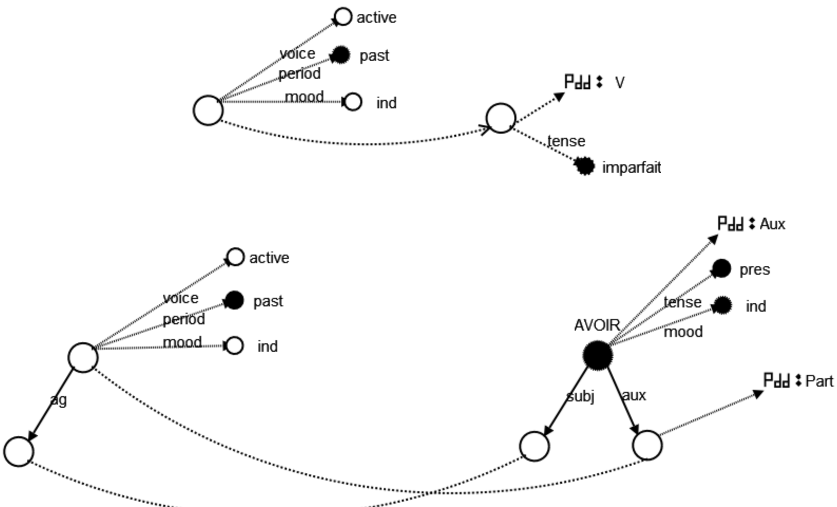
Notre modélisation est illustrée ci-dessous. La présence du grammène ‹`isHead`› peut interroger ; à première vue, la représentation sémantique devrait être uniquement constituée du modifieur (l'adjectif) et du modifié (le nom). Ce serait oublier que l'insertion de la copule n'est admissible que si l'adjectif dénote la *tête communicative* de l'énoncé. Sans cette contrainte, notre interface pourrait tenter de générer des phrases telles que "`* Pierre veut manger la pomme est rouge`".

En fait, le caractère grammatical de l'utilisation d'une copule est subordonnée à la structure communicative ; c'est en effet elle qui fixe quel est le noeud au centre de la représentation sémantique. Mais notre implémentation n'inclut à l'heure actuelle aucun module permettant de réellement traiter la structure communicative... Pour pallier à ce manque, nous utilisons donc un grammène spécifique ‹`isHead`› indiquant la tête communicative de l'énoncé.



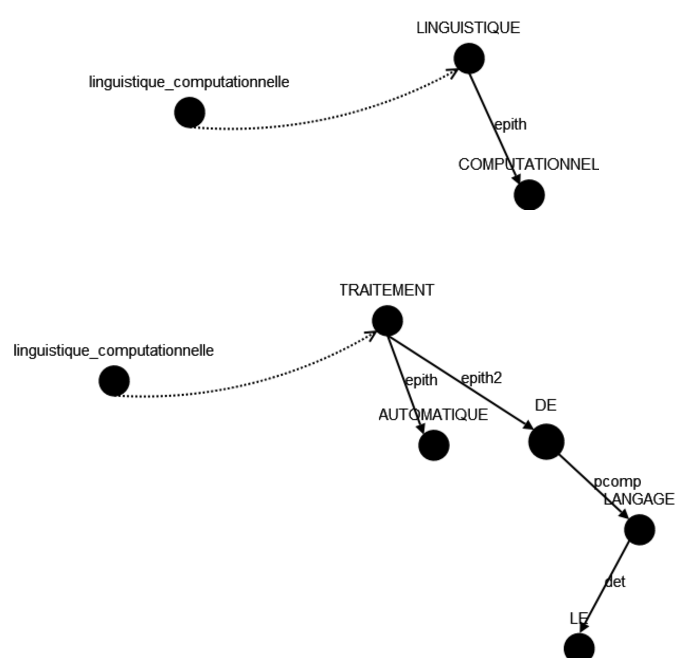
4.2.6 Conjugaison

Pour le même sémantème grammatical ‹period : past›, deux possibilités s'offrent à nous : l'utilisation de l'imparfait ou du passé composé[4]. Le traitement de l'imparfait est trivial, et le passé composé fait bien sûr usage de l'auxiliaire (que nous simplifions ici au verbe "avoir"[5]).



4.2.7 Synonymes

Le traitement des synonymes est très aisé en GUST/GUP : deux constructions syntaxiques sont synonymes si elles correspondent au même élément sémantique. Lors de la génération, le moteur d'inférence aura deux possibilités pour la saturation de cet élément et distribuera donc son analyse en fonction de celles-ci.



[4]Comme me l'a fait remarquer à juste titre F. Lareau, cette équivalence postulée entre imparfait et passé composé ne représente que très grossièrement la réalité de la langue ; il existe des différentes sémantiques importantes entre ces deux temps. Nous concervons celle-ci pour les besoins de la démonstration, mais en gardant toujours à l'esprit que celle-ci constitue une simplification linguistiquement peu fondée.

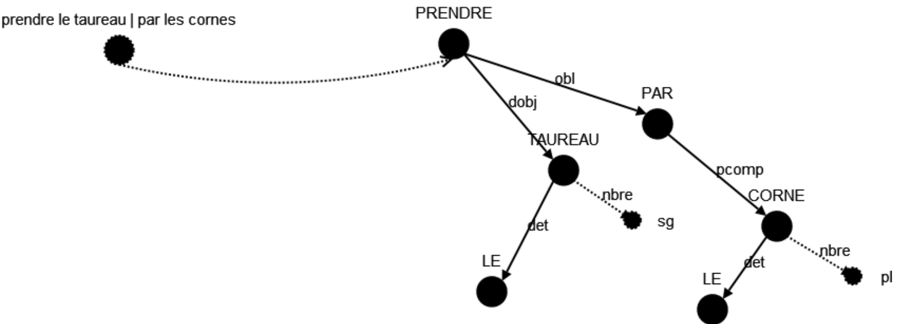
[5]On pourrait affiner la description par l'utilisation d'un trait syntaxique lié à certains verbes, indiquant que celui-ci exige l'auxiliaire "être" au passé composé, et prendrait lieu et place de l'auxiliaire "avoir"

4.2.8 Expressions figées

Les expressions figées sont « des unités polylexicales présentant un caractère figé définies selon deux types de contraintes : syntaxiques (liberté restreinte) et sémantique (opacité) » (Gross, 1996).

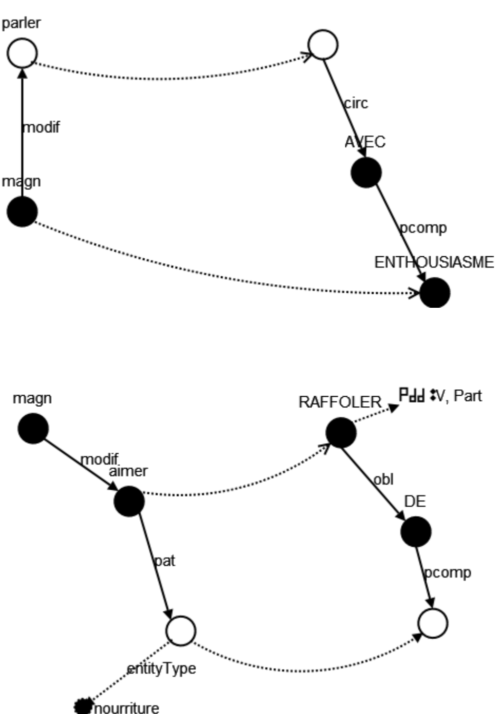
Prenons l'exemple de la locution "`Prendre le taureau par les cornes`". Sa liberté syntaxique est évidemment restreinte : "`* Il prend le grand taureau par les cornes`", et son sens est également opaque (l'expression est tout à fait idiomatique, le sens ne peut être déduit de ses composantes).

GUST permet de modéliser très simplement les expressions figées :



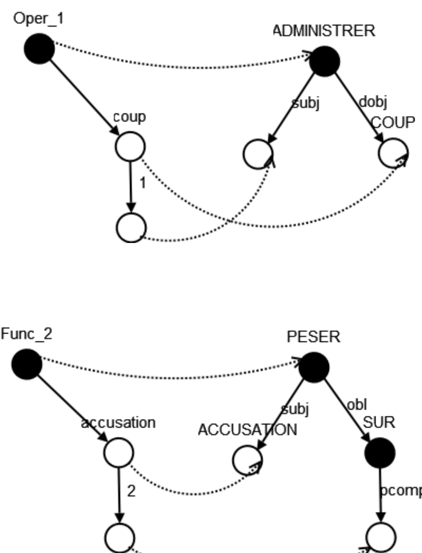
4.2.9 Collocations

Comme nous l'avions indiqué à la section 2.2.4, les Fonctions Lexicales de la Théorie Sens-Texte sont un excellent moyen de modéliser les collocations. En voici deux exemples basés sur la fonction $Magn$:



4.2.10 Verbes supports

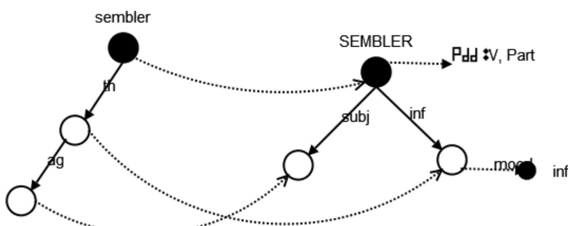
Les constructions à verbes support peuvent également être modélisées par des Fonctions Lexicales, comme le montrent les figures ci-dessous présentant les fonctions $Oper_1$ et $Func_2$ (cfr. section 2.2.4).



4.2.11 Verbes de montée

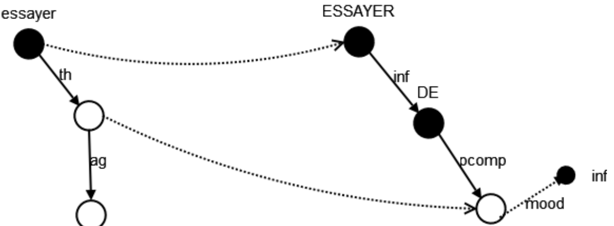
Nous appelons “verbe de montée” tout sémantème verbal s’exprimant par un prédicat dont l’un des arguments est un verbe dont un des arguments va “monter” en position de sujet syntaxique du verbe de montée. Exemple prototypique : “`sembler`” dans “`Pierre semble dormir`”.

Signalons au passage que l’insertion du grammène “`mood : inf`” sur le verbe qui monte (“`dormir`”) permet d’interdire à celui-ci d’acquérir un sujet syntaxique, étant donné que seuls les verbes à l’indicatif admettent un sujet dans notre grammaire.



4.2.12 Verbes de contrôle

Un verbe de contrôle est un verbe dont la représentation sémantique est un prédicat à au moins deux arguments, dont l’un est un verbe et l’autre est à la fois argument du verbe de contrôle et du verbe “contrôlé”. Exemple : “`essayer`” dans “`Pierre essaye de dormir`”.

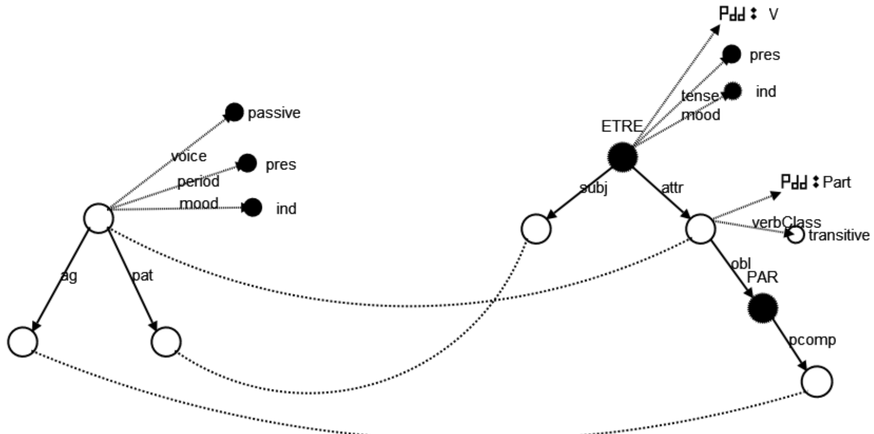


4.2.13 Passivation

Notre traitement de la passivation est illustré à la figure ci-dessous. Remarquons que cette modélisation a le grand avantage d'être *non transformationnelle*, ie. contrairement à la plupart des formalismes, elle ne décrit *pas* le passif comme une transformation de l'actif, mais comme une règle classique de réalisation syntaxique.

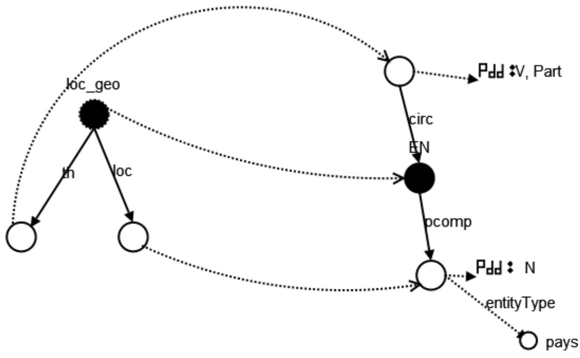
De nombreux travaux (Candito, 1999) ont démontré que l'utilisation de règles transformationnelles au sein du lexique posait de très sérieux problèmes dans l'architecture du lexique (toute modification ultérieure d'une règle risquant de rendre l'ensemble incohérent). Notre grammaire évite donc cet écueil, et ce au contraire de la plupart des formalismes actuels.

Remarquons bien sûr que l'application de la règle est subordonnée au caractère transitif du verbe. La règle est ici limitée au traitement de la forme du présent, la forme du passé composé passif (“`la pomme a été mangée par Pierre`”) étant plus compliquée.



4.2.14 Compléments circonstanciels

Terminons par une modélisation de compléments circonstanciels : temps, manière, lieu, but, cause, etc. La figure ci-dessous illustre un locatif, plus particulièrement un locatif exprimant un lieu “géographique”.



Chapitre 5

Axiomatisation de GUST/GUP

Ce chapitre sera consacré à l'axiomatisation de notre formalisme en un **problème de satisfaction de contraintes** (angl. *Constraint Satisfaction Problem*, ou CSP) :

- Nous commençons par préciser ce que recouvre la notion de *programmation par contraintes* ;
- Nous introduisons ensuite dans la section 5.2 *Extensible Dependency Grammar* [XDG], un nouveau formalisme grammatical conçu par Ralph Debusmann, et pour lequel existe un environnement de développement complet, *Extensible Development Kit* [XDK], écrit en Mozart/Oz et utilisant la programmation par contraintes ;
- Nous montrons enfin dans 5.3 comment axiomatiser GUST/GUP en une grammaire XDG.

5.1 Programmation par contraintes

5.1.1 Généralités

Le traitement du langage, tel qu'il est réalisé par des êtres humains, émerge de l'interaction simultanée de nombreuses contraintes situées à différents niveaux d'analyse. Diverses expériences de psycholinguistique ont montré que le traitement d'énoncés linguistiques par des êtres humains n'opérait pas de manière incrémentale, mais résultait plutôt de l'action conjointe et parallèle de nombreuses contraintes linguistiques et pragmatiques, relayées par des "modules" cognitifs spécialisés et extrêmement performants (O'Grady *et al.*, 1996, p. 438-460).

L'utilisation de la **programmation par contraintes** [CP ci-après] en TALN cherche en quelque sorte à imiter ce processus, en séparant nettement les questions déclaratives (les connaissances lexicales et grammaticales) des questions procédurales. La notion de **contrainte** est ici définie comme une relation formelle entre un ensemble d'inconnues appelées **variables**.

L'approche par contraintes se différencie très nettement des techniques classiques d'analyse en TALN (i.e. *"chart parsing"*), où l'algorithme construit une analyse complète en combinant des morceaux d'analyse. La CP se distingue en ceci qu'elle spécifie des conditions *globales* de bonne formation et cherche à en énumérer les *modèles*. Pour une phrase de longueur n, il existe un nombre fini d'arbres ou de graphes permettant de relier n noeuds, et l'objectif de l'analyse sera de sélectionner parmi ce nombre l'ensemble de ceux qui sont grammaticaux.

Au contraire des algorithmes d'unification, cette opération ne s'effectue pas en générant directement des structures et en vérifiant leur grammaticalité, mais se fonde sur l'*élimination* progressive des **modèles**[1]. La programmation par contraintes cherche donc une solution de manière *indirecte*, en supprimant au fur et à mesure les assignations qui ne vérifient pas les conditions, et ce jusqu'à aboutir aux solutions finales.

[1]La notion de modèle est ici à comprendre au sens mathématique du terme, cf. la *théorie des modèles* en logique mathématique : m est un modèle d'une formule α ssi la formule α est vraie dans le modèle m. Il s'agit donc en quelque sorte d'un "monde possible" de la formule α.

La recherche de solutions se décline en deux processus : la **propagation** et la **distribution**. La propagation est l'application de *règles d'inférence* déterministes en vue de réduire l'*espace de recherche*. La distribution correspond quant à elle à un *choix* non-déterministe. Les deux processus sont bien sûr liés : la distribution a lieu lorsque la propagation ne permet plus de réduire l'espace de recherche et est directement suivie par une nouvelle étape de propagation. Les deux opérations se succèdent donc jusqu'à aboutir à une solution.

Il est important de signaler que la distribution est une opération coûteuse en termes computationnels, et il est donc essentiel de réaliser des propagations aussi "fortes" que possible, et permettant ainsi de limiter au maximum le nombre d'étapes de distribution[2].

Terminons par mentionner que la technologie actuelle en CP a beaucoup évolué ces dernières années et a démontré sa capacité à résoudre de nombreux problèmes pratiques à forte complexité combinatoire : planification, gestion des stocks, conception de circuits électroniques, optimisations en recherche opérationnelle, calcul numérique, bioinformatique, et bien sûr le TALN.

5.1.2 Définitions

Soit une suite finie de variables $\mathcal{Y} := y_1, y_2, ..., y_k$ (avec $k > 0$), avec pour domaines respectifs $D_1, D_2, ..., D_k$. Les valeurs possibles d'une variable y_i se situent donc à l'intérieur de D_i.

Une **constrainte** C sur l'ensemble de variables $\mathcal{Y}$ est définie comme un sous-ensemble $D_1 \times D_2 \times ... \times D_k$. Lorsque $k = 1$ (resp. $= 2$), la contrainte est dite unaire (resp. binaire).

Par **problème de satisfaction de contraintes** [CSP], nous dénotons un ensemble fini de variables $\mathcal{X} := x_1, x_2, ..., x_n$, avec pour domaines respectifs $D_1, D_2, ..., D_n$ associé à un ensemble fini $\mathcal{C}$ de contraintes, dont chacune porte sur un sous-suite de $\mathcal{X}$. Nous notons un tel CSP par le couple $(\mathcal{C}, \mathcal{DE})$, où $\mathcal{DE} := x_1 \in D_1, x_2 \in D_2, ...x_n \in D_n$.

La solution d'un CSP est bien sûr une suite de valeurs admissibles pour l'ensemble des variables, i.e. où toutes les contraintes sont **satisfaites**. Un n-uple $\sigma = (d_1, d_2, ...d_n)$ appartenant à $D_1, D_2, ..., D_n$ satisfait une contrainte $C \in \mathcal{C}$ portant sur les variables $x_{i_1}, ..., x_{i_m}$ si $(d_{i_1}, ..., d_{i_m}) \in C$. Il s'agit donc d'une assignation σ de valeurs tels que C est satisfait.

En CP, l'assignation σ est calculée incrémentalement par une suite d'approximations successives, i.e. les domaines contenant les valeurs possibles sont progressivement réduits jusqu'à aboutir à une solution (ou à un résultat contradictoire, si le CSP est insoluble). Ce calcul se décline, nous l'avons dit, en deux phases alternants l'une l'autre : la propagation et la distribution.

Dans la programmation par contraintes *concurrente* (comme implémentée en Mozart/Oz), σ s'intitule un *store* et les contraintes primitives sont instanciées par des agents en concurrence qui observent le store et cherchent en permanence à améliorer l'approximation en dérivant de nouvelles contraintes basiques en accord avec leur sémantique déclarative.

5.1.3 Types de contraintes

Il existe bien entendu de nombreux types de contraintes. En pratique, nous utiliserons dans notre implémentation trois familles de contraintes :

1. Les contraintes sur les **ensembles finis** (angl. *Finite Set Constraints*) sont au coeur du formalisme. Elles servent dans notre implémentation à décrire tout ce qui a trait aux *problèmes de configuration de graphes* - cfr. (Debusmann *et al.*, 2004c) - i.e. la formalisation de contraintes linguistiques telles que la valence, les relations de dominance, les restrictions lexicales, l'accord, l'interface entre différents niveaux, etc. Par exemple, nous pouvons assurer que chaque verbe fini possède un (et un seul) sujet en spécifiant que

[2] Dans notre implémentation, le nombre d'étapes de distribution dépasse ainsi rarement la dizaine, et de nombreuses analyses sont possibles sans aucune distribution.

$$\forall n \in nodes : pos(n) =' V' \Rightarrow |subj(n)| \in \{1\} \quad (5.1)$$

Les variables sur des ensembles finis portent sur des ensembles fini d'entiers. Une variable S est approximée par une limite inférieure $\lfloor S \rfloor$ et une limite supérieure $\lceil S \rceil$, telle que :

$$\lfloor S \rfloor \subseteq S \subseteq \lceil S \rceil \quad (5.2)$$

La propagation sera bien sûr utilisée pour réduire au maximum ces limites.

2. Les contraintes de **sélection** sont utilisés pour traiter efficacement de l'ambiguité lexicale. Soit un mot w possédant plusieurs entrées lexicales possibles $L_1, ...L_n$. Nous introduisons la variable E_w pour dénoter l'entrée lexicale qui sera *in fine* sélectionnée entre elles, et un variable entière I_w dénotant sa position dans la séquence. Nous pouvons alors lier ses variales par la contrainte suivante, avec pour sémantique déclarative $E_w = L_{I_w}$:

$$E_w = \langle L_1, ...L_n \rangle [I_w] \quad (5.3)$$

3. *"Deep Guards"* pour propagateurs disjonctifs : XDG fait usage de la construction `or` G_1 `[]` G_2 `end` pour assurer efficacement le respect de conditions mutuellement exclusives.

5.2 *Extensible Dependency Grammar*

Extensible Dependency Grammar est un nouveau *formalisme grammatical* développé par Ralph Debusmann (notamment) dans le cadre de sa thèse de doctorat (Debusmann, 2006). Nous pouvons résumer ses principales caractéristiques en quatre points :

1. Utilisation des **Grammaires de dépendance** comme type de représentation syntaxique : voir le chapitre 3 pour une description de cette famille de théories linguistiques. .
2. Syntaxe **"model-theoretic"** : la grammaire est définie comme une *description* logique de modèles bien formés, et où une expression E est considérée grammaticale selon G ssi E est un *modèle* de G. La perspective "model-theoretic" est clairement plus déclarative que son pendant "proof-theoretic", car elle se détache complètement des mécanismes procéduraux pour ne se centrer que sur la description linguistique.
3. **Architecture parallèle** : contrairement à la plupart des formalismes grammaticaux, XDG n'est pas "syntactico-centrique" mais prend en compte, sur un pied d'égalité, des différents niveaux de représentation linguistique, comme le prescrit (Jackendoff, 2002).
4. **Analyse par contraintes** : XDG est entièrement basé sur la programmation par contraintes.

XDG a été conçu de manière *modulaire* et *multi-stratale* ; les grammaires peuvent ainsi être étendues à souhait pour traiter d'aspects linguistiques aussi divers que la sémantique (prédicat-argument, portée, structure communicative), syntaxe, topologie, morphologie, prosodie, etc.

Il suffit pour le concepteur d'une grammaire d'indiquer quels sont les niveaux linguistiques qu'il entend étudier, et de spécifier pour chacun d'eux, de manière totalement *déclarative*, les conditions de bonne formation et les règles d'interface.

5.2.1 Formalisation

Une **grammaire XDG** est définie comme un langage de description de **multigraphes**.

Un multigraphe est un *graphe multi-dimensionnel* constitué d'un nombre quelconque de graphes (appelés **dimensions**) partageant le même ensemble de noeuds.

Nous illustrons à la figure 5.1 un exemple de multigraphe constitué de trois dimensions : PA (*Predicate-Argument*), qui constitue l'équivalent de notre représentation sémantique, ID (*Immediate Dominance*), l'équivalent de notre représentation syntaxique, et LP (*Linear Precedence*), qui traite de la topologie. Les noeuds grisés dénotent des noeuds supprimés dans une dimension.

Un multigraphe est défini formellement par un tuple $(V, Dim, Word, W, Lab, E, Attr, A)$, où :

V est un ensemble fini de noeuds ;

Dim est un ensemble fini de dimensions ;

$Word$ est un ensemble fini de mots ;

$W \in V \rightarrow Word$ est une fonction attribuant un mot à chaque noeud ;

Lab est un ensemble fini d'étiquettes (pour les arcs orientés) ;

$E \subseteq V \times V \times Dim \times Lab$ est un ensemble fini d'arcs orientés ;

$Attr$ est un ensemble fini d'attributs complémentaires (traits grammaticaux), arrangés dans des matrices AVM (*Attribute-Value Matrices*) ;

$A \in V \rightarrow Dim \rightarrow Attr$ est une fonction attribuant des attributs à chaque noeud.

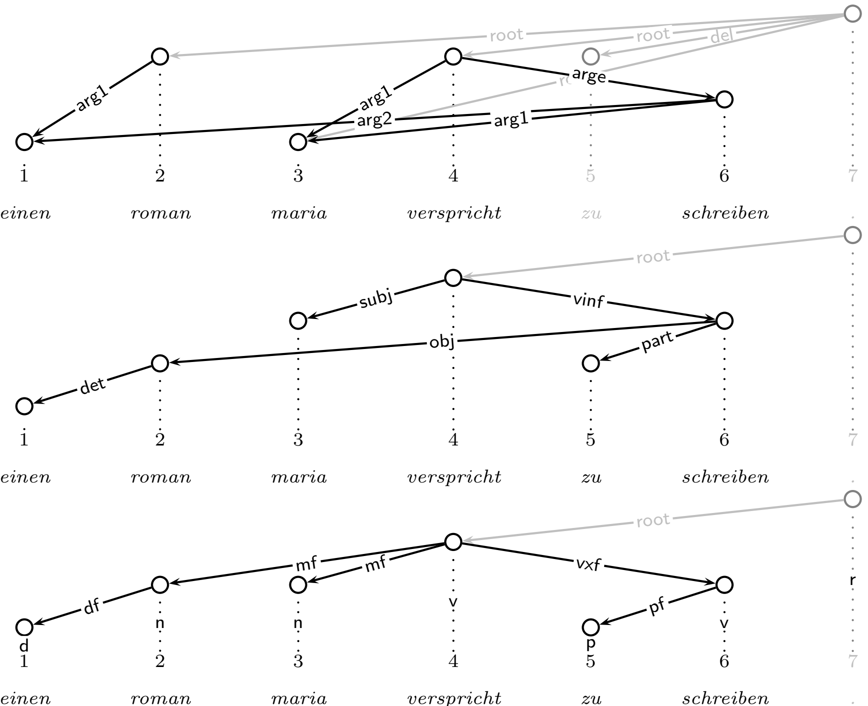


FIG. 5.1 – Analyse XDG de la phrase "`Einen roman Maria verspricht zu schreiben`", ‹Maria promet d'écrire un roman› : niveau sémantique dans la figure du haut, syntaxique dans la figure du milieu, et topologique dans la figure du bas.

Pour chaque dimension $d \in Dim$, nous pouvons induire deux relations :

Arc étiqueté $\xrightarrow{\cdot}_d$. Soit deux noeuds v_1 et v_2 et une étiquette $l \in Lab$, la relation $v_1 \xrightarrow{l}_d v_2$ sera définie comme valide ssi il existe un arc de v_1 à v_2 étiquetée l sur la dimension d :

$$\xrightarrow{\cdot}_d = \{(v_1, v_2, l) \mid (v_1, v_2, d, l) \in E\} \tag{5.4}$$

Précédence $\prec$. [3] Soit deux noeuds v_1 et v_2, la relation $v_1 \prec v_2$ est définie comme valide ssi $v_1 < v_2$, où $<$ désigne l'ordre total sur les nombres naturels.

Opérer l'analyse ou la synthèse d'un énoncé linguistique équivaut à énumérer l'ensembler de ses *modèles*, qui sont, nous l'avons dit, des multigraphes. Il est possible de démontrer (Debusmann et Smolka, 2006) qu'en toute généralité, l'analyse XDG est un problème NP-complet, par réduction à partir de `SAT`.

[3] Attention, la relation $\prec$ définie ici n'a rien à voir avec celle mentionnée à la section 2.1.6

Néammoins, les grammaires déjà conçues pour XDG (pour les langues suivantes : arabe, tchèque, néérlandais, anglais, français et allemand) montrent en pratique un comportement calculatoire bien meilleur que celui théoriquement prédit[4]. Des recherches sont en cours en vue de trouver des fragments de XDG dont la complexité peut être garantie polynomiale.

Par manque de place, nous arrêtons là notre discussion des aspects formels de XDG. Le lecteur intéressé pourra se référer à (Debusmann, 2006) pour plus d'informations.

5.2.2 Grammaires XDG

Concevoir une grammaire XDG concrète se décline en trois étapes :

1. Définition des **dimensions** et des structures de traits composant chacune de celles-ci : à chaque dimension est associé un nom, un ensemble de labels possibles pour les arcs orientés, et un ensemble d'attributs attachés au noeuds.
2. Définition du **lexique**, décrivant les comportements linguistiques de chaque entité. XDG étant un formalisme *lexicalisé*, les descriptions grammaticales sont, à la base, attachés aux unités individuelles (les “mots”). Bien sûr, il est possible de factoriser une partie de ces descriptions pour obtenir une grammaire moins redondante.
   Cette factorisation est réalisée par l'utilisation d'une hiérarchie de classes lexicales. Ces classes lexicales peuvent être combinées mais aussi être “disjonctées”, i.e. il est possible de spécifier qu'une entrée lexicale appartient à une classe *ou* à une autre.
3. Définition des **principes** : contraintes globales à appliquer aux modèles. Les contraintes assurant l'analyse sont intégrées dans une *librairie de principes*.
   Ces principes sont composés de foncteurs de contraintes. Le principe de *valence* se compose ainsi de deux foncteurs dénommés *in* et *out*, qui contraignent respectivement les arcs rentrants et sortants de chaque noeud, selon les spécifications de la grammaire. A chaque principe est associé une *priorité* qui permet de réguler l'ordre dans lequel les contraintes sont appliquées.
   Bien sûr, cette librairie de contraintes est extensible et de nouveaux principes peuvent être écrits (en Oz) et intégrés au XDK, comme nous l'avons d'ailleurs fait.

5.2.3 *XDG Grammar Development Kit*

XDK est un environnement de développement complet pour XDG. Il définit trois syntaxes concrètes pour la spécification des grammaires ainsi qu'une série d'outils de tests et de debugging inclus dans une GUI.

Les trois syntaxes qu'il est possible d'utiliser pour définir des grammaires sont :

1. *User Language* : syntaxe lisible, utilisée pour spécifier “manuellement” des grammaires ;
2. *XML Language* : équivalent XML de la syntaxe UL, surtout utilisée pour le développement automatique à partir de corpus ;
3. *Inermediate Language* : syntaxe utilisée comme représentation intermédiaire entre une spécification UL ou XML, et la structure compilée utilisée par le moteur d'inférence.

De plus, XDK utilise une syntaxe de grammaire “compilée”, dénommée *Solver Language*, utilisée par le moteur d'inférence par contraintes. Bien évidemment, des compilateurs existent pour passer d'une représentation à l'autre, comme l'illustre la figure 5.2.

Pour implémenter notre interface sémantique-syntaxe, nous avons essentiellement travaillé via la syntaxe UL. Nous reproduisons dans le listing 5.1 un extrait de notre grammaire écrite au format UL. La signification des différents champs sera détaillée dans les sections suivantes.

[4]Notons au passage que des formalismes aussi populaires que LFG et HPSG sont aussi des formalismes NP-complets, ce qui n'empêche en rien leur utilisation massive par les linguistes.

Listing 5.1 – Extrait d'une grammaire GUST transcrite au format UL

```
defentry {
"N"&
"Root-gsem"&
"Root-gsynt"&
"Root-isemsynt"

dim gsem {       id:   poivre
                 grams:  {[entitytype  nourriture]}
                 name:  "poivre"}
dim gsynt {      id:   poivre
                 pos:  {n}
                 grams:  {[genre  masc]  [nametype  acceptepartitif]}
                 name:  "POIVRE"}
dim lex{         word:  "poivre"}}

defentry {
"N"&
"Root-gsem"&
"Root-gsynt"&
"Root-isemsynt"

dim gsem {       id:   pomme_de_terre
                 name:  "pomme_de_terre"}
dim gsynt {      id:   pomme
                 pos:  {n}
                 grams:  {[genre  fem]}
                 name:  "POMME"}
dim isemsynt {   link:{{  synt:{    syntOut:  {epith}
                                    syntOutOut:  {[epith  pcomp]}
                                    syntOutOutLabels:  {[epith  {pcomp:{terre}}]}
                                    syntOutLabels:  {epith:{de}}
                                    group:{after:{[de  terre]}}}
                          lexicalised:     true}}}
dim lex{         word:  "pomme_de_terre"}}

defentry {
"V"&
"Part"&
"Root-gsem"&
"Root-gsynt"&
"Root-isemsynt"

dim gsem {       id:   parler
                 out:  { ag!  benef!  th!}
                 name:  "parler"}
dim gsynt {      id:   parler
                 pos:  {v  part}
                 name:  "PARLER"}
dim isemsynt {   link:{{  sem:{     semOut:  {benef  th}}
                          synt:{    syntOut:  {obl  iobj}
                                    syntOutOut:  {[obl  pcomp]  [iobj  pcomp]}
                                    syntOutPOS:  {obl:{prep}}
                                    syntOutLabels:  {obl:{de}  iobj:{a}}
                                    group:{after:{[de  a]}}}
                          linking:{subcats_start:  {benef:{iobj}th:{obl}}
                                   subcats_end:  {benef:{pcomp}th:{pcomp}}}
                          lexicalised:     true}}}
dim lex{         word:  "parler"}}
```

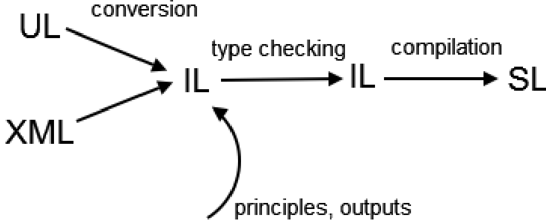


FIG. 5.2 – Etapes du traitement d'une grammaire XDG

5.2.4 Evolution future du formalisme

XDG est un formalisme très récent et sera à n'en point douter appelé à se développer significativement dans les prochaines années, tant au niveau théorique que technique.

Mentionnons brièvement quelques axes majeurs de développement :
- Compréhension plus profonde de la complexité calculatoire de XDG, et découverte d'éventuels fragments garantis polynomiaux.
- Utilisation de la libraire de contraintes *GeCode* (Schulte et Stuckey, 2004) en lieu et place de la librairie standard fournie en Mozart/Oz et ajout de *contraintes globales* dans XDG pour intensifier la propagation.
- Amélioration de la *stratégie de distribution* : il est possible d'optimiser sensiblement celle-ci grâce à l'utilisation d'une *guidance statistique* : un "Oracle" fournit à l'analyseur des informations statistiques lui permettant d'effectuer sa recherche en se dirigeant directement vers les solutions les plus probables. L'algorithme sous-jacent est une heuristique de type A*, cfr. (Dienes *et al.*, 2003). Il est également envisageable d'intégrer des techniques statistiques comme le *supertagging* (Clark et Curran, 2004) pour réduire l'ambiguité lexicale.
- Amélioration de l'adéquation de XDG sur divers *phénomènes linguistiques* encore peu pris en compte, notamment les unités polylexicales, la coordination et l'ellipse, etc, et traitement plus efficace des grammaires *extraites automatiquement* à partir de corpus.

5.2.5 Adéquation à GUST/GUP

Les formalismes GUST/GUP et XDG partagent de nombreux points communs :

1. Ils possèdent tous deux une *architecture multi-stratale*, distinguant explicitement plusieurs niveaux de représentation linguistique et les liant par des interfaces. Une nuance néanmoins : dans GUST/GUP, les interfaces n'existent qu'entre niveaux adjacents[5].

2. Les interfaces entre niveaux sont relationnelles en GUST/GUP et en XDG : la correspondance entre deux niveaux adjacents est définie comme une relation "m-to-n", où une même structure d'entrée peut avoir plusieurs réalisations possibles en sortie, et vice-versa. Ceci contraste avec l'approche *fonctionnelle* souvent utilisée en sémantique formelle classique, où la représentation de sortie est calculée déterministiquement à partir des entrées.

3. Utilisation dans les deux cas d'*arbres de dépendance* come représentation syntaxique.

[5]Ceci n'est pas entièrement exact. En effet, il est théoriquement possible d'intégrer des polarités d'interface pour des niveaux non adjacents... Mais l'introduction d'une telle fonctionnalité pose à nos yeux d'importants problèmes (doit-on par exemple en déduire que tous les niveaux de représentation seraient munis d'une triple ou quadruple polarité ? Et si oui, quid de la saturation de toutes ces polarités ?), et nous sommes sceptiques quant à la généralisation d'une telle procédure dans l'entièreté du formalisme.

4. Les deux formalismes sont essentiellement *déclaratifs*, i.e. les aspects grammaticaux proprement dits et les aspects procéduraux sont autant que possible séparés[6].

5. Il s'agit enfin dans les deux cas de grammaires à la fois *lexicalisées* et *factorisables* : les descriptions grammaticales sont attachées aux unités individuelles, mais il est également possible de définir des classes lexicales en vue de regrouper des unités possédant les même comportements linguistiques. Le lexique de GUST/GUP, prenant appui sur les travaux lexicologiques issus de la TST, est néammoins beaucoup plus développé que celui d'XDG.

   En particulier, XDG assume *a priori* une correspondance 1 :1 entre les unités des différents niveaux, i.e. à chaque sémantème correspond un et un seul lexème, et vice-versa - le nombre total de noeuds doit donc être identique dans chaque dimension. Ceci nous a initialement posé un problème majeur puisqu'il nous était alors impossible de prendre en compte des phénomènes aussi répandus que les mots composés, les prépositions régimes, les auxiliaires, les locutions, les collocations, etc. GUST/GUP permettant, lui, de modéliser facilement ces phénomènes, il nous fallait trouver un moyen de "briser" cette correspondance 1 :1.

   La solution que nous avons adoptée, inspirée de (Debusmann, 2004), se fonde sur la notion de *groupe lexical*, défini comme un *sous-graphe dépendanciel*. Pour chaque sémantème réalisé syntaxiquement par plusieurs lexèmes (par ex. un mot composé), nous ajoutons à la structure sémantique un certain nombre de noeuds "vides", qui seront gouvernés par un arc étiqueté *del* indiquant leur suppression au niveau sémantique. Au niveau syntaxique, la cohérence du groupe lexical sera assuré par l'utilisation d'un principe spécialisé[7].

5.3 Axiomatisation de GUST/GUP en système de contraintes

Il est possible d'*axiomatiser* GUST/GUP en un CSP, et par là de le traduire dans un grammaire XDG[8]. Une propriété importante des GUP est en effet leur **monotonicité** : pour un système de polarités P = {◍,○,●} muni de l'ordre ◍< ○< ●, nous avons en effet :

$$\forall x, y \in P,\ x.y \geq max(x, y) \tag{5.5}$$

En d'autres termes, le produit de deux polarités donne toujours une polarité d'un rang $\geq$ au rang le plus élevé des deux. L'unification des structures ne peut donc pas boucler à l'infini, et progresse inexorablement vers les polarités les plus élevées de l'échelle. Deux corrolaires cruciaux de cette propriété sont que (1) les structures peuvent être combinées dans n'importe quel ordre, et (2) l'analyse en GUP peut être traduite en un CSP, cfr. (Duchier et Thater, 1999).

L'intuition générale de notre axiomatisation est la suivante : on peut considérer que chaque objet insaturé (= de polarité ○) *induit une contrainte* réclamant sa saturation en polarité ●, car le formalisme GUP exige la saturation complète d'une structure pour que celle-ci soit considérée comme grammaticalement correcte.

L'opération d'analyse/synthèse de structures polarisées par unification correspond donc à un problème de satisfaction de contraintes dans lequel les contraintes majeures spécifient que chaque objet (noeud, arc, grammène) doit être saturé dans sa dimension. Pour rappel, chaque objet GUP - noeud, arc, grammène - possède une double polarité : les objets sémantique une double polarité $(p_{\mathcal{G}_{sem}}, p_{\mathcal{I}_{sem-synt}})$, et les objets syntaxiques une double polarité $(p_{\mathcal{I}_{sem-synt}}, p_{\mathcal{G}_{synt}})$, comme nous l'avons expliqué à la section 3.6.5.

[6]XDG est néammoins plus avancé que GUST/GUP sur ce point, puisqu'il est basé sur la *description* "model-theoretic" de multigraphes, tandis que GUST/GUP reste un formalisme d'unification, donc génératif.

[7]Nous discuterons plus amplement de cette technique dans les pages suivantes.

[8]Notons que GUST n'est pas le seul formalisme d'unification a avoir été traduit en XDG : les TAG (angl. *Tree Adjoining Grammars*) ont également être axiomatisés en XDG, voir (Debusmann *et al.*, 2004d).

Ceci nous donne donc 4 contraintes de base à respecter pour notre interface, chacune d'entre elles étant chargée de contraindre les objets de la structure à être saturés dans une polarité particulière. En pratique, la notion de "saturation" d'un objet dans une dimension sera assurée par une variable d'ensemble finis attachée à cet objet. Si le domaine de cette variable est fixé à $\{1\}$, l'objet a été correctement saturé. Sinon, nous le considérons comme insaturé.

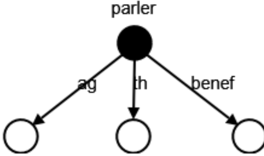
Un ensemble de "règles" permettront d'assurer cette saturation, et ces règles pourront à leur tour spécifier des contraintes propres à leur utilisation. Les règles d'interface permettent ainsi la saturation des polarités $p_{\mathcal{I}_{sem-synt}}$, mais exigent en échange le respect d'un ensemble de contraintes de *liaison* contraignant la "géométrie" des structures. Nous obtenons *in fine* un système complexe de contraintes imbriquées dont la satisfaction permet la saturation complète des structures, gage de leur grammaticalité.

Il nous sera naturellement impossible de présenter dans ce travail une axiomatisation intégralement spécifiée sur le plan formel[9] (via des formules de la logique du 1er ordre décrivant les modèles admissibles de la grammaire). Nous adoptons donc ici une présentation plus intuitive, mais néammoins rigoureusement étayée. L'étude détaillée des fondements mathématiques de notre axiomatisation est laissée à d'éventuels travaux ultérieurs.

Nous commencons par détailler l'axiomatisation de "cas simples", qui permettent une transcription directe en XDG sans l'ajout de structures ou contraintes supplémentaires, et poursuivons ensuite l'analyse de cas plus compliqués, qui ont nécessité un important travail de "traduction" de notre part, et abouti à la création de nouvelles structures de traits.

5.3.1 Cas basiques

Valence obligatoire : Analysons la règle illustrée à la figure suivante :



Informellement, cette règle signifie que : "pour réussir à saturer le sémantème ‹parler›, il faut également saturer ses trois dépendants, respectivement étiquetés par les rôles *ag*, *th* et *benef*". Elle spécifie donc une valence obligatoire portant sur le sémantème ‹parler› :

$$\forall n \in Nodes : n.gsem.name =' parler' \Rightarrow [|n.gsem.out.ag| = \{1\} \wedge \\ |n.gsem.out.th| = \{1\} \wedge |n.gsem.out.benef| = \{1\}] \quad (5.6)$$

Transcrit en UL, ceci nous donne[10]

```
defentry {
dim gsem {        name: "parler"
                  out: {ag! th! benef!}}}
```

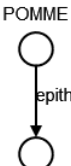
Evidemment, la règle signifie également que tout noeud, quel qu'il soit, peut reçevoir un nombre quelconque d'arcs entrants labellisés *ag*, *th* ou *benef* :

[9] Il nous faudrait pour cela présenter préalablement toute la formalisation de XDG, forte de 40 pages dans (Debusmann, 2006). De plus, au vu du nombre de nouvelles contraintes et variables ajoutées dans notre implémentation, une telle formalisation nécessiterait - au bas mot - plusieurs dizaines de pages et quelques mois de travail supplémentaires, ce qui est malheureusement très au dessus de nos capacités.

[10] Petite précision concernant la notation utilisée : le sigle " !" désigne une valence obligatoire i.e. $\in \{1\}$, le sigle " ?" une valence optionnelle $\in \{0, 1\}$ et le sigle "*" une valence $\{0... + \infty\}$

$$\forall n \in Nodes : |n.gsem.in.ag| = \{0... + \infty\} \wedge$$
$$|n.gsem.in.th| = \{0... + \infty\} \wedge |n.gsem.in.benef| = \{0... + \infty\} \quad (5.7)$$

Valence optionnelle : Il existe également des règles spécifiant des actants optionnels :



Cette règle signifie que le lexème “POMME” peut optionnellement recevoir un dépendant *epith* (un adjectif, par exemple). Elle spécifie donc une valence optionnelle portant sur “POMME”.

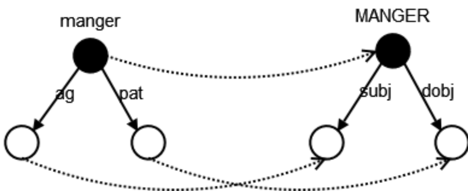
$$\forall n \in Nodes : n.gsynt.name =' POMME' \Rightarrow$$
$$|n.gsynt.out.epith| = \{0... + \infty\} \quad (5.8)$$

Ceci nous donne en UL :

```
defentry {
dim gsynt {      name: "POMME"
                 out: {epith*}}}
```

Interface : La figure ci-dessous présente une règle d'interface $\mathcal{I}_{sem-synt}$. Nous pouvons y distinguer deux contraintes :

1. Le sémantème ‹manger› est traduit au niveau syntaxique par le lexème “MANGER” ;
2. Les arguments sémantiques *ag* et *pat* sont traduits par la diathèse *ag* → *subj* et *pat* → *dobj*.



Nous pouvons donc spécifier la contrainte suivante (dans le sens de la génération) :

$$\forall n \in Nodes : [n.gsem.name =' manger' \wedge |n.gsem.out.ag| = \{1\} \wedge$$
$$|n.gsem.out.pat| = \{1\}] \Rightarrow [n.gsynt.name =' MANGER' \wedge$$
$$|n.gsynt.out.subj| = \{1\} \wedge |n.gsynt.out.dobj| = \{1\} \wedge$$
$$n.gsem.daughters.ag = n.gsynt.daughters.subj \wedge$$
$$n.gsem.daughters.pat = n.gsynt.daughters.dobj] \quad (5.9)$$

Nous pouvons transcrire cette contrainte en UL :

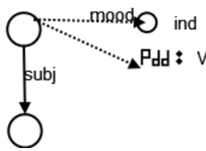
```
defentry {
dim gsem {       name:"manger"
                 out: {ag! pat!}}
dim gsynt {      name: "MANGER"
                 out: {subj! dobj!}}
dim isemsynt {   subcats: {ag:{subj} pat:{dobj}}}}
```

Notons que les liens d'interface sont traduits dans notre axiomatisation par une relation d'égalité[11]..

Evidemment, la plupart des règles GUST ne peuvent se traduire aussi facilement, et nécessitent la création de structures de traits particulières pour les modéliser, ainsi que l'ajout de nouveaux principes pour les traiter. Les trois sections suivantes discutent de l'axiomatisation de trois types de règles : les règles sagittales, les règles d'accord et les règles d'interface.

5.3.2 Règles sagittales

Analysons la règle illustrée à la figure ci-dessous, qui indique d'un verbe au mode indicatif peut reçevoir un sujet. Cette règle fait intervenir deux conditions (une partie du discours et un trait particulier) qu'il n'est pas possible de transcrire directement en XDG.



Nous avons alors créé un nouveau type de structure, dénotée `sagit`, et constitué de deux traits : `sagitConditions` et `sagitModifications` :

`sagitConditions` décrit un ensemble de préconditions (listées à la table 5.1) qui doivent être satisfaites pour que la règle puisse opérer ;

`sagitModifications` décrit la contrainte elle-même (dont les traits sont listés à la table 5.2).

Tab. 5.1: Traits de la structure `sagitConditions`

Nom du trait	Description
`gramsSagit`	Ensemble de grammènes
`posSagit`	Partie du discours

Tab. 5.2: Traits de la structure `sagitModifications`

Nom du trait	Description
`in`	Valence entrante du noeud
`out`	Valence sortante du noeud

Ainsi, pour reprendre l'exemple, la contrainte sera transcrite en UL de la manière suivante :

```
sagit:{{sagitConds: {    gramsSagit:        {[mood ind]}
                         posSagit:          {v}}
        sagitModifs:{    out:               {subj}}}}
```

Le typage de cette nouvelle structure en XDG est reproduit ci-dessous :

Listing 5.2 – Typage de la structure `sagit`

```
deftype "gsynt.sagitConditions" {
        gramsSagit:set(tuple("gsynt.grams" "gsynt.gramvalues"))
        posSagit: set("gsynt.pos")}
```

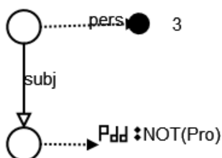
[11]Plus précisément, il s'agit en XDG d'un contrainte d'inclusion : il est exigé que la fonction syntaxique entre les deux noeuds soit incluse dans un ensemble donné, en général constitué d'un seul élément.

```
deftype "gsynt.sagitModifications" {
        in:  valency      ("gsynt.label")
        out: valency      ("gsynt.label")}

deftype "gsynt.sagit"    {
        set ({  sagitConds: "gsynt.sagitConditions"
                sagitModifs: "gsynt.sagitModifications"})}}
```

Bien sûr, le traitement de cette nouvelle structure exige l'ajout d'une nouvelle contrainte assurant son application effective : nous avons implémenté à cet effet un nouveau principe dénommé `GUSTSagittalConstraints`, qui a été intégré au XDK. Ce principe n'est appliqué qu'au niveau de $\mathcal{G}_{synt}$, car celui-ci n'a selon nous pas d'utilité réelle au niveau de $\mathcal{G}_{sem}$.

5.3.3 Règles d'accord

Les règles d'accord fonctionnent selon le même schéma que les règles sagittales. Soit la règle suivante, assurant qu'un sujet non pronominal induise toujours un verbe à la troisième personne :



De nouveau, nous pouvons distinguer d'une part un ensemble de conditions à respecter pour que la contrainte puisse s'appliquer, et la contrainte elle-même. Les tables 5.3 et 5.4 en détaillent les traits, et le listing 5.3 reproduit le typage XDG de cette structure.

TAB. 5.3: Traits de la structure `agrsConditions`

Nom du trait	Description
`gramsAgrs`	Ensemble de grammènes
`outAgrs`	Arcs sortants
`inAgrs`	Arcs entrants
`posAgrs`	Partie du discours
`notPosAgrs`	Exclusion de parties du discours
`outPosAgrs`	Partie du discours d'un noeud dépendant
`inPosAgrs`	Partie du discours d'un noeud gouverneur

TAB. 5.4: Traits de la structure `agrsModifications`

Nom du trait	Description
`grams`	Grammènes à polarité
`unsatGrams`	Grammènes à instancier
`inGrams`	Grammènes d'un noeud gouverneur
`outGrams`	Grammènes d'un noeud dépendant

Pour reprendre l'exemple fourni, la contrainte sera transcrite en UL comme ceci :

```
agrs:{{ agrsConds:{      notPosAgrs:      {pro}
                         inAgrs:          {subj}}
        agrsModifs:{     inGrams:         {subj:{[pers  trois]}}}}}
```

Listing 5.3 – Typage de la structure `agrs`

```
deftype "gsynt.agrsConditions" {
        gramsAgrs:set(tuple("gsynt.grams" "gsynt.gramvalues"))
        outAgrs: valency("gsynt.label")
        inAgrs: valency("gsynt.label")
        posAgrs: set("gsynt.pos")
        notPosAgrs: set("gsynt.pos")
        outPosAgrs: map ("gsynt.label" set("gsynt.pos"))
        inPosAgrs: map ("gsynt.label" set("gsynt.pos"))}

deftype "gsynt.agrsModifications" {
        inGrams: map ("gsynt.label"
                set(tuple("gsynt.grams" "gsynt.gramvalues")))
        outGrams: map ("gsynt.label"
                set(tuple("gsynt.grams" "gsynt.gramvalues")))
        grams: set(tuple("gsynt.grams" "gsynt.gramvalues"))
        unsatGrams: set("gsynt.grams")}

deftype "gsynt.agrs"     {
        set ({  sagitConds: "gsynt.agrsConditions"
                sagitModifs: "gsynt.agrsModifications"})}
```

Le traitement de cette nouvelle structure exige l'ajout d'une nouvelle contrainte assurant son application effective : nous avons implémenté à cet effet un nouveau principe dénommé `GUSTAgreementConstraints`, qui a été intégré au XDK, et appliqué au niveau de $\mathcal{G}_{synt}$.

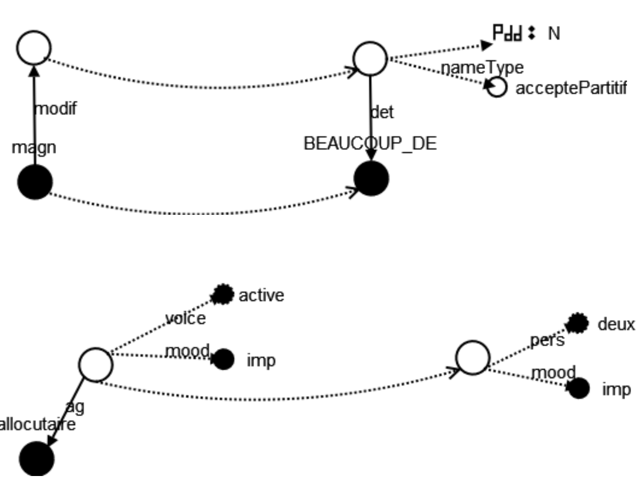
5.3.4 Règles d'interface

Axiomatiser les règles d'interface $\mathcal{I}_{sem-synt}$ a été très laborieux, étant donné la variété des constructions possibles, et l'exigence du correspondance 1 :1 entre niveaux (cfr section 5.2.5).

Chacune des cinq sous-sections suivantes est consacrée à un aspect particulier de l'interface sémantique-syntaxe que nous avons transcrite en XDG.

Préconditions : En analysant les deux figures ci-dessous, nous pouvons remarquer que ces règles ne peuvent s'appliquer que si certaines *préconditions* précises sont satisfaites :

- Pour la 1re figure, il est ainsi exigé que l'équivalent syntaxique de l'argument de ‹`magn`› soit un nom, et qu'il accepte l'article partitif (il désigne donc une "quantité massive").
- Pour la 2e figure, la précondition porte sur l'argument sémantique *ag* du verbe à mettre à l'impératif : il est exigé que celui-ci possède le label ‹`allocutaire`›.



Pour arriver à intégrer ces particularités dans XDG, il nous est donc nécessaire d'utiliser une structure de traits permettant d'exprimer ces préconditions, que nous dénommons `link.preconditions`.

TAB. 5.5: Traits de la structure `link.preconditions`

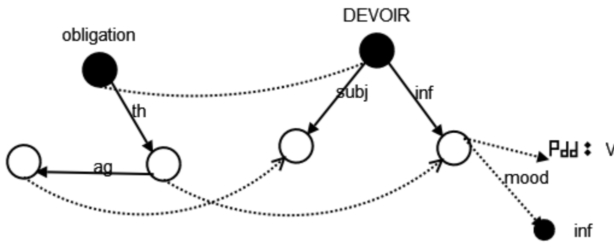
Nom du trait	Description
`semOut`	Arcs sortants sémantiques
`semIn`	Arcs sortants sémantiques
`semOutLabels`	Labels de noeuds dépendants
`semInLabels`	Labels de noeuds gouverneurs
`syntGrams`	Grammènes syntaxiques
`syntInGrams`	Grammènes de noeuds gouverneurs
`syntOutGrams`	Grammènes de noeuds dépendants
`syntPOS`	Parties du discours
`syntInPOS`	Partie du discours de noeuds gouverneurs
`syntOutPOS`	Partie du discours de noeuds dépendants

Ainsi, les préconditions des deux règles illustrées ci-dessus peuvent être respectivement exprimées par les deux extraits de code UL suivants :

```
link:{{ preconditions:{ syntInPOS:       {det:{N}}
                        syntInGrams:     {det:{[nametype acceptepartitif]}}}}}
```

```
link:{{ preconditions:{ semOutLabels:    {ag:{allocutaire}}}}}
```

Saturation sémantique : Observons à présent la partie sémantique de la règle ci-dessous . Nous remarquons que les polarités $p_{\mathcal{I}_{sem-synt}}$ des arcs *th* et *ag* sont saturées. L'un est directement issu du noeud principal, et l'autre est issu d'un noeud dépendant du noeud principal.



Il sera nécessaire d'exprimer dans le formalisme XDG quels sont précisément les arcs sémantiques saturés par une structure particulière : `link.sem`.

TAB. 5.6: Traits de la structure `link.sem`

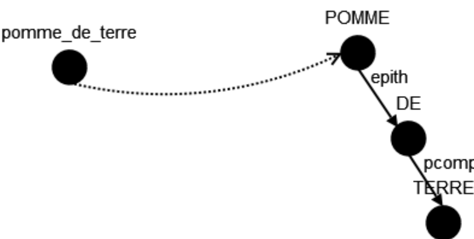
Nom du trait	Description
`semIn`	Arcs entrants sémantiques
`semOut`	Arcs sortants sémantiques
`semGrams`	Grammènes sémantiques
`semOutOut`	Arcs sortants d'un noeud dépendant
`semInIn`	Arcs entrants d'un noeud gouverneur
`semInOut`	Arcs sortants d'un noeud gouverneur
`semOutIn`	Arcs entrants d'un noeud dépendant
`semInGrams`	Grammènes d'un noeud gouverneur
`semOutGrams`	Grammènes d'un noeud dépendant

Il nous faudra évidemment une contrainte assurant que, pour une opération de génération aboutissant à une solution, tous les arcs de la représentation sémantique sont saturés. Cette contrainte est appelée `GUSTSemEdgeConstraints` et intégrée au XDK. Sans l'ajout de cette contrainte, il deviendrait possible de générer une représentation syntaxique en "laissant de côté" une portion de la représentation sémantique, ce qui est inacceptable.

La saturation sémantique de la règle donnée en exemple peut ainsi être transcrite en :

```
link:{{ sem:{    semOut:        {th}
                 semOutOut:     {[th ag]}}}}
```

Saturation syntaxique : La même observation peut se faire pour la partie syntaxique. Ainsi, dans la figure ci-dessous, les arcs *prep* et *pcomp* sont saturés par la règle, l'un étant directement issu du noeud principal, et l'autre indirectement.



La structure `link.synt` exprime l'ensemble des saturations syntaxiques possibles :

Tab. 5.7: Traits de la structure `link.synt`

Nom du trait	Description
`syntIn`	Arcs entrants syntaxiques
`syntOut`	Arcs sortants syntaxiques
`syntGrams`	Grammènes
`syntInLabels`	Labels de noeuds gouverneurs
`syntOutLabels`	Labels de noeuds gouvernés
`syntOutOut`	Arcs sortants d'un noeud dépendant
`syntInIn`	Arcs entrants d'un noeud gouverneur
`syntInOut`	Arcs sortants d'un noeud gouverneur
`syntOutIn`	Arcs entrants d'un noeud dépendant
`syntOutOutLabels`	Label d'un noeud dépendant d'un noeud dépendant
`syntInOutLabels`	Label d'un noeud dépendant d'un noeud gouverneur
`syntOutGrams`	Grammènes d'un noeud dépendant
`syntInGrams`	Grammènes d'un noeud gouverneur
`syntOutOutGrams`	Grammènes d'un noeuds dépendant d'un noeud dépendant
`syntOutOutOut`	Arcs sortants d'un noeud dépendant d'un noeud dépendant
`group`	Labels de noeuds sém. vides à générer autour du noeud

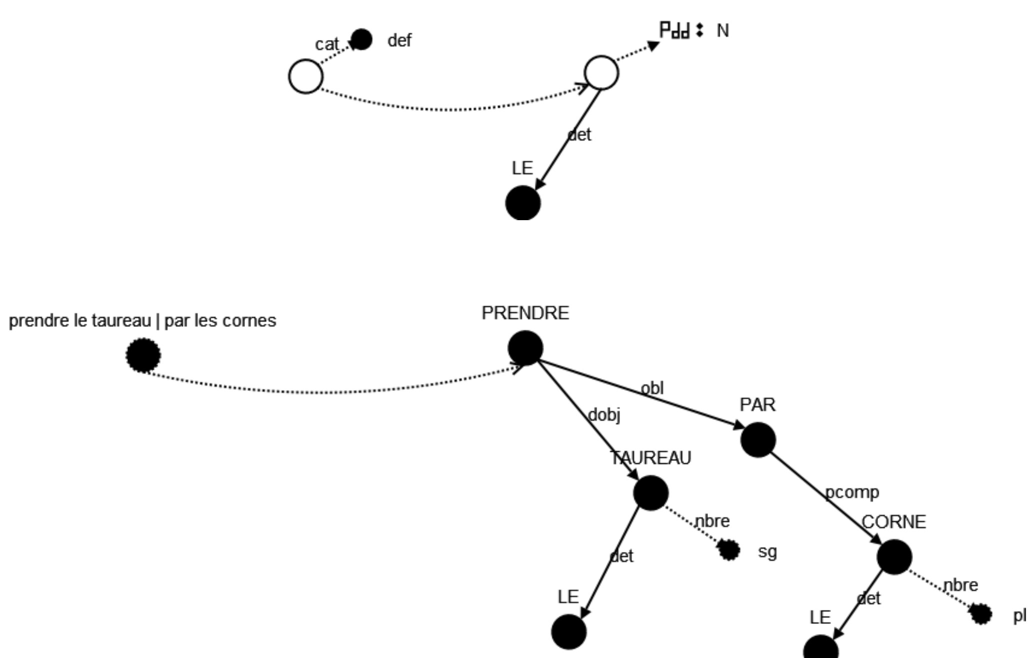
Une contrainte additionnelle, `GUSTSyntEdgeConstraints`, est intégrée à XDK qui permet d'assurer que tous les arcs syntaxiques sont bien saturés.

La saturation syntaxique de la règle illustrée peut donc être transcrite en :

```
link:{{ synt:{   syntOut:       {epith}
                 syntOutOut:    {[epith pcomp]}}}}
```

Génération de noeuds sémantiquement vides : Un problème délicat surgit en observant les règles illustrées aux deux figures suivantes : il s'agit de la génération, au niveau syntaxique,

de noeuds absents de la sémantique. Ainsi, générer la locution ‹prendre le taureau par les cornes› nécessite la génération des lexèmes "le", "taureau", "par", "le" et "corne".



Il s'agit là d'un problème difficile à résoudre, pour la raison mentionnée à la section 5.2.5. Néammoins, nous avons réussi, après diverses tentatives, à implémenter une solution viable : pour chaque sémantème réalisé syntaxiquement par plusieurs lexèmes (par ex. un mot composé), nous ajoutons à la structure sémantique un certain nombre de noeuds "vides", qui reçevront au niveau sémantique un arc étiqueté *del* indiquant leur suppression au niveau sémantique.

Le nombre de noeuds vides ajoutés est calculé à partir de la génération "de taille maximale". Pour la locution ‹Prendre le taureau par les cornes› par ex., nous insérons donc 5 noeuds vides.

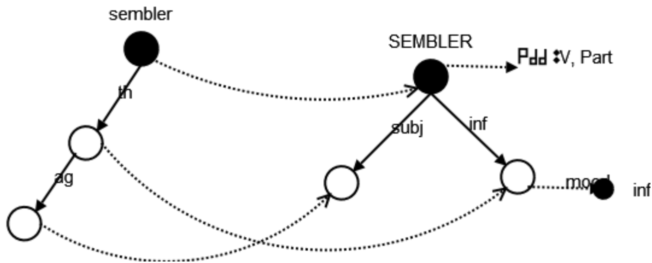
Au niveau syntaxique, la cohérence du groupe lexical sera assuré par l'application d'un principe, dénommé `GUSTEmptyNodesConstraints`. Celui-ci fait usage du trait `group` indiquant les identificateurs des lexèmes composant le sous-graphe représentant le groupe lexical.

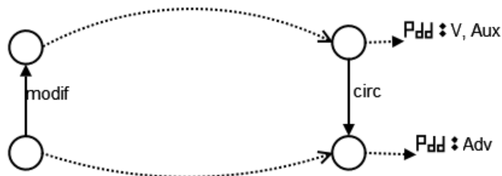
Ainsi, les deux règles instancient respectivement le trait `group` de la manière suivante :

```
link:{{ synt:{  group:{ after:{[le taureau par le corne]}}}}}
```

```
link:{{ synt:{  group:{ before:{[le]}}}}}
```

Liaison : Le dernier aspect qu'il nous faudra traiter porte sur la liaison ("*linking*") proprement dite, c'est-à-dire la manière dont la configuration "géométrique" des noeuds va se modifier en passant d'une dimension à l'autre. Les deux figures ci-dessous en illustrent deux exemples.





Ces “distorsions” vont être exprimées par la structure de traits `linking`. Il ne nous a pas été nécessaire d’écrire de nouveaux principes pour manipuler cette structure : en effet, XDK intègre déjà par défaut de nombreux principes de liaison. Nous en avons utilisé 5 : `LinkingDaughterEnd`, `LinkingBelowStartEnd`, `LinkingMotherEnd`, `LinkingAboveStartEnd` et `LinkingSisters`. Par manque de place, nous ne pouvons détailler les spécifications de chacun de ces principes, le lecteur intéressé pourra se référer au manuel de XDK : (Debusmann et Duchier, 2005).

Tab. 5.8: Traits de la structure `isemsynt.linking`

Nom du trait	Description
`subcats`	équivaut à l’argument `End` du principe `LinkingDaughterEnd`
`subcats_start`	équivaut à l’argument `Start` du principe `LinkingBelowStartEnd`
`subcats_end`	équivaut à l’argument `End` du principe `LinkingBelowStartEnd`
`mod`	équivaut à l’argument `End` du principe `LinkingMotherEnd`
`mod_start`	équivaut à l’argument `Start` du principe `LinkingAboveStartEnd`
`mod_end`	équivaut à l’argument `End` du principe `LinkingAboveStartEnd`
`sisters`	équivaut à l’argument `Which` du principe `LinkingSisters`
`reverseSubcats_start`	idem que `subcats_start`, mais sens inversé
`reverseSubcats_end`	idem que `subcats_end`, mais sens inversé

Les contraintes de liaison des règles illustrées sont donc respectivement traduites par les structures suivantes :

```
link:{{ linking:{        subcats: {th:{inf}}
                         reverseSubcats_start: {subj:{th}}
                         reverseSubcats_end: {subj:{ag}}}}}
```

```
link:{{ linking:{        mod: {modif:{circ}}}}}
```

Nous avons à présent terminé d’examiner les différents aspects des règles d’interface à transcrire dans le formalisme XDG. Le traitement global de la structure `link` est assuré par le principe `GUSTLinkingConstraints`. Celui-ci vérifie les préconditions de chaque règle, sature des arcs sémantiques et syntaxiques, génère les noeuds vides et règle les paramètres de liaison. Lorsque plusieurs règles alternatives existent pour assurer une correspondance, `GUSTLinkingConstraints` opère une distribution sur ces règles et examine chacune d’elles.

Le listing 5.4 reproduit le typage complet de la structure `link`, qui résume donc notre axiomatisation des règles d’interface de GUST/GUP en XDG.

Listing 5.4 – Typage de la structure `link`

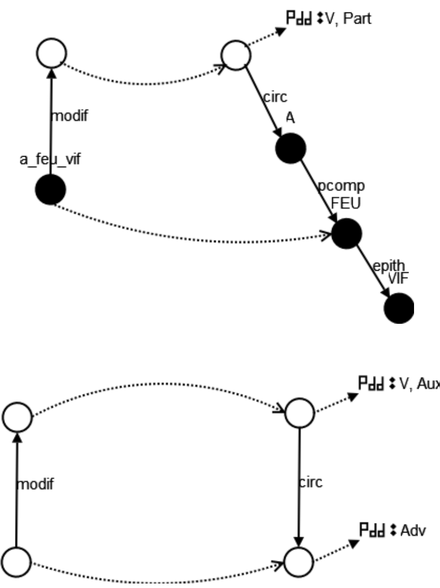
```
deftype "isemsynt.preconditions" {
        semOut:valency ("gsem.label")
        semIn:valency ("gsem.label")
        semOutLabels:map("gsem.label" iset("gsem.id"))
        semInLabels:map("gsem.label" iset("gsem.id"))
        syntGrams: set(tuple("gsynt.grams" "gsynt.gramvalues"))
        syntPOS: set("gsynt.pos")
        syntInPOS: map("gsynt.label" set("gsynt.pos"))
        syntOutPOS: map("gsynt.label" set("gsynt.pos"))
```

```
deftype "isemsynt.sem" {
        semIn:valency ("gsem.label")
        semOut:valency ("gsem.label")
        semGrams:set(tuple("gsem.grams" "gsem.gramvalues"))
        semOutOut: set(tuple (valency("gsem.label")
                  valency("gsem.label")))
        semInIn: set(tuple (valency("gsem.label")
                  valency("gsem.label")))
        semInOut: set(tuple (valency("gsem.label")
                  valency("gsem.label")))
        semOutIn: set(tuple (valency("gsem.label")
                  valency("gsem.label")))
        semInGrams:map("gsem.label"
                  set(tuple("gsem.grams" "gsem.gramvalues")))
        semOutGrams:map("gsem.label"
                  set(tuple("gsem.grams" "gsem.gramvalues"))) }

deftype "isemsynt.synt" {
        syntIn:valency ("gsynt.label")
        syntOut:valency ("gsynt.label")
        syntGrams:set(tuple("gsynt.grams" "gsynt.gramvalues"))
        syntInLabels:map("gsynt.label" iset("gsynt.id"))
        syntOutLabels:map("gsynt.label" iset("gsynt.id"))
        syntInIn:set(tuple (valency("gsynt.label")
                  valency("gsynt.label")))
        syntInOut:set(tuple (valency("gsynt.label")
                  valency("gsynt.label")))
        syntOutIn:set(tuple (valency("gsynt.label")
                  valency("gsynt.label")))
        syntOutOut: set(tuple (valency("gsynt.label")
                  valency("gsynt.label")))
        syntOutOutLabels:  set(tuple(valency("gsynt.label")
                 map ("gsynt.label"iset("gsynt.id"))))
        syntInOutLabels: set(tuple(valency("gsynt.label")
                 map ("gsynt.label"iset("gsynt.id"))))
        syntOutGrams:map ("gsynt.label"
                  set(tuple("gsynt.grams" "gsynt.gramvalues")))
        syntInGrams:map ("gsynt.label"
                  set(tuple("gsynt.grams" "gsynt.gramvalues")))
        syntOutOutGrams: set(tuple(valency("gsynt.label")
                 map ("gsynt.label"
                          set(tuple("gsynt.grams" "gsynt.gramvalues")))))
        syntOutOutOut: set(tuple (valency("gsynt.label")
                  valency("gsynt.label") valency("gsynt.label")))
        group:{before:set(list("gsynt.id")) after:set(list("gsynt.id"))}}

deftype "isemsynt.linking" {
        subcats:map("gsem.label" set("gsynt.label"))
        subcats_start:map("gsem.label" set("gsynt.label"))
        subcats_end:map("gsem.label" set("gsynt.label"))
        mod: map("gsem.label" set("gsynt.label"))
        mod_start: map("gsem.label" set("gsynt.label"))
        mod_end: map("gsem.label" set("gsynt.label"))
        sisters:set("gsem.label")
        reverseSubcats_start:map("gsynt.label" set("gsem.label"))
        reverseSubcats_end:map("gsynt.label" set("gsem.label"))}
```

```
deftype "isemsynt.link" {
        preconditions:"isemsynt.preconditions"
        sem:"isemsynt.sem"
        synt:"isemsynt.synt"
        linking:"isemsynt.linking"})
        lexicalised: {true false}}
```

5.3.5 Classes et unités lexicales

Un dernier point mérite d'être souligné. Les deux règles d'interface ci-dessous se distinguent en ceci que la 1re s'applique à une unité (poly)lexicale particulière qu'elle sature, tandis que la 2e s'applique à un ensemble d'unités, sans en saturer aucune.



Cette distinction est intégrée dans notre implémentation par un système de *classes XDG*. Ces classses sont alors *hérités* par les unités lexicales. Par exemple, dans le listing 5.1, le mot "`poivre`" hérite de quatre classes : "`N`" (la classe des noms), ainsi que trois classes génériques "`Root-gsem`", "`Root-gsynt`", et "`Root-isemsynt`", qui reprennent un ensemble de règles appliquables à tout type d'unités.

Bien sûr, les classes peuvent être combinées et s'hériter l'une l'autre. Notre implémentation fait un grand usage de cette fonctionnalité et gènère une véritable *hiérarchie* complexe de classes.

5.3.6 Résumé

Pour conclure ce chapitre, rappelons l'intuition générale présidant à cette axiomatisation :

1. Les contraintes de base de notre grammaire assurent que tous les objets des différents niveaux soient intégralement *saturés*, c'est-à-dire que les objets sémantiques possèdent tous une polarité $(p_{\mathcal{G}_{sem}}, p_{\mathcal{I}_{sem-synt}}) = (\bullet, \bullet)$, et que les objets syntaxiques possèdent tous également une polarité $(p_{\mathcal{G}_{synt}}, p_{\mathcal{I}_{sem-synt}}) = (\bullet, \bullet)$;
2. Pour opérer cette saturation, un ensemble de *règles* sont spécifiées : règles sagittales, règles d'accord, règles d'interface ;
3. Ces règles peuvent être intégréee à une *classe* ou une *unité lexicale* particulière ;
4. Celles-ci ne sont opérantes que sous certaines *conditions* précises, et elles peuvent également ajouter de *nouvelles contraintes* propres.

Chapitre 6

Implémentation de l'Interface Sémantique-Syntaxe

Ce chapitre est consacré à la discussion de notre implémentation de $\mathcal{I}_{sem-synt}$. Nous l'avons déjà mentionné à plusieurs reprises, cette implémentation s'est concrétisée en deux phases :

1. Conception d'un **compilateur** de grammaires GUST/GUP en grammaires XDG, baptisé `auGUSTe`. Il a été programmé en Python et "pèse" environ 6.000 lignes de code ;
2. Adaptation de XDG, via l'ajout de **structures de traits** et de **contraintes**, pour l'ajuster à nos besoins. En pratique, il s'agit là d'environ 2.200 lignes de code en Oz.

Dans la section suivante, nous détaillons la méthodologie générale que nous avons adoptée pour ce travail. Nous présentons ensuite dans la section 6.2 notre compilateur `auGUSTe`. Nous y détaillons son cahier des charges, son architecture, et les étapes principales de son fonctionnement. Enfin, la section 6.3 discutera des contraintes que nous avons intégrées au XDK.

6.1 Méthodologie

Le travail que nous avons effectué tout au long de l'année en vue d'implémenter notre interface a suivi une méthodologie bien précise, que nous résumons en 9 étapes :

1. Comprendre : Nous avons bien sûr commencé par *défricher* le problème que nous avions à résoudre. Nous avons donc étudié quantité de travaux relatifs à GUST/GUP, à l'approche par contraintes appliquée au TALN, et bien sûr aux interfaces sémantique-syntaxe.

2. Analyser : Une fois les tenants et aboutissants plus ou moins maîtrisés, nous sommes passés à l'analyse proprement dite. Nous avons ainsi élaboré un *cahier des charges* complet de notre implémentation, comprenant notamment une définition formelle des structures en entrée et en sortie, l'élaboration d'une liste de phénomènes linguistiques que l'implémentation devra traiter, et la mise au point d'une procédure de validation expérimentale.

3. Concevoir : Nous nous sommes alors interrogés sur la meilleure manière de construire notre interface. Nous devions réaliser un choix sur deux plans :

1. Au niveau du type d'*algorithme* à implémenter, nous avions le choix entre une approche "classique" basée sur l'unification de structures, ou une approche par contraintes ;
2. Dans les deux cas se posait également la question de l'éventuelle *réutilisation* de systèmes existants ; plusieurs logiciels ayant déjà été développés dans ce domaine.

Après mûre réflexion, nous avons finalement choisi de travailler via l'approche par contraintes et de réutiliser un programme open-source déjà existant : *Extensible Dependency Kit*. Ce logiciel est fondé sur une approche théorique assez proche de la nôtre (modélisation multi-stratale, utilisation d'arbre de dépendance, etc.), et convenait donc idéalement à notre entreprise.

Le cahier des charges de notre implémentation a de ce fait été substantiellement modifié : son rôle ne consiste plus à assurer de bout en bout la synthèse de représentations sémantiques en arbres syntaxiques, mais devient un *compilateur* GUST/GUP ⇒ XDG.

4. Axiomatiser : Après avoir choisi d'utiliser XDK au sein de notre implémentation, il nous était bien sûr nécessaire d'axiomatiser GUST/GUP en CSP, ce qui ne fut pas directement évident puisque GUST/GUP est au départ un formalisme d'unification (i.e. génératif).

5. Modulariser : Nous avons ensuite travaillé sur l'architecture de notre implémentation, que nous voulions modulaire. Nous avons donc défini un ensemble de modules, classes, méthodes, ayant chacun un rôle bien défini. `auGUSTe` est ainsi composé de 16 modules.

6. Spécifier : Après avoir conçu la charpente du compilateur, nous sommes passés aux spécifications. Plutôt qu'un développement "en cascade" qui était très mal adapté à notre cas, nous avons préféré travailler selon un processus évolutif, "spiralé" (Sommerville, 2004, p. 73) : nous commencions à spécifier et implémenter le "prototype" d'un module, l'intégrions dans l'ensemble, analysions son fonctionnement et ses dysfonctionnements, et nous efforcions alors de le modifier et l'enrichir jusqu'à aboutir à une version suffisamment stable.

Etant donné qu'il était impossible, au vu du caractère relativement "expérimental" du problème à résoudre, de spécifier dès le départ l'ensemble du compilateur, cette approche fondée sur l'extension progressive d'un prototype s'est révélée à nos yeux plutôt efficace.

7. Implémenter : Comme indiqué ci-dessus, notre implémentation s'est réalisée de manière cyclique, en enrichissant graduellement un prototype jusqu'à arriver au résultat escompté. Nous avons dû faire face à de nombreux obstacles, dont un sérieux problème de performance qui nous a amené à réviser environ 1/3 de notre implémentation.

En parallèle avec l'implémentation du compilateur, nous avons également dû procéder à divers aménagements du programme XDK pour l'adapter à nos besoins. Nous avons ainsi notamment ajouté au système un ensemble de 8 contraintes spécialement conçues pour notre interface.

8. Modéliser : Un formalisme est inutile s'il n'est pas utilisé pour y exprimer des données linguistiques. Nous nous sommes alors attachés à élaborer les modélisations, sous formes de GUP, de phénomènes linguistiques intéressants, dont un échantillon a été présenté au chapitre 4.

9. Expérimenter : Enfin, nous avons procédé à la validation expérimentale de notre système, par l'application d'un mini-corpus centré sur le vocabulaire culinaire. Cette étape fut évidemment aussi l'occasion, outre l'obtention des résultats finaux, de déceler de nouvelles imperfections dans notre logiciel, et d'y remédier au mieux. Le chapitre 7 sera entièrement consacré à ce sujet.

6.2 `auGUSTe` : Compilateur XDG ⇒ GUST

6.2.1 Cahier des charges

L'objectif de notre compilateur consiste à **traduire** une grammaire écrite sous le formalisme GUST/GUP en une grammaire écrite sous le formalisme XDG, de telle sorte que l'utilisation de

cette dernière au sein de XDK pour générer des représentations sémantiques donne précisément les **mêmes résultats** que cette même opération effectuée via le formalisme initial GUST/GUP, i.e. l'unification de structures polarisées.

Notre compilateur respectera donc son cahier des charges si les grammaires d'entrée et de sortie sont équivalentes du point de vue des résultats qu'elles produisent : les arbres syntaxiques générés à partir d'un graphe sémantique quelconque doivent être strictement *identiques*.

Il est évident que cette exigence a pour conséquence que l'adéquation de notre cahier des charges ne peut être mesurée qu'*indirectement*, en vérifiant à partir d'une batterie de tests si les résultats produits correspondent à ceux attendus.

Il est néammoins possible de poser le problème de manière différente : l'unification de structures polarisées et la résolution d'un CSP ont en effet en commun la propriété d'être des opérations **monotones**, i.e. le résultat final dépend uniquement de la nature des règles ou des contraintes utilisées et non de l'ordre dans lequel celles-ci sont appliquées.

Nous pouvons déduire de cette observation cruciale le principe suivant : si, pour chaque structure GUP compilée, nous arrivons à extraire rigoureusement l'ensemble des contraintes qui permettent de caractériser ses modèles, nous pouvons alors être formellement assurés que la grammaire GUST/GUP et la grammaire XDG seront équivalentes.

C'est la raison pour laquelle nous nous sommes donc penchés au chapitre précédent sur la classification des différents types de structures GUP, et avons étudié une axiomatisation CSP pour chacune d'elles.

6.2.2 Entrées et sorties

Détaillons à présent les formes concrètes que prendront les entrées et sorties de notre compilateur. La grammaire GUST en entrée pourra être fournie sous deux formats alternatifs :

1. Un ensemble d'**images vectorielles** crées à l'aide du logiciel Dia[1] (logiciel libre disponible sur Unix et Windows) et enregistrées au format XML ;
2. Un **fichier texte** contenant l'ensemble des règles GUST/GUP.

Images vectorielles

Chaque règle est représentée graphiquement à l'aide de noeuds, d'arcs, de grammènes, muni de leurs labels et polarités. Nous avons créé à cet effet une nouvelle “feuille” au sein de Dia regroupant les objets utilisés[2]. La figure 6.1 donne un exemple.

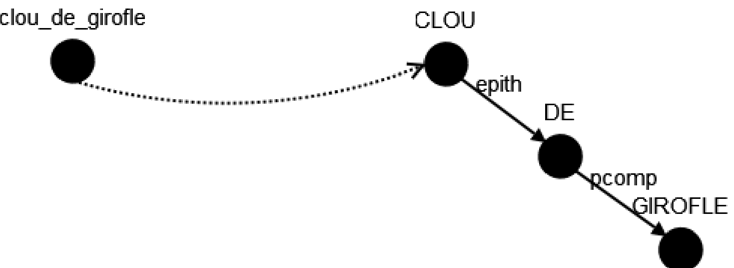


Fig. 6.1 – Règle graphique enregistrée dans le fichier `clou-de-girofle_isemsynt.dia`

[1]Plus d'infos sur `http ://www.gnome.org/projects/dia`

[2]Signalons au passage que la quasi totalité des figures illustrées dans ce mémoire ont été réalisées à l'aide de cette même feuille.

A chaque règle correspond un fichier possédant une extension `.dia` et dont le format interne est XML. Les règles de $\mathcal{G}_{sem}$ sont distinguées par un suffixe "_gsem", les règles de $\mathcal{G}_{synt}$ par un suffixe "_gsynt", et les règles de $\mathcal{I}_{sem-synt}$ par un suffixe "_isemsynt". L'ensemble des règles doit être regroupé dans un unique répertoire.

De par son caractère intuitif, ce format graphique est particulièrement adapté pour le développement manuel de grammaires. C'est d'ailleurs sous ce format que nous avons conçu l'ensemble de la grammaire utilisée pour la validation expérimentale (cfr. chapitre suivant).

Il est néammoins évident que celui-ci n'est pas directement manipulable pour la compilation. Nous avons donc implémenté un analyseur XML assurant la conversion automatique de ce format graphique au format textuel.

Format textuel

Ce format est sans nul doute moins lisible que le précédent, mais est bien plus facilement manipulable. Le listing ci-dessous présente la forme prise par la règle de la figure 6.1 dans ce format. Comme on peut le constater, chaque règle est constituée d'un ensemble d'objets - noeuds, arcs ou grammènes - référencés par un nombre entier. Chacun de ces objets possède un certain nombre d'attributs : label, polarité, source et cible, etc.

Ce format s'inspire notamment de celui-ci décrit dans (Lareau, 2004).

Nous avons développé une syntaxe formelle de type BNF pour ce format, qui est présentée à la table 6.1. Nous utilisons pour le traitement un système d'analyse lexicale et syntaxique de type LR, où les règles BNF sont exprimées par des fonctions Python.

Listing 6.1 – Equivalent textuel de la règle 6.1

```
rule241 {       1:node.gsem
                2:node.gsynt
                3:node.gsynt
                4:node.gsynt
                5:edge.gsynt
                6:edge.gsynt
                7:edge.isemsynt
                label(1) = 'clou_de_girofle'
                p(1) = +
                linearPos(1) = 1
                label(2) = 'CLOU'
                p(2) = +
                linearPos(2) = 2
                label(3) = 'DE'
                p(3) = +
                linearPos(3) = 3
                label(4) = 'GIROFLE'
                p(4) = +
                linearPos(4) = 4
                label(5) = 'prep'
                p(5) = +
                target(5) = 3
                source(5) = 2
                label(6) = 'pcomp'
                p(6) = +
                target(6) = 4
                source(6) = 3
                p(7) = +
                target(7) = 2
                source(7) = 1}
```

TAB. 6.1: Syntaxe BNF du format défini pour GUST/GUP

```
Grammar ::= Dimension | Grammar Dimension
Dimension ::= Dim_id { DimensionContents }
DimensionContents ::= Rule | DimensionContents Rule
Rule ::= Rule_id { RuleContents }
RuleContents ::= RuleHeader RuleBody
RuleHeader ::= ObjDeclaration| RuleHeader ObjDeclaration
ObjDeclaration ::= Obj_id : ( node | edge | gram ) . Dim_id
RuleBody ::= ObjSpecification | RuleBody ObjSpecification
ObjSpecification ::= Feat ( Obj_id ) = Value
Value ::= Obj_id | Quotedstring | Pol | List
Feat ::= string
Dim_id ::= string
Rule_id ::= string
Obj_id ::= string
Quotedstring ::= " string "
Pol ::= + | -
List ::= { ListContents }
ListContents ::= Value | ListContents Value
```

6.2.3 Architecture

Nous présentons à présent l'architecture de notre compilateur. Le schéma UML complet est illustré à la figure 6.2. Voici une brève description des rôles joués de nos différents modules :

auGUSTE : Module "racine" de notre implémentation, comprenant le menu principal à partir duquel l'utilisateur peut choisir l'opération qu'il désire effectuer.

ConvertDia2GUST : Module chargé de convertir les fichiers Dia contenus dans un répertoire en un unique fichier textuel représentant la grammaire GUST. Ce module fait appel aux trois modules ci-dessous pour le traitement des règles de $\mathcal{G}_{sem}$, $\mathcal{G}_{synt}$ et $\mathcal{I}_{sem-synt}$, respectivement.

Dia2GUST_gsem, Dia2GUST_gsynt, Dia2GUST_isemsynt : Modules de conversion d'un fichier graphique Dia au format XML en une règle grammaticale.

PUGLexer et PUGParser : Modules d'*analyse lexicale* et *syntaxique* (respectivement) de type LR du fichier contenant la grammaire. Les règles BNF sont exprimées par des fonctions Python. Ces modules importent les modules externes `lex` et `yacc`[3].

Compiler : Module assurant la *compilation* proprement dite de la grammaire. Il reçoit en entrée une structure de type dictionnaire contenant la grammaire analysée par `PUGParser`, et en sort une autre structure contenant l'ensemble des entrées XDG.

Constants et Utils : Modules regroupant respectivement les constantes utilisées dans `auGUSTe`, et un ensemble de fonctions générales (parcours de graphes, extraction d'attributs, etc.) partagées par les autres modules.

ClassProcessing : Module assurant le traitement des *classes* dans les grammaires GUST/-GUP. Par le terme "classe", nous entendons les règles grammaticales qui ne sont pas directement liées à une unité linguistique particulière, mais expriment des règles "générales" s'appliquant à une série d'unités vérifiant certaines conditions. Ainsi, la règle d'accord exprimant qu'un verbe fini exige un sujet est une classe s'appliquant à tous les verbes.

LinkingProcessing : Module assurant le traitement des *règles de correspondance*. Il analyse donc, pour chaque règle, ses préconditions, les arcs saturés au niveau sémantique et au niveau syntaxique, les noeuds sémantiquement vides à générer pour la réalisation syntaxique, et l'ensemble des contraintes de liaison.

[3] Il s'agit de deux librairies inspirées de l'utilitaire `Bison`, sous licence LGPL.

`NodeProcessing` : Module assurant le traitement des *unités lexicales*. Il analyse les valences, attributs, PdD, classes lexicales, etc. de chaque unité.

`XDGTextGeneration` : Module chargé de l'écriture proprement dite de la grammaire au format UL de XDG. Il prend en entrée une structure de données contenant l'ensemble des unités XDG à générer, et produit en sortie un texte qu'il suffit alors d'écrire dans un fichier.

`CompileTests` : Module "racine" pour la compilation de la batterie de tests. Il extrait les contraintes structurales permettant de caractériser le graphe sémantique, et les intègre dans XDK, en les associant à une phrase incluse dans le fichier de test.

`GUSTEmptyNodesGenerator` : Module implémenté en Oz chargé de déterminer, à partir de la grammaire XDG, quelles sont les unités (potentiellement) polylexicales, i.e. les éléments de $\mathcal{G}_{sem}$ qui devront être accompagnées de noeuds sémantiquement vides lors de la confection de la batterie de tests. Lorsque plusieurs réalisations syntaxiques sont possibles, le module se base sur l'expansion maximale envisageable.

6.2.4 Etapes principales

Il nous est évidemment impossible de présenter le fonctionnement détaillé de notre implémentation, celle-ci étant constituée de près de 6.000 lignes de code ($\cong$ une centaine de pages au format A4). Nous présentons néammoins les étapes majeures de notre compilateur ci-dessous (attention, le pseudo-code s'étend sur deux pages).

Algorithme 1 : Pseudo-code résumant les étapes principales de la compilation

Data : A grammar is given in input, in the form of a set of graphical Dia files or in a textual GUST/GUP grammar file.
Result : The GUST/GUP grammar has been compiled in a XDG Grammar.

```
/* Step 0 : Dia ⇒ GUST Conversion                                    */
if user has not already converted the Dia grammar files to GUST format then
    for File in grammar directory do
        Add Convert(File) to GUSTGrammarFile;
    end
end

/* Step 1 : Grammar Parsing                                          */
Grammar = Parse GUSTGrammarFile ;

/* Step 2 : Compilation                                              */
Verify well-formedness of Grammar ;
SemClasses = Extract G_sem classes from Grammar ;
SemUnits = Extract G_sem lexical units from Grammar ;
SemEdges = Extract G_sem edge labels from Grammar ;

SyntClasses = Extract G_synt classes from Grammar ;
SyntUnits = Extract G_synt lexical units from Grammar ;
SyntEdges = Extract G_synt edge labels from Grammar ;

LexicalLinking = Extract I_sem-synt lexical linking from Grammar ;
ClassLinking = Extract I_sem-synt classes linking from Grammar ;

XDGUnits = Merge (SemUnits, SyntUnits, LexicalLinking) ;
XDGClasses = Merge (SemClasses,SyntClasses, ClassLinking) ;

/* Step 3 : XDG Grammar Generation                                   */
XDGGrammar = GeneratePreamble +
             GenerateXDGClasses(XDGClasses) +
             GenerateXDGUnits (XDGUnits);
(... )
```

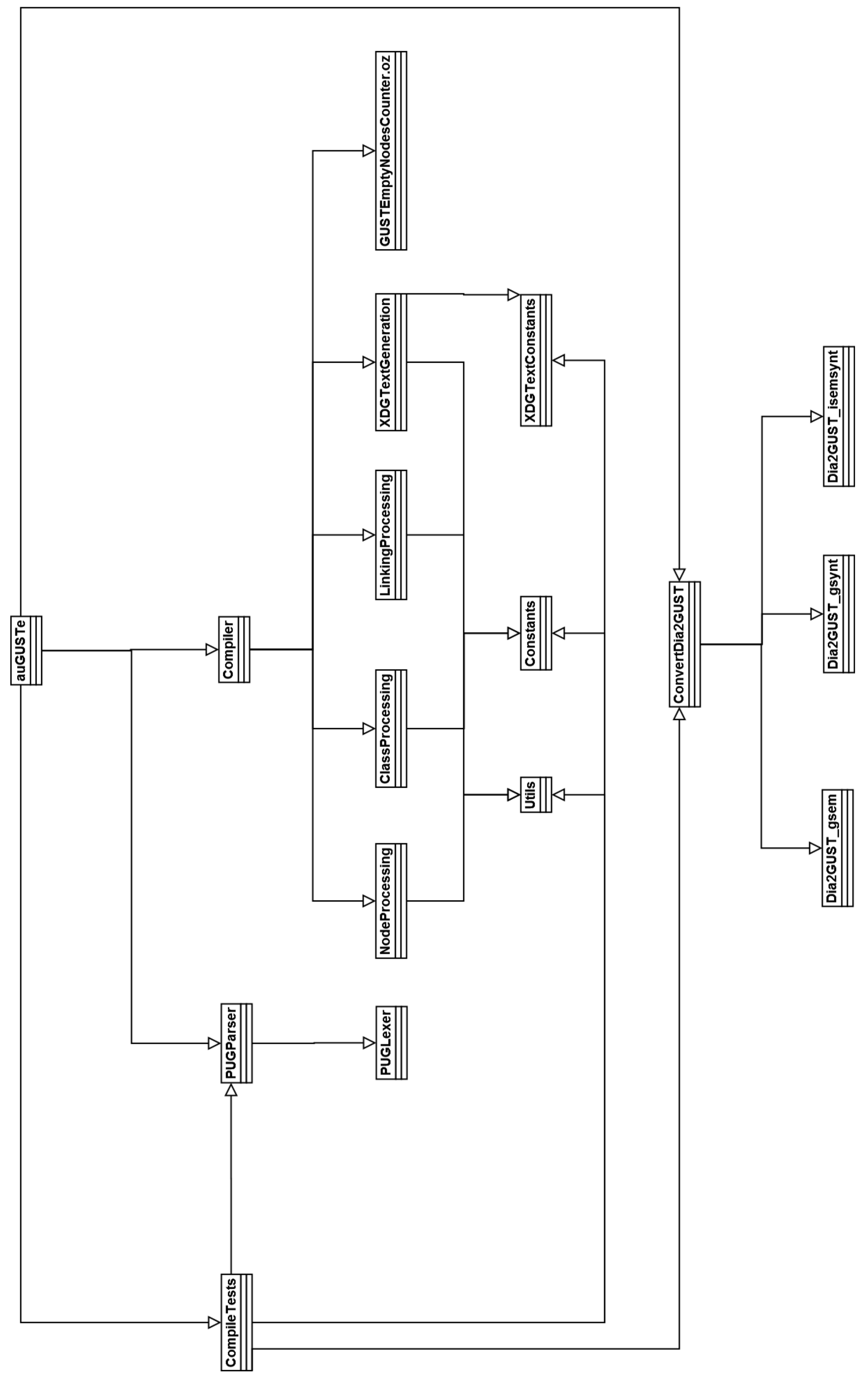


FIG. 6.2 – Schéma UML des différents modules d'auGUSTe

```
(...)
Write XDGGrammar to file ;

/* Step 4 : Final operations                                        */
Generate and Compile Oz file GUSTAlterGrams ;
Generate and Compile Oz file GUSTEmptyNodesCounter ;
```

Voici également le pseudo-code représentant les étapes de la compilation de la batterie de tests (conversion de fichiers Dia en contraintes structurales intégrées au XDK) :

Algorithme 2 : Pseudo-code résumant les étapes de traitement de la batterie de tests

```
Data : A test suite is given in input, in the form of a set of graphical Dia files or in a textual
       GUST/GUP test file.
Result : The GUST/GUP test suite has been integrated in the XDK.

if user has not already compiled the GUST Grammar then
    Compile Grammar;
end

if user has not already converted the Dia test files to GUST format then
    for File in test directory do
        Add Convert(File) to GUSTTestFile;
    end
end

TestSuite = Parse GUSTTestFile ;

for Test in TestSuite do
    XDGConstraint = Extract XDG structural constraint from Test ;
    InputSentence = Extract input sentence from Test ;
    Add XDGConstraint to the GUSTGeneration principle ;
    Recompile GUSTGeneration ;
    Add InputSentence to the GUSTExample file ;
end
```

6.2.5 Complexité

Nous avons réalisé une petite analyse de la complexité de notre compilateur. La grosse majorité des fonctions implémentées possèdent une complexité en $O(r \times o)$, où r désigne le nombre de règles, et o le nombre moyen d'objets dans chaque règle. Une petite minorité de fonctions travaillent en $O(r^2)$, ce qui reste évidemment très bon.

La vitesse de compilation est d'ailleurs très rapide (moins d'une seconde pour traiter environ 900 règles), l'opération la plus lourde étant en fait l'écriture même du fichier XDG.

6.3 Contraintes XDG

6.3.1 Généralités

En plus du compilateur `auGUSTe`, nous avons également implémenté une série de 8 nouvelles contraintes au sein de XDG, totalisant environ 2.000 lignes de code ($\cong$ 30 pages au format A4). Ces contraintes ont bien sûr été développées sous Mozart/Oz, et font en particulier un usage intensif de la librairie de contraintes sur les ensembles finis (Schulte et Smolka, 1999).

L'intuition sous-jacente à ces contraintes ayant déjà été présentée au chapitre précédent, nous nous contentons ici de résumer brièvement leur rôle :

`GUSTGeneration` : Constraint la structure sémantique pour qu'elle "colle" à l'un des exemples choisis dans la batterie de tests [4] ;

`GUSTInitialConstraints` : "Initialisation" de certaines contraintes générales ;

`GUSTLinkingConstraints` : Application des règles d'interface :
(1) Saturation des arcs sémantiques et syntaxiques concernés par la règle ;
(2) Application des contraintes de liaison ;
(3) "Génération" de nouveaux noeuds pour les unités polylexicales ;

`GUSTEmptyNodesConstraints` : Les noeuds sémantiquement vides de la structure sont contraints à être utilisés au sein d'une construction polylexicale, ou à être supprimés ;

`GUSTSagittalConstraints` : Les valences `'in'` et `'out'` des noeuds sont contraintes aux valences spécifiées par les règles sagittales respectant les préconditions ;

`GUSTSemEdgeConstraints` : Contraint les arcs de $\mathcal{G}_{sem}$ à être saturés par une règle d'interface ;

`GUSTSyntEdgeConstraints` : Contraint les arcs de $\mathcal{G}_{synt}$ à être saturés par une règle d'interface ;

`GUSTAgreementConstraints` : Application des règles d'accord.

Une petite remarque s'impose concernant `GUSTGeneration` : on peut en effet s'étonner du fait que la batterie de tests doive être préalablement intégrée au système des contraintes de XDK. Pourquoi ne peut-on fournir directement les tests en entrée, sans compiler de contraintes ?

En fait, le problème provient du fait que XDK est, pour le moment, un logiciel essentiellement axé sur l'analyse et non la génération ; et que les seuls "inputs" possibles dans l'état actuel de XDK sont des séquences de mots. Il nous aurait été assurément fastidieux de reprogrammer le coeur du logiciel pour qu'il puisse également accepter des graphes sémantiques, nous nous sommes donc orientés vers une solution plus simple : nous fournissons bien au XDK une séquence de mots en entrée, mais nous ajoutons également une contrainte supplémentaire contraignant le graphe sémantique à une structure bien particulière.

Signalons également que toutes les contraintes ci-dessus possèdent une priorité particulière exprimant leur importance respective et donc l'ordre dans lesquelles elles vont s'appliquer :

1. `GUSTGeneration` : priorité 170
2. `GUSTInitialConstraints` : priorité 160
3. `GUSTLinkingConstraints` : priorité 150
4. `GUSTEmptyNodesConstraints` : priorité 145
5. `GUSTSagittalConstraints` : priorité 140
6. `GUSTSemEdgeConstraints` et `GUSTSyntEdgeConstraints` : priorité 0
7. `GUSTAgreementConstraints` : priorité -100

6.3.2 Complexité et Performances

Nous l'avons déjà indiqué au chapitre précédent, il est démontré dans (Debusmann et Smolka, 2006) qu'en toute généralité, l'analyse XDG est un problème NP-complet, par réduction à partir de `SAT`. Néammoins, le comportement pratique de notre grammaire (ainsi que d'autres) est expérimentalement bien meilleur que le "worst-case" théorique.

Nous avons passé de très nombreuses heures à rendre notre interface aussi efficace que possible, et ce par plusieurs moyens :

1. Minimiser autant que possible le *nombre* de variables (entendez ici variables de contraintes sur des ensembles finis) nécessaires à notre interface, tout en gardant un formalisme aussi expressif et élégant que possible. Il a bien sûr parfois fallu opérer des compromis entre

[4] Cette contrainte est donc modifiée chaque fois que la batterie de tests est regénérée

sophistication de la description linguistique et efficacité algorithmique.

2. Renforcer au maximum la *propagation* de nos contraintes. Nous voulions éviter à tout prix l'usage de la distribution, opération très coûteuse en temps de calcul et en espace mémoire. Néammoins, il existe certaines configurations qui ne peuvent être résolues sans distribution.

3. Définir un ensemble d'*heuristiques* de conception de grammaires GUST/GUP susceptibles d'améliorer les performances. Nous avons en effet remarqué que certaines règles grammaticales posaient, de par leur configuration "géométrique", des problèmes calculatoires importants, et qu'il était souvent possible de les réécrire d'une manière plus respectueuse des limitations techniques de notre implémentation, tout en gardant exactement la même puissance descriptive au niveau linguistique.

Lors des divers tests que nous avons effectués sur notre implémentation (cfr. chapitre 7), nous avons observé une durée moyenne de calcul (définie comme le temps requis pour générer l'ensemble des réalisations syntaxiques d'un graphe sémantique donné) comprise entre 250 ms. et 10 sec., ceci sur un portable à 1.7 GHz, 256 Mb de RAM. Quant au temps requis pour trouver une seule solution, il est compris entre 250 ms. et 5 sec.

Ce résultat n'a évidemment rien de déshonorant pour un "prototype" tel que le nôtre, mais nous avouons ne pas être pas entièrement satisfaits de ces résultats. Malgré la quantité importante de travail que nous avons investie à optimiser notre implémentation, nous n'avons pas réussi à faire descendre la vitesse moyenne de traitement en dessous de la barre de la seconde. Nous avons longuement analysé les causes de cette relative "lenteur" de traitement, dont trois sont clairement ressorties :

1. En premier lieu, le *logiciel XDK*, aussi intéressant et novateur soit-il, n'est pas encore totalement au point au niveau de l'efficacité de traitement. Comme nous l'avons mentionné à la section 5.2.4, il reste encore de nombreux axes de recherche à explorer pour son amélioration notamment au niveau technique : utilisation d'une "guidance probabiliste", *supertagging*, ajout de contraintes globales, réimplémentation sous *GeCode*, etc..

2. Ensuite, nous avons observé que l'*ajout de noeuds vides* ralentit significativement la vitesse de calcul. L'ajout de noeuds vides nous est nécessaire pour briser la correspondance 1 :1 entre unités sémantiques et syntaxiques, puisque XDG ne propose pas actuellement de support réellement adéquat pour des constructions linguistiques telles que les locutions, les mots composés, etc. Néammoins, cette opération a des conséquences non négligeables sur le temps de calcul, puisqu'elle augmente naturellement le nombre de variables du CSP.

3. Enfin, la troisième source de lenteur est la *distribution*. Nous avons cherché à minimiser autant que possible le nombre d'étapes de distribution[5], mais celles-ci sont parfois indispensable. Ainsi, lorsque plusieurs réalisations syntaxiques distinctes sont possibles pour un même sémantème (par ex. des synonymes), la distribution est inévitable. Et lorsqu'une même phrase combine plusieurs phénomènes de ce genre, le nombre d'embranchements enfle évidemment très rapidement.

Une optimisation plus poussée de notre interface passe donc par la résolution de ces trois problèmes. Aucun de ceux-ci ne constitue un obstacle insurmontable[6], et il est donc permis d'être optimiste quant à l'amélioration à terme des performances de notre implémentation.

[5] La plupart des exemples de notre batterie de tests nécessitent ainsi moins de 5 embranchements

[6] Plusieurs travaux de recherche sur XDG se sont déjà attaqués à ces questions et certaines solutions concrètes ont été proposées, cfr par exemple (Pelizzoni et das Gracas Volpe Nunes, 2005)

6.3.3 Algorithmes

Nous présentons ci-dessous le pseudo-code de 5 de nos contraintes :

Algorithme 3 : Pseudo-code pour la contrainte `GUSTLinkingConstraints`

```
Input : Nodes : the set of Nodes
Result : All the linking rules satisfying the preconditions have been applied :
(1) Saturation of the appropriate semantic and syntactic edges
(2) Posting of the linking constraints
(3) "Generation" of new nodes for multiword expressions

for Node in Nodes do
    RulesSatisfyingPreconds = {} ;
    for Rule in Node.link do
        if Verify (Rule.preconditions) = OK then
            Add Rule to RulesSatisfyingPreconds ;
        end
    end
    if ∀ Rule1, Rule2 ∈ RulesSatisfyingPreconds :
    Rule1.sem ∩ Rule2.sem = ∅^a then
        RulesToApply = RulesSatisfyingPreconds ;
    else
        Distribute the rules which are in conflict ;
        RulesToApply = the distributed rules ;
    end
    for Rule in RulesToApply do
        Saturate the edges in Rule.sem ;
        Saturate the edges in Rule.synt ;
        Apply the linking constraints in Rule.linking ;
        Use the group feature to "generate" new nodes for polylexical units ;
    end
end
```

[a] Il y aura incompatibilité entre deux règles d'interface lorsque (1) les préconditions `link.preconditions` des deux règles sont satisfaites *et* (2) l'intersection des saturations sémantiques `link.sem` des deux règles constitue un ensemble non vide, i.e si l'on applique une règle, on ne peut appliquer l'autre et vice-versa.

Algorithme 4 : Pseudo-code pour la contrainte `GUSTEmptyNodesConstraints`

```
Input : Nodes : set of Nodes
Result : All the semantically empty nodes of the structure are constrained to be either used in a
         multiword expression or deleted

for Node in Nodes do
    if Node is semantically empty then
        if one of the two nodes surrounding Node has a non-empty group feature then
            Node is constrained to one of the element in the group feature;
        else
            Node is deleted
        end
    end
end
```

Algorithme 5 : Pseudo-code pour la contrainte `GUSTSagittalConstraints`

```
Input : Nodes : set of Nodes
Result : The 'in' and 'out' valencies are constrained to the valencies specified in the sagittal rules
         satisfying the preconditions

Initialisation ;
for Node in Nodes do
    for Rule in Node.sagittalRules do
        if Verify (Rule.conditions) = OK then
            Rule.modifications.in ⊆ Node.in ;
            Rule.modifications.out ⊆ Node.out ;
        end
    end
end
for Node in Nodes do
    Node.in ⊆ lowerbound (Node.in) ;
    Node.out ⊆ lowerbound (Node.out) ;
end
```

Algorithme 6 : Pseudo-code pour la contrainte `GUSTSemEdgeConstraints`

```
Input : Node1, Node2 : two nodes, et LA : a semantic edge label
Result : Constrain the edge LA to be saturated by a linking rule

if LA is an edge between Node1 and Node2 then
    LA must be saturated by a linking rule attached to Node1 or Node2 ;
end
```

Algorithme 7 : Pseudo-code pour la contrainte `GUSTAgreementConstraints`

```
Input : Nodes : set of Nodes
Result : All the agreement rules satisfying the preconditions have been applied

Initialisation ;
for Node in Nodes do
    for Rule in Node.agrsRules do
        if Verify (Rule.conditions) = OK then
            Apply Rule.modifications;
        end
    end
end
```

6.4 Regards croisés sur GUST/GUP et XDG

Clôturons ce chapitre par quelques remarques générales à propos de GUST/GUP et XDG. Notre implémentation nous a en effet permis de mettre en lumière les forces et faiblesses de ces deux formalismes et de jeter des ponts entre ceux-ci.

En ce qui concerne GUST, la force principale de ce formalisme réside, selon nous, dans la qualité de sa modélisation linguistique. Comme S. Kahane l'indique : « Contrairement à certains modèles linguistiques comme TAG ou les grammaires catégorielles, GUST n'est pas né de l'adaptation d'un formalisme à une théorie linguistique. Au contraire, le formalisme a été développé à partir des données linguistiques et des propriétés de la langue. ». (Kahane, 2002, p. 73).

Comparativement à XDG, la description de certains phénomènes linguistiques y est donc plus

élaborée, et le lexique est bien plus conséquent - Ralph Debusmann, qui connait bien la TST et les travaux de S. Kahane l'indique d'ailleurs lui-même, cfr. (Debusmann, 2006, p. 42).

Les faiblesses de GUST/GUP sont, selon nous, essentiellement d'ordre formel et technique. Au niveau formel, deux considérations nous viennent à l'esprit :

- D'une part, GUST/GUP garde une perspective *"proof-theoretic"* sur la grammaire : on dérive une analyse en combinant un ensemble d'unités par l'application de règles de production (cfr. p. 41). Cette perspective reste selon nous problématique pour assurer le caractère déclaratif de la grammaire. Il existe actuellement de nombreux travaux de recherche axés sur la reformulation de formalismes *"proof-theoretic"* - notamment TAG : (Rogers, 1998) - en formalismes purement *"model-theoretic"*, et GUST pourrait s'en inspirer.
- D'autre part, le parallélisme (au sens de Jackendoff) de l'architecture GUST/GUP reste encore hypothétique : l'usage de polarité d'interfaces pourrait permettre en théorie l'interaction entre des niveaux non adjacents, mais il n'existe à l'heure actuelle aucune étude approfondie sur la question. (cfr. la note au bas de la page 74).

Quant aux aspects techniques , j'ai été notamment confronté aux problèmes suivants durant mon travail d'implémentation :

- Absence initiale de rôles thématiques au niveau prédicatif pour distinguer différents prédicats possibles, cfr. page 48 ;
- Absence initiale de grammènes au niveau prédicatif, avec pour conséquence une inflation du nombres de noeuds au niveau sémantique, cfr. 48 ;
- Difficultés pour spécifier certaines valences dans GUST : sauf erreur de notre part, il est ainsi impossible de spécifier aisément qu'un noeud accepte un nombre d'arcs rentrant ou sortants compris dans une certaine fourchette[7], ou qu'un classe lexicale exige une valence précise[8]. L'expression des valences en XDG nous semble par contraste bien plus claire et complète, voir (Debusmann et Duchier, 2005, p. 31).

En ce qui concerne XDG, les forces et faiblesses du formalisme nous semblent être l'exact inverse de celles de GUST/GUP : XDG est très poussé au niveau formel - quatre gros chapitres sont entièrement consacrés à la formalisation mathématique dans (Debusmann, 2006) - et technique - XDK est sans conteste un logiciel techniquement très réussi.

Néammoins, les modélisation linguistiques constituent selon nous clairement son talon d'Achille : ainsi l'hypothèse de correspondance 1 :1 entre les noeuds des différentes dimensions n'est jamais vérifiée dans la réalité langagière, et ceci même pour des phénomènes tout-à-fait basiques, comme les mots composés, les auxiliaires ou les prépositions régimes. (Pelizzoni et das Gracas Volpe Nunes, 2005) fournit d'ailleurs à ce titre un argumentaire très convaincant concernant la nécessité de remédier rapidement à ce problème dans XDG.

[7]Comment exprimer par ex. qu'un nom accepte 0 ou 1 arc sortant labellisé "det", mais jamais plus de 1 ?
[8]Comment exprimer par ex. qu'un verbe exige un (et un seul) sujet, en n'utilisant qu'une seule règle sagittale ?

Chapitre 7

Validation expérimentale

Ce dernier chapitre s'intéresse à la *validation expérimentale* de notre travail. Celle-ci a été réalisée par le biais d'une petite **grammaire** d'environ 300 mots appartenant au vocabulaire culinaire, et d'une **batterie de tests** de 50 graphes sémantiques.

Bien que cette étape ait constitué un part importante de notre travail, ce chapitre est relativement court, car il nous est impossible (pour des raisons évidentes de place) d'y détailler l'ensemble de nos résultats. Nous nous contentons donc de présenter ici notre méthodologie, notre grammaire, notre batterie de tests, et bien sûr les résultats globaux de l'expérience.

L'annexe B illustre le détail de la génération de 20 graphes sémantiques via notre interface.

7.1 Méthodologie

1. Nous avons commencé par feuilleter plusieurs livres de cuisine pour en extraire quelques centaines de mots, ainsi que diverses constructions grammaticales possibles. Notre objectif était bien sûr d'obtenir un *lexique* aussi riche et diversifié que possible pour le domaine du discours étudié, et qui pouvait donc être capable de construire de nombreuses phrases "simples" (i.e. sans coordination ni extraction), semblables à celles que l'on peut typiquement retrouver dans une recette de cuisine.

2. Après avoir établi cette liste, nous avons procédé à son *encodage* dans le formalisme GUST/-GUP. Nous avons à cet effet utilisé le logiciel Dia enrichi d'une "feuille" spécifique au formalisme GUST. Cet encodage, qui a nécessité de très nombreuses heures, a abouti à la constitution de plus de 900 règles[1] - et donc de 900 fichiers Dia.

3. Nous avons ensuite vérifié la *bonne formation* et la *cohérence* de notre grammaire en la compilant sous `auGUSTe`. De nombreuses corrections subséquentes furent nécessaires (dues tant à de simples fautes d'encodage qu'à des erreurs plus fondamentales liées aux modélisations linguistiques choisies) avant d'obtenir une grammaire conforme à nos exigences.

4. Une fois notre grammaire construite, nous avons procédé à la création de notre *batterie de tests*. Pour ce faire, nous avons fourni à une personne extérieure[2] le lexique que nous avions constitué, et lui avons demandé de construire 50 phrases avec celui-ci, en veillant bien à (1) n'utiliser que des mots et constructions permises par le lexique, et (2) diversifier au maximum la variété et complexité des phrases.

[1] En effet, un même mot nécessite en général au moins trois règles, issues de $\mathcal{G}_{sem}$, $\mathcal{G}_{synt}$, et $\mathcal{I}_{sem-synt}$.

[2] Il s'agissait en l'occurrence de mon frère, que je remerçie vivement pour ce travail.

5. Muni de ce mini-corpus de 50 phrases, nous avons alors encodé leur *représentation sémantique* sous GUST/GUP. Cette étape a donc résulté en un ensemble de 50 graphes sémantiques décrits dans des fichiers Dia.

6. A cette étape-ci de notre travail, toutes les ressources requises pour la validation expérimentale sont à notre disposition. Nous avons donc appliqué notre batterie de tests à notre interface sémantique-syntaxe, et en avons analysé les résultats.

7. Pour chaque arbre syntaxique généré, nous avons soigneusement examiné si celui-ci était bien *grammatical*, i.e. respectait les spécifications de notre grammaire. Pour les cas où la génération avait manifestement échoué, nous avons cherché à analyser les causes de cet échec. Lorsque celles-ci trouvaient leur source non pas dans notre interface elle-même, mais dans la grammaire que nous avions construite (par exemple à cause de règles incohérentes ou incorrectes), nous avons modifié notre grammaire en conséquence. Par contre, si la faute incombait à notre implémentation, nous comptabilisions le test comme ayant échoué[3].

7.2 Grammaire "culinaire"

Nous présentons ici les éléments grammaticaux et lexicaux que nous avons introduits dans notre grammaire et ont donné lieu à la constitution de quelques 900 règles GUST/GUP.

7.2.1 Éléments grammaticaux

Rôles thématiques : Nous avons ici repris *grosso modo* les listes de rôles thématiques présents dans des ouvrages de référence tels que (Jurafsky et Martin, 2000) :

ag	Agent du procès	*th*	Thème
pat	Patient du procès	*manner*	Complément de manière
benef	"Bénéficiaire" d'un procès	*time*	Cadre temporel
modif	Modifieur	*goal*	Complément de but
loc	Cadre spatial		

Fonctions syntaxiques : Nous les avons déjà détaillées dans le tableau 2.1. Elles sont très largement inspirées de (Tesnière, 1959). Les fonctions utilisées dans notre grammmaire sont :

subj	Sujet du verbe fini	*aux*	Complément de l'auxiliaire
dobj	Objet direct du verbe	*epith*	Epithète du nom
iobj	Objet oblique (=prépositionnel) du verbe	*acirc*	Complément circonstanciel de l'adjectif ou de l'adverbe
obl	Objet indirect du verbe	*det*	Déterminant du nom
circ	Complément circonstanciel du verbe	*pcomp*	Complément d'une préposition
attr	Attribut du sujet de la copule		

Accord en genre et en nombre : Les articles et adjectifs s'accordent en ‹genre› et en ‹nombre›, et les verbes s'accordent en ‹temps›, ‹personne› et en ‹nombre›.

[3]Pour déterminer la provenance de l'erreur, la procédure est relativement simple : il suffit d'examiner la grammaire, d'extraire de celle-ci l'ensemble des règles concernées par le problème (i.e. qui sont nécessaires à la saturation d'un objet de la structure en entrée), et d'analyser si leur combinaison "à la main" permet la saturation complète de la structure. Très souvent, la réponse saute directement aux yeux, avant même de réaliser la moindre opération : l'on s'aperçoit qu'on a de toute évidence "oublié" tel ou tel objet (par ex., on a omis d'inclure une fonction syntaxique, une partie du discours, un attribut indispensable). Si à la suite de cet examen, l'on est assuré que les règles fonctionnent parfaitement mais que l'implémentation ne donne aucune réponse (ou une réponse incorrecte), alors nous avons effectivement détecté un problème.

Conjugaison : Les verbes peuvent reçevoir trois modes (‹indicatif›, ‹impératif› et ‹infinitif›) et quatre temps[4] : ‹imparfait›, ‹passé composé›, ‹présent› et ‹futur›.

Verbe copule : Le verbe copule “`être`” a une rection particulière, puisqu'il demande un sujet et un attribut, qui peut être un adjectif (“`La pomme est rouge`”), un nom (“`Les borgnes sont rois`”), un adverbe (“`Ils sont ensemble`”) ou une préposition (“`Il est à l'ouest`”).

Participes passés et présents : les participes passés et présents ont la particularité de pouvoir fonctionner comme épithètes du nom : “`La soupe préparée par Pierre`”, “`Pierre préparant la soupe`”. Dans certains cas, le rôle *ag* du verbe est sous-entendu : “`Le plat disposé sur la table`” ; dans ce cas il est signalé comme étant ‹indéfini›.

Alternance actif/passif : les verbes s'expriment selon deux diathèses : ‹actif› et ‹passif›.

Négation : Les verbes peuvent être entourés de la forme “`ne ... pas`” exprimant la négation.

Expressions figées : Diverses expressions figées (“`clou de girofle`”, “`moulin à légumes`”, etc.) sont présentes dans la grammaire.

Collocations : Il existe également plusieurs collocations, exprimées selon le formalisme des fonctions lexicales (cfr. 2.2.4), principalement la fonction d'intensification $Magn$.

Compléments du nom : les expressions prépositionnelles telles que N “`de`” N (“`La soupe de Pierre`”) sont également admissibles.

7.2.2 Éléments lexicaux

Verbes :

N ajouter N à N	N eponger N	N plier N
N assaisonner N	N essayer de V_{inf}	N poser N
N badigeonner N	N étaler N	N prendre N de N
N battre N	N être Adj	N preparer N
N beurrer N	N évider N	N raffoler de N
N boire N	Il faut V_{inf}	N raper N
N bouillir	N fondre	N reduire
N chauffer	N frire	N refroidir
N commencer à V_{inf}	N frotter N	N remuer N
N continuer à V_{inf}	N hacher N	N retirer N de N
N couper N	N laisser V_{inf}	N revenir
N cuire	N laver N	N rissoler N
N déconseiller N à N	N manger N	N saler N
N déguster N	N mettre N	N saupoudrer N
N devoir V_{inf}	N mouiller N	N secouer N
N disposer N	N nettoyer N	N sembler V_{inf}
N dorer N	N melanger N	N servir N
N écraser N	N oter N	N s'ouvrir
N egoutter N	N ouvrir N	N tapisser N
N enfariner N	N peler N	N travailler sur N
N envelopper N	N percer N	N tremper N
N eplucher N	N piquer N	N verser N

[4]Bien sûr, seule la syntaxe est prise en compte, nous ne traitons pas la morphologie verbale.

Noms :

ail	goût	pinceau
amande	graine	plante
arête	gratinee	plat
assaisonner	grumeau	plat à four
basilic	hachoir	plume
beurre	haricot	poele
biere	herbe	poireau
bol	huile	pois chiche
boulette	ingredient	poisson
café	intérieur	poivre
calmar	jambon	pomme
carotte	jus	pomme de terre
casserole	lait	porto
céleri	laitue	pulpe
cerfeuil	lanière	puree
chapelure	lard	pyrex
chocolat	lardon	quartier
citron	laurier	recette
citrouille	legume	refrigerateur
clou de girofle	lendemain	repas
courgette	liquide	riz
couteau	Marc	rouleau
crevette	matin	salade
cuillerée	mayonnaise	sauce
cuisson	melange	saucisse
dessert	menthe	sautoir
eau	minute	saveur
ébullition	mixeur	sel
élément	moitié	soupe
entre-temps	morceau	spatule
epice	moulin à légumes	sucre
extérieur	muscade	volaille
farine	navet	viande
fenouil	oeuf	veille
feu	oignon	terrine
feuille	olive	tete d’ail
formation	partie	tomate
four	pâtes	thym
fourchette	patisserie	table
fromage	pepite au chocolat	tarte
gibier	perdrix	tarte à le/la/les N
gousse d’ail	persil	

Adjectifs :

aromatique	froid	petit
blanc	grand	possible
chaud	intéressant	principal
coloré	leger	provencal
delicieux	lourd	savoureux
effilé	median	succulent
environ	meilleur	tout
ferme	necessaire	traditionnel
frais	paysan	vert

Adverbes :

abondamment	de nouveau	moyennement
à l'avance	également	plus
à feu doux	encore	rapidement
à feu vif	entre-temps	seulement
avec enthousiasme	finement	tres
bien	légèrement	trop
complètement	longtemps	un peu

Compléments prépositionnels de lieu, temps, manière, but :

pendant N	dans N	pour N
jusque N	de façon à V	pour V
après N	dès N	sans N
avant N	durant N	selon N
avant de V	en N	sur N
avec N	en V	

Déterminants :

1. **Articles :** définis, indéfinis et partitifs : "le", "la", "les", "un", "des", "du", "de la".
2. **Adjectifs possessifs :** "mon", "ton", "son", "ma", "ta", "sa", etc.
3. **Numéraux :** "deux", "trois", etc.
4. **Quantiticateur :** "beaucoup de"

Pronoms :

1. **Pronoms personnels :** "vous", "tu", "on", "nous", "je", "elle", "elles", "ils"
2. **Pronoms réfléchis :** "me", "te", "se"

7.3 Corpus de validation

Nous détaillons à présent les 50 phrases qui ont servi de "corpus" de validation à notre interface. Chacune de ces phrases a été retranscrire à la main pour obtenir une représentation sémantique.

1. "Marc a ajouté du sel à la sauce"
2. "Marc ne doit pas trop laver la salade à l'eau"
3. "Beurrez le plat avec du beurre paysan avant de verser le mélange dans la poële"
4. "Vous servirez le poisson après la soupe"
5. "Dégustons une cuillerée de mayonnaise sans sucre"
6. "Nous ajouterons une gousse d'ail au riz pour le goût"
7. "Nous avons trempé la perdrix dans le porto un peu rapidement"
8. "Ma meilleure terrine a un goût froid"
9. "Nettoyez les plats avec de l'eau froide"
10. "Faites refroidir les pâtisseries au réfrigérateur avant de servir les cafés"
11. "Nettoyez également la poële"
12. "Ce riz provencal est savoureux"
13. "Faites dorer l'ail avec les amandes"
14. "Ne dégustez pas la soupe avec une spatule"
15. "Ajoutons une tête d'ail aux pommes de terre"

16. “Il planta sa fourchette dans la tomate”
17. “La cuisson du navet sera intéressante”
18. “Un oeuf est le principal ingrédient”
19. “Hachez abondamment le laurier”
20. “Vous rissolerez la viande à feu vif”
21. “Essaye d'égouter les légumes”
22. “Tapissons l'extérieur de la perdrix”
23. “Il versait le chocolat dans la citrouille”
24. “Nous pelions les pommes de terre avec un couteau”
25. “L'eau refroidissait sans le café”
26. “Le citron est un ingrédient traditionnel”
27. “Ils disposaient les lourds oignons dans les plats”
28. “Ce savoureux mélange refroidissait rapidement”
29. “Nous préparions un repas coloré”
30. “La meilleure bière est fraiche”
31. “Elle dégustaient nos saucisses avec enthousiasme”
32. “Elles égoutaient les pâtes”
33. “La saveur du sucre est légère”
34. “Il commencait à couper la courgette”
35. “Ne salez pas la sauce blanche”
36. “Ne servez pas la mayonnaise avec de la purée”
37. “Le meilleur beurre est provencal”
38. “Un grand calmar cuisait dans le four”
39. “Il faut mettre les courgettes dans le réfrigérateur”
40. “Les desserts sont également préparés par Pierre”
41. “Les terrines de Marc sont bien nettoyées”
42. “Les pommes de terre sont écrasées par Marc pour les servir pendant le repas”
43. “Marc raffole de fromage”
44. “Il essaye de préparer des haricots mélangés avec des pommes de terre”
45. “Marc semble essayer de disposer les ingrédients dans la poële”
46. “La soupe est bien chaude”
47. “Pierre doit faire bouillir l'eau à l'avance”
48. “Les carottes sont prises du moulin à légumes par Pierre pour essayer de préparer un plat pour le lendemain”
49. “Le gibier étalé par Pierre sur la table est enveloppé par de la sauce”
50. “Marc évide les crevettes avec un fourchette”

7.4 Résultats

7.4.1 Généralités

Les résultats de l'expérience sont très positifs. Sur les 50 graphes de notre batterie de tests, seuls *deux* ont abouti à une erreur dans la génération (i.e. le CSP a été déclaré insoluble). Pour les 48 autres, notre interface génère correctement le ou les arbres de dépendance exprimant la réalisation syntaxique de notre graphe prédicatif.

Bien sûr, lors de nos premiers tests, les résultats étaient bien plus médiocres, mais il s'est avéré que la grande majorité des erreurs provenaient non pas de notre interface, mais des spécifications de notre grammaire, qui n'avait pas pris en compte tel ou tel comportement linguistique.

Concevoir une grammaire est un travail long et difficile, quel que soit le formalisme, comme l'ont bien montré les projets de TALN portant sur l'élaboration de grammaires à large couverture.

Nous avons néammoins pu observer que le formalisme GUST/GUP, de par sa nature à la fois lexicalisée et articulée, offrait une grande souplesse dans la description. Il est en effet possible de spécifier un comportement syntaxique précis pour chaque unité lexicale (par ex. pour exprimer le régime de certains verbes), mais également de factoriser une grande partie des spécifications, par l'intégration de règles applicables à un grand nombre d'unités.

Les figures 7.1 et 7.2 illustrent deux exemples de génération réussies. Ces graphiques ont été générés automatiquement à partir de notre interface, qui offre plusieurs formats de sortie possibles, dont une sortie LATEXparticulièrement utile.

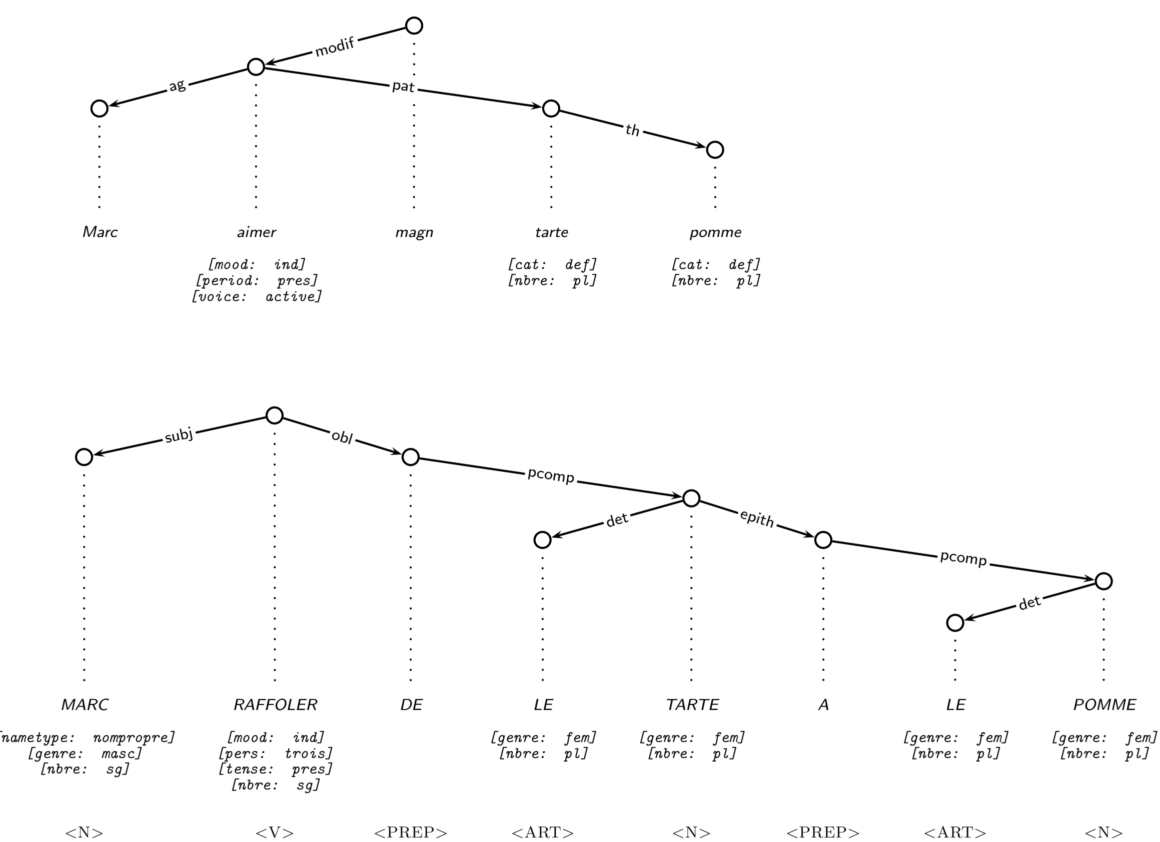


FIG. 7.1 – Exemple de génération automatique du graphe prédicatif (haut) en un arbre de dépendance (bas) via notre interface $\mathcal{I}_{sem-synt}$

Pour un graphe sémantique donnée, il peut exister *plusieurs* réalisations syntaxiques possibles[5]. Notre interface peut fonctionner de deux manières : soit elle s'arrête à la 1re solution, soit elle énumère l'ensemble des solutions. L'utilisateur peut choisir entre ces deux options via la GUI. L'outil *Oz Explorer* permet de visualiser la recherche des solutions en temps réel.

Notons pour terminer que nous n'avons pas abordé explicitement la question de la *surgénération*, i.e. est-il possible que notre grammaire accepte des structures agrammaticales ? Nous n'avons pas de résultats quantifiés sur cette question, et il serait assurément intéressant d'en obtenir. Néammoins, au vu de l'expérience engrangée au long des nombreuses heures passées à élaborer cette grammaire, nous sommes convaincus qu'une grammaire GUST/GUP bien conçue ne surgénère pas, ou seulement à titre anecdotique. Nous n'avons en tout et pour tout rencontré qu'un seul cas de surgénération lors de nos travaux[6]

[5]Rappelons que l'interface $\mathcal{I}_{sem-synt}$ est définie comme une relation m-to-n.

[6]L'erreur provenait de surcroit non pas de notre grammaire mais de notre implémentation.

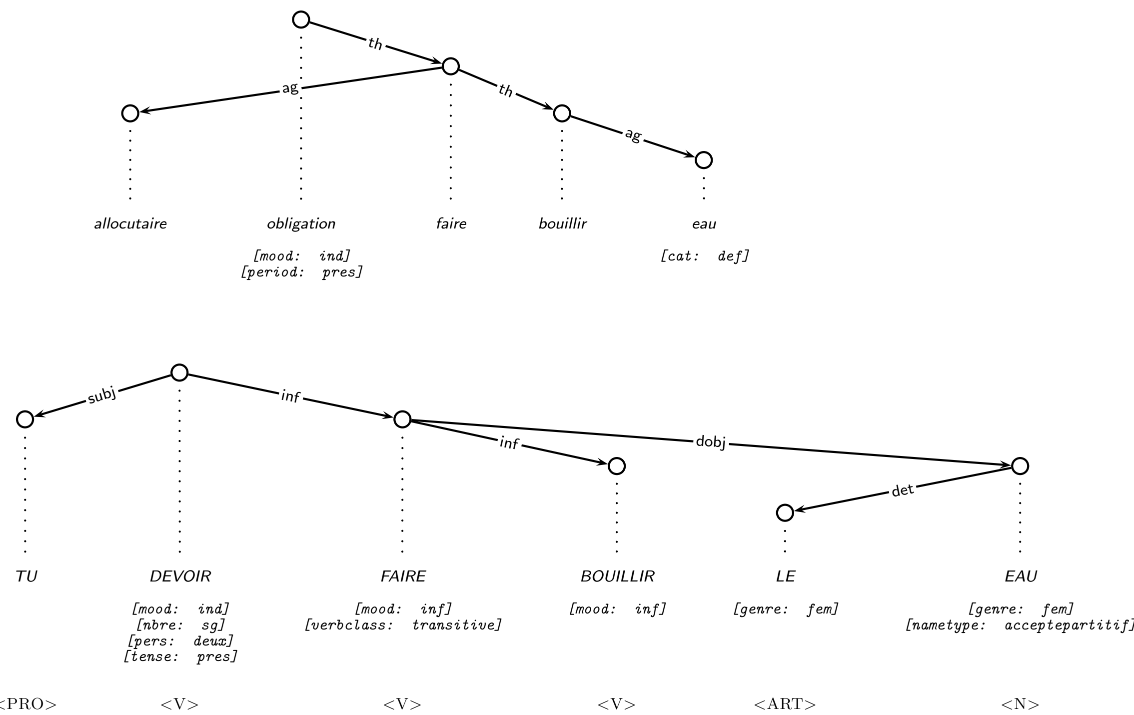


FIG. 7.2 – Exemple de génération automatique du graphe prédicatif (haut) en un arbre de dépendance (bas) via notre interface $\mathcal{I}_{sem-synt}$

7.4.2 Aspects quantitatifs

Temps d'élaboration : La grammaire construite pour la validation expérimentale nous a demandé environ 2 semaines de travail, à un rythme de ± 12h / jour. Il a en effet fallu :

- Constituer la liste des éléments lexicaux et grammaticaux à intégrer ;
- Etudier la meilleure manière de modéliser la "grammaire noyau" du français[7] ;
- Encoder tout ceci dans plus de 900 règles GUST/GUP ;
- Constituer la batterie de tests, en veillant à respecter les spécifications de $\mathcal{G}_{sem}$;
- Corriger, au sein de notre grammaire, les nombreuses erreurs mises en lumière par la batterie de tests. La plupart du temps, il s'agissait d'erreurs dues à des omissions (fonction syntaxique, partie du discours, trait grammatical manquant) ou des incohérences ;
- Factoriser au maximum l'information contenue dans les règles grammaticales pour obtenir une grammaire structurée et facilement manipulable ;
- Et poursuivre ce travail jusqu'à obtenir une grammaire entièrement opérationnelle.

Il est important de signaler que ce temps d'élaboration peut difficilement être généralisé à d'autres contextes. Une bonne partie de notre travail a en effet consisté en l'élaboration de la "grammaire noyau" du français, tâche particulièrement ardue au niveau linguistique. De toute évidence, doubler la taille de cette grammaire devrait prendre largement moins de deux semaines, car toutes les "briques" de base sont déjà placées.

A première vue, le temps d'élaboration d'une grammaire GUST/GUP semble donc plutôt suivre une progression de type logarithmique en fonction de la taille de celle-ci[8].

[7]Ceci est loin d'être un tâche aisée, car les grammaires traditionnelles se contredisent parfois allègrement !

[8]Cette intuition demande bien sûr à être vérifiée. En particulier, il est possible que, passé un certain stade de développement, la progression redevienne linéaire (voire pire) au vu de la complexité de gestion d'une grammaire à large couverture, cfr (Candito, 1999). Une grammaire GUST/GUP, de par sa nature lexicalisée et articulée, se prête toutefois mieux à une développement à grande échelle que des TAG. Dans tous les cas, les données récoltées dans le cadre de ce travail sont largement insuffisantes pour permettre de trancher cette question.

Evolution de la couverture : De nouveau, les résultats sont relativement peu "parlants", nous n'avons pas décelé de réelle régularité sous-jacente dans l'évolution de la couverture de notre batterie de tests (à part bien entendu le fait que celle-ci augmentait peu à peu au fil des corrections effectuées sur la grammaire, mais ceci est presque une lapalissade).

Ainsi, durant plusieurs jours, la couverture de notre grammaire fut tout simplement nulle, car certaines règles de base posaient problème (par ex. concernant la fonction "sujet du verbe fini"), et rendait inopérant l'ensemble de notre batterie de tests.

Une fois résolus ces problèmes initiaux, la couverture s'éleva à environ 30 %. L'un ou l'autre problème subsistait en effet dans la grammaire et empêchait le moteur d'inférence de compléter son analyse. Nous avons ainsi remarqué qu'il était possible de générer avec succès la majorité des structures de tests constituant les 70 % fautifs en désactivant un seul de nos 8 principes[9].

Puisque nous avions conçu nos tests de manière à diversifier au maximum les phénomènes linguistiques (passivation, négation, verbes de contrôle, circonstanciels, etc.), la correction d'une erreur aboutissait la plupart du temps à l'augmentation de notre couverture d'une seule unité, parfois deux ou trois quand le phénomène en question se répétait dans plusieurs structures.

Il est également arrivé que la qualité de notre "couverture" régresse suite à des modifications, heureuses ou malheureuses, de la grammaire.

Les règles développées sont à présent bien établies, et nous sommes convaincus que l'ajout de nouvelles structures donnerait des taux de réussite approchant 100 %. Nous avons d'ailleurs récemment conçu 5 nouveaux tests pour le vérifier, et ceux-ci ont tous été parfaitement générés.

Commentaire méthodologique : L'analyse de l'évolution de la couverture (selon diverses variables : taille de la grammaire, nombre de corrections effectuées, etc.) est indéniablement utile concernant les grammaires probabilistes, mais se prête difficilement à des grammaires basées sur des règles, surtout quand la grammaire et le corpus sont de taille réduite, comme dans notre cas.

Nous pouvons en effet observer ici l'inadéquation, pour la validation expérimentale de notre travail, d'une approche uniquement basée sur la notion de *couverture*. Bien que cette dernière soit *in fine* proche des 100 %, elle ne reflète en rien la qualité intrinsèque du travail, et encore moins l'évolution de cette dernière[10].

Il est à notre sens plus fructueux de vérifier le bon fonctionnement de notre implémentation en terme de *phénomènes linguistiques* :

1. Quel est la quantité et la complexité des phénomènes linguistiques abordés ?
2. La modélisation de ceux-ci correspond-elle à la réalité de la langue ?
3. Et bien sûr, l'implémentation informatique constitue-t-elle un reflet fidèle de ces modèles ?

Ces trois questions ont été abondamment discutées dans les chapitres précédents, et la qualité de notre travail peut se mesurer à l'aune de celles-ci.

Au vu des limitations et imperfections importantes de notre implémentation, il est donc évident qu'il reste encore du chemin à faire avant d'obtenir un outil de qualité professionnelle, mais l'objectif que nous nous étions fixé au départ (implémenter une interface sémantique-syntaxe basée sur GUST/GUP) est indéniablement atteint.

[9] XDK possède en effet une fonctionnalité très intéressante : il est possible d'activer ou de désactiver des principes (i.e. des ensembles de contraintes) depuis la GUI.

[10] La majorité des formalismes linguistiques ne donnent d'alleurs aucun détail sur leurs "performances expérimentales" sur des corpus, ou seulement de manière marginale. Des ouvrages incontournables en TALN tels que (Chomsky, 1981; Bresnan, 1982; Pollard et Sag, 1994; Joshi, 1987) ne font ainsi aucune mention de résultats en terme de couverture, et ce pour une raison évidente : l'objectif de ces formalismes (et de leurs implémentations associées) est avant tout d'offrir une modélisation systématique de phénomènes linguistiques, et non d'obtenir une quelconque couverture quantifiée. Ce qui ne signifie nullement que de telles préoccupations puissent être négligées, mais elles n'ont d'intérêt réel que pour des systèmes de TALN de qualité industrielle (il s'agit alors souvent de systèmes hybrides, cfr. 1.1.2) et non pour les formalismes linguistiques en eux-mêmes.

Chapitre 8

Conclusions et Perspectives

8.1 Résumé

Nous avons étudié au sein de ce travail la question des *interfaces sémantique-syntaxe* dans les modèles linguistiques formels utilisés en TALN. Après avoir effectué un rapide aperçu des grammaires de dépendance et de la Théorie Sens-Texte (chap. 2), nous nous sommes plus spécifiquement intéressés au modèle des *Grammaires d'Unification Sens-Texte*. Nous avons présenté ses fondements linguistiques, son architecture, son formalisme de description, et avons illustré son mécanisme sur un exemple détaillé (chap 3).

Nous avons ensuite analysé au chap. 4 la place de la représentation logique au sein de l'architecture du modèle, et avons illustré divers phénomènes linguistiques pertinents pour notre interface sémantique-syntaxe.

Le chap. 5 a été consacré à l'axiomatisation de GUST/GUP en problème de satisfaction de contraintes. Nous avons rappelé les grandes principes de la programmation par contraintes, avons introduit le formalisme de *Extensible Dependency Grammar*, avons montré les similarités entre celui-ci et GUST/GUP, et avons procédé à la traduction de ce dernier en grammaire XDG.

Nous sommes alors passés (chap. 6) à la description de notre *implémentation*, constituée d'un compilateur de grammaires GUST/GUP en grammaires XDG et d'un ensemble de 8 contraintes supplémentaires intégrées à XDG. Nous avons détaillé le cahier des charges de notre compilateur, son architecture et les étapes de son fonctionnement. Quant aux contraintes XDG, nous avons expliqué le rôle de chacune d'elles et avons discuté leurs performances. Nous avons également réalisé en fin de chapitre une petite analyse comparative de nos deux formalismes.

Enfin le chap. 7 a présenté la validation expérimentale de notre travail par l'application de notre implémentation à une mini-grammaire réalisée par nos soins et axée sur le vocabulaire culinaire et un "corpus" de 50 graphes prédicatifs.

8.2 Perspectives

De toute évidence, notre travail offre de nombreuses perspectives pour être ultérieurement étendu dans diverses directions, tant au niveau linguistique que formel et technique :

Aspects linguistiques :

1. Dans le cadre de ce travail, nous avons ouvertement privilégié l'opération de *génération* au détriment de l'*analyse*. Il serait donc intéressant d'étudier plus avant la question de l'**analyse** (§ 1.1.1), à la fois dans ses aspects linguistiques (comment réaliser la levée d'ambiguïté) et techniques (la réversibilité de la grammaire est-elle assurée ?).

2. L'extension de notre implémentation à d'**autres niveaux linguistiques** (§ 2.2.3, 3.3) - topologie, morphologie, phonologie - est un autre sujet digne d'intérêt. XDG est capable de fonctionner avec un nombre arbitraire de dimensions et d'interfaces, cette extension est donc théoriquement tout à fait envisageable.

3. Il est de surcroit également possible d'étendre les niveaux sémantiques et syntaxiques *eux-mêmes*, notamment via l'intégration de la **structure communicative** (§ 2.2.3) et/ou des **relations de portée** (§ 4.1) en vue du traitement de la quantification logique.

4. Au niveau purement linguistique, nous avons laissé de côté de nombreux phénomènes très intéressants, notamment tout ce qui concerne l'*extraction* (relatives, interrogatives indirectes) et la *coordination*. Une étude plus approfondie du fonctionnement des **arbres à bulles** (§ 3.9) serait sans nul doute bienvenue pour traiter de ces questions.

5. Enfin, l'architecture de notre **lexique** gagnerait à être enrichie. Le lexique que nous avons conçu dans le cadre de la validation expérimentale (§ 7.2.2) est encore relativement peu *structuré*, et la question de sa *maintenance* (sa cohérence est-elle assurée suite à des modifications ?) n'a pas été étudiée en détail.

Aspects formels :

6. Le traitement des **unités polylexicales** reste peu satisfaisant. Bien que nous ayons réussi à les prendre en compte par le biais d'ajouts de *noeuds sémantiquement vides* (§ 5.2.5, 5.3.4, 6.2.3), cette solution pose des problèmes de performance (§ 6.3.2). La question de la correspondance 1 :1 entre unités de différents niveaux est en train d'être étudiée concernant XDG, et des solutions concrètes seront vraisemblablement proposées dans les prochains mois.

7. Notre axiomatisation de GUST/GUP en grammaires XDG (§ 5.3) n'est pas réellement formalisée au niveau mathématique ; il serait intéressant de développer une formalisation complète de notre travail (via des formules de la logique du 1er ordre exprimant les modèles admissibles de la grammaire), à l'instar de (Debusmann et Smolka, 2006; Debusmann, 2006).

Aspects techniques :

8. La **vitesse** de l'interface $\mathcal{I}_{sem-synt}$ gagnerait à être optimisée en renforcant la propagation et en diminuant le nombre de variables (§ 6.3.2).

9. Nous attendons avec impatience les prochaines versions de XDG intégrant de nouvelles **fonctionnalités** telles que la guidance statistique, le *supertagging*, l'ajout de contraintes globales, l'utilisation de la librairie *GeCode*, etc. (§ 6.3.2). Ces fonctionnalités permettront sans nul doute d'améliorer très sensiblement les performances de notre interface.

10. Plutôt que d'intégrer la structure sémantique des éléments de notre batterie de tests dans des contraintes spéciales, comme nous le faisons actuellement (§ 6.3), l'idéal serait de réécrire XDG pour qu'il accepte directement en entrée non seulement des phrases linéaires mais également des graphes quelconques, définis selon un formalisme particulier.

11. Bien que nous nous soyions efforcés de concevoir une implémentation aussi solide que possible (§ 6.2.3), il est évident que celle-ci pourrait encore gagner en **robustesse**, notamment via l'ajout de modules permettant de *détecter* tout type d'erreur dans l'élaboration de grammaires, et éventuellement guider l'utilisateur dans sa *correction*.

12. Dans la même veine, l'interface utilisateur d'`auGUSTe` est à l'heure actuelle encore rudimentaire et devrait dans l'idéal être intégralement repensée pour offrir plus d'*efficacité* et de *convivialité* à l'utilisateur.

13. Enfin, la possibilité d'**extraire automatiquement** des grammaires GUST/GUP à partir de *corpus* via des techniques d'apprentissage supervisé mériterait d'être étudiée en détail. Des recherches ont été menées à ce propos sur XDG à partir du tchèque (Bojar, 2004), et les résultats sont très intéressants. Ce procédé permettrait d'étudier le comportement de grammaires GUST/GUP *à large couverture*, ce qui n'a jamais été réalisé à ce jour.

8.3 En guise de conclusion

Ce mémoire fut assurément un travail très prenant, et je suis rétrospectivement très heureux d'avoir choisi ce sujet au printemps dernier et de l'avoir proposé à mes promoteurs. Ce fut une réelle satisfaction personnelle de pouvoir combiner des disciplines ausi diverses que la linguistique, les mathématiques, l'informatique (et même un peu de psycholinguistique et de philosophie du langage !) pour mener à bien une recherche multidisciplinaire.

Je ne pense pas m'être pour autant éloigné du métier d'ingénieur, au contraire. Après tout, ce qui définit l'ingénieur réside moins dans un série de *disciplines* (chimie, mécanique, électronique, etc.) que dans une *méthode de travail* : l'ingénieur est avant tout un scientifique - et un spécialiste des technologies - qui cherche à résoudre des problèmes complexes en conçevant des modèles, des descriptions abstraites (processus chimique, schéma électrique, etc.), et en appliquant celles-ci pour développer des systèmes techniques souvent très élaborés (centrale nucléaire, moteur à explosion, circuit électronique, etc.) et les mettre au service de l'homme et de la société.

En ce sens, le travail présenté ici est un pur travail d'ingénierie : nous avons étudié des phénomènes empiriques, les avons modélisés au sein d'un formalisme, et avons développé à partir de là un système informatique permettant de les traiter. La seule différence existant par rapport à des sujets plus "classiques" concerne dans la nature du matériau étudié : plutôt que d'analyser un moteur, un produit chimique ou une structure architecturale, notre objet de travail est tout simplement... le langage humain. Matériau particulièrement riche et complexe s'il en est, mais qu'il est néammoins possible - c'est en tout cas notre conviction - de modéliser et de systématiser. Et qui sait, peut-être définira-t-on un jour la linguistique comme une "science de l'ingénieur" comme les autres ?

Le sujet de ce mémoire nous a également attiré par son caractère directement *utile* à la recherche actuelle en TALN : étant personnellement intéressé par la modélisation mathématique des langues, nous avons parcouru la littérature sur ce thème, avons étudié quelques articles en détail, et avons ensuite contacté le chercheur à l'origine de ces derniers (Sylvain Kahane, en l'occurrence) pour l'interroger sur les sujets "porteurs" en terme de recherche, et susceptibles de déboucher sur un travail de fin d'études. GUST/GUP étant un formalisme essentiellement mathématique et qui n'avait pour l'heure jamais été réellement implémenté, l'occasion était rêvée de proposer une implémentation partielle de celui-ci, et c'est que nous avons fait.

Nous espérons donc de tout coeur que notre travail portera ses fruits et qu'`auGUSTe` pourra être d'une utilité certaine aux chercheurs de ce domaine. Il m'aura en tout cas convaincu du caractère passionnant de la recherche en TALN, et c'est dans cette voie que j'ai décidé d'orienter ma carrière professionnelle dans les années à venir.

Annexe A

Installation et mode d'emploi d'`auGUSTe`

A.1 Installation

A.1.1 Installez Python :

L'installation de **Python** est indispensable pour démarrer le compilateur. Ce logiciel peut-être téléchargé à l'adresse `http://www.python.org`, rubrique "downloads".

(Pour les utilisateurs de Windows, il faut télécharger le "Windows Installer")

Suivez les instructions indiquées dans la procédure d'installation, comme à l'habitude.

A.1.2 Installez Mozart/oz :

Pour pouvoir utiliser notre interface sémantique-syntaxe, l'installation de **Mozart** est également nécessaire. Le logiciel est disponible à l'adresse `http://www.mozart-oz.org`, rubrique "download".

Suivez les instructions indiquées dans la procédure d'installation, comme à l'habitude.

A.1.3 Installez auGUSTe

Notre logiciel est téléchargeable à l'adresse suivante :

`http://ece.fsa.ucl.ac.be/plison/memoire/auGUSTe.zip`

Une fois le fichier téléchargé, décompressez-le sur votre disque. Ensuite :

- Pour les utilisateurs de **Windows**, il n'y a normalement plus rien à effectuer. Nous avons déjà inclus dans le logiciel une version compilée de "XDG version auGUSTe" pour Windows. Vérifiez néammoins que tout fonctionne sans accroc en double-cliquant sur l'exécutable `xdk.exe` présent dans le répertoire `plison-xdk`. Si une fenêtre s'affiche, tout est parfait. Sinon, il faudra vraisemblablement compiler le fichier à partir de la source (recontactez-nous si vous rencontrez ce problème).
- Pour les utilisateurs **Unix/Linux** : il est nécessaire de recompiler "XDK version auGUSTe" à partir du code source. La marche à suivre est la suivante :
  1. Supprimer le répertoire `plison-xdk` ;
  2. Exécutez la commande `ozmake -extract -p plison-xdk-0.3.pkg` ;
  3. Changez de répertoire : `cd plison-xdk` ;
  4. Préparez l'installation : `scripts/prepinstall` ;
  5. Enfin, compilez le logiciel : `ozmake`.

A.1.4 Installation de Dia et sa feuille de style

L'installation de **Dia** est optionnelle, et n'a d'intérêt que dans l'éventualité où vous désirez créer de nouvelles grammaire ou modifier des règles déjà existantes.

1. Installez tout d'abord le logiciel Dia.
   – Pour les systèmes Unix, il est téléchargeable à cette addresse : `http://gnome.org/projects/dia`
   – Pour Windows, voyez cette addresse et suivez les instructions : `http://dia-installer.sourceforge.net`
2. Dans le répertoire `auGUSTe/dia-sheet`, lancez le fichier `install-sheet.py`, qui copiera les fichiers à l'endroit approprié.

Note : pour une raison encore inconnue, il arrive que la feuille de style donne des résultats graphiques "anormaux" (objets de taille excessive, absence de couleur, etc.) sur certaines plateformes. Contactez-nous si vous rencontrez ce problème.

A.2 Utilisation d'auGUSTe

A.2.1 Menu principal

Pour utiliser notre logiciel, il suffit de démarrer le fichier `auGUSTe.py` dans le répertoire de base. Un menu similaire à la figure A.1 doit s'afficher.

Ce menu comprend 6 fonctionnalités :

1. Conversion d'un ensemble de fichiers graphiques Dia représentant une grammaire en un seul fichier textuel GUST/GUP ;
2. Compilation d'une grammaire GUST/GUP en son équivalent XDG ;
3. Conversion d'un ensemble de fichiers graphiques Dia représentant une batterie de tests en un seul fichier textuel GUST/GUP ;
4. Intégration d'une batterie de tests dans XDK ;
5. Lancement de l'interface sémantique syntaxe.
6. Sortie.

```
/cygdrive/c/Documents and Settings/Proximedia/Bureau/Pierre/memoire/final

Welcome to auGUSTe, the PUG ==> XDG grammar compiler !
This is auGUSTe release 0.3, by Pierre Lison <plison@agora.eu.org>.
This software is licensed under the terms of the General Public License [GPL]
----------

Main Menu:
1: Conversion of a set of Dia files (grammar rules) to PUG format
2: Compilation of PUG grammar to XDG format
3: Conversion of a set of Dia files (test suite) to PUG format
4: Integration of PUG test suite into XDK
5: Launch the semantics-syntax interface !
6: Exit

Make your choice [1/2/3/4/5/6]: _
```

FIG. A.1 – Menu principal d'auGUSTe

Attention, ces étapes ne peuvent **pas** être utilisées dans n'importe quel ordre ! En effet :
- L'étape de compilation de la grammaire GUST/GUP (étape 2) requiert évidemment l'existence d'un fichier spécifiant cette grammaire ! A moins d'avoir développé celle-ci directement dans le fichier texte, il faut donc passer préalablement par l'étape 1 ;
- L'étape d'intégration de la batterie de tests dans le logiciel (étape 4) n'est possible qu'après avoir (1) obtenu un fichier spécifiant textuellement cette ensemble de tests, et (2) avoir déjà compilé la grammaire se rapportant à cette batterie[1].
- Enfin, le lancement de notre interface (étape 5) exige d'avoir compilé la grammaire et d'avoir intégré la batterie de tests.

La figure A.2 illustre graphiquement les dépendances entre ces différentes étapes.

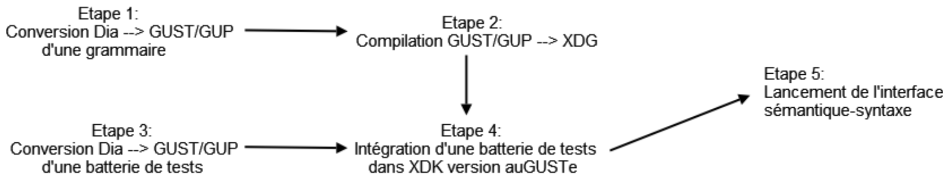


FIG. A.2 – Etapes à réaliser avant le lancement de l'interface

A.3 Utilisation de XDK version auGUSTe

Lorsque l'utilisateur sélectionne l'étape 5, "lancement de l'interface sémantique-syntaxe", il démarre en fait le programme XDK (Debusmann et Duchier, 2005).

L'utilisateur arrive alors dans un fenêtre similaire à celui présenté à la figure A.3.

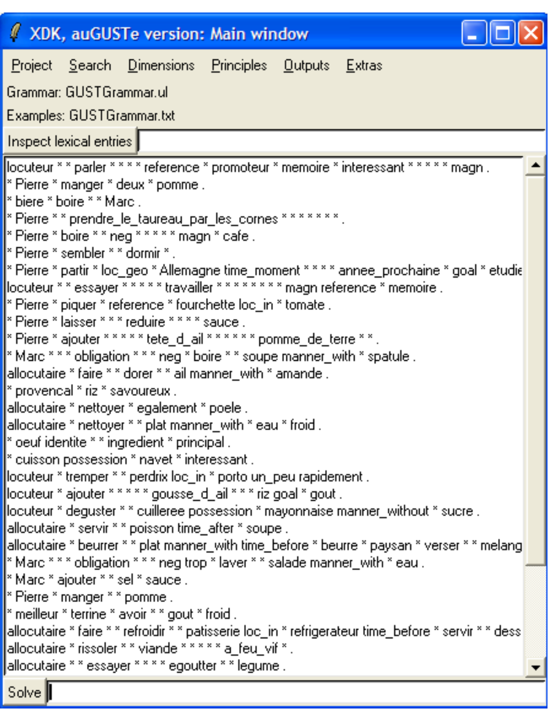


FIG. A.3 – Menu principal de XDK, version auGUSTe

[1]En effet, l'intégration des contraintes requiert la connaissance de certaines informations présentes dans la grammaire, telles que l'expansion maximale d'unités polylexicales.

La partie centrale de la fenêtre est occupée par la liste des exemples. Il suffit de double cliquer sur l'un d'entre eux pour lancer l'opération de génération via l'interface. Rappelons qu'à chaque exemple est associée une contrainte particulière spécifiant la structure du graphe sémantique.

En haut de la fenêtre, nous trouvons un certains nombre d'options. (Debusmann et Duchier, 2005) explique chacun d'eux en détail. Contentons-nous d'en épingler quatre :

- Le menu *Project* permet de charger une autre grammaire que celle utilisée par défaut pour notre interface. Il vous est ainsi loisible d'expérimenter d'autres grammaires que la nôtre construites en XDG, dans diverses langues (français, anglais, allemand, arabe).

- Dans le menu *Search*, il est possible de spécifier si l'on désire s'arrêter à la 1re solution trouvée ou si on veut toutes les énumérer. En sélectionnant le premier choix, le temps total de calcul peut ainsi être divisé en moyenne par trois.

- Le menu *Principles* spécifie les principes utilisés dans chaque dimension. Il est possible d'activer et de désactiver temporairement chaque principe. Ceci peut être d'une grande utilité lors du "débuggage" d'une grammaire : lorsqu'un test ne débouche sur aucune solution sans que l'on en comprenne la raison, l'on peut appliquer la technique suivante pour trouver la provenance de l'erreur : il suffit de désactiver un à un les principes et de relancer l'opération de génération, jusqu'à ce que celle-ci puisse déboucher sur des solutions.

  Cette technique nous offre ainsi une information cruciale pour la correction : si l'on observe par ex. que le principe `GUSTSagittalConstraints` est le responsable du caractère insoluble de notre test, nous pouvons en conclure que l'erreur provient d'une règle sagittale.

- Le menu *Output* permet de choisir le type de sortie voulu. Par défaut, il s'agit d'une fenêtre illustrant les graphes sur chacune des dimensions. Une sortie LaTeXest également possible.

Une fois qu'un exemple a été sélectionné via un double clic, le fenêtre d'*Oz Explorer* (figure A.4) s'ouvre. Cet outil permet de visualiser quasiment en temps réel l'évolution de la recherche des solutions. Les carrés rouges indiquent une "impasse" (aucune solution possible), les losanges verts indiquent une solution trouvée, les ronds bleus indiquent un embranchement (i.e. une distribution), et les grands triangles rouges dénotent un sous-arbre n'ayant donné aucune solution.

En bas d'*Oz Explorer* est signalé le temps total de la recherche, ainsi que d'autres informations utiles : nombre de solutions, d'embranchements, profondeur de la recherche, etc.

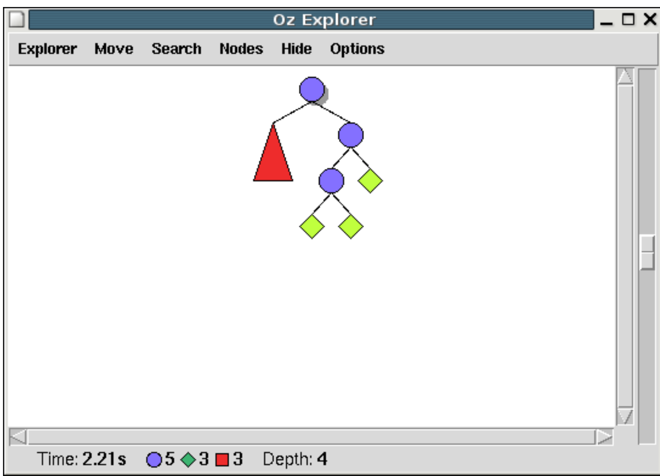


FIG. A.4 – Oz Explorer

Pour découvrir les solutions, il suffit de double-cliquer sur l'un des losanges verts dénotant une solution. Une fenêtre similaire à celle illustrée à la figure A.6 s'ouvre alors. La structure sémantique est présentée en bas et la structure syntaxique en haut. En dessous de chaque unité lexicale, nous indiquons les éventuels grammènes. La structure syntaxique indique de plus la partie du discours de l'unité.

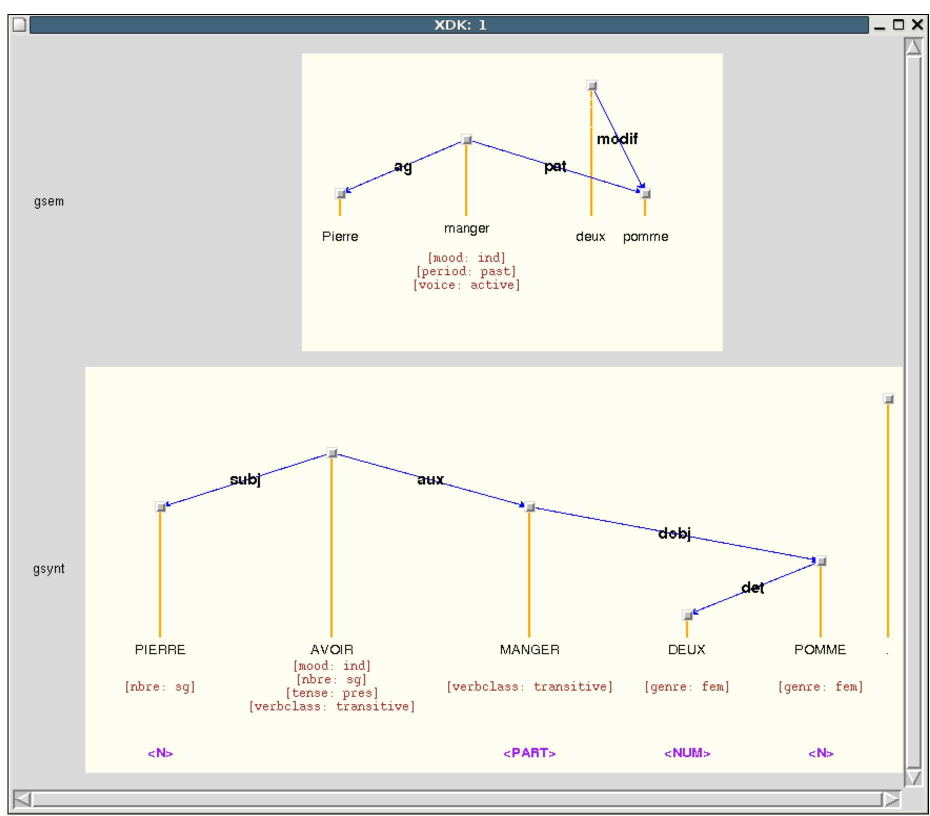


Fig. A.5 – Exemple de résultat généré par notre interface

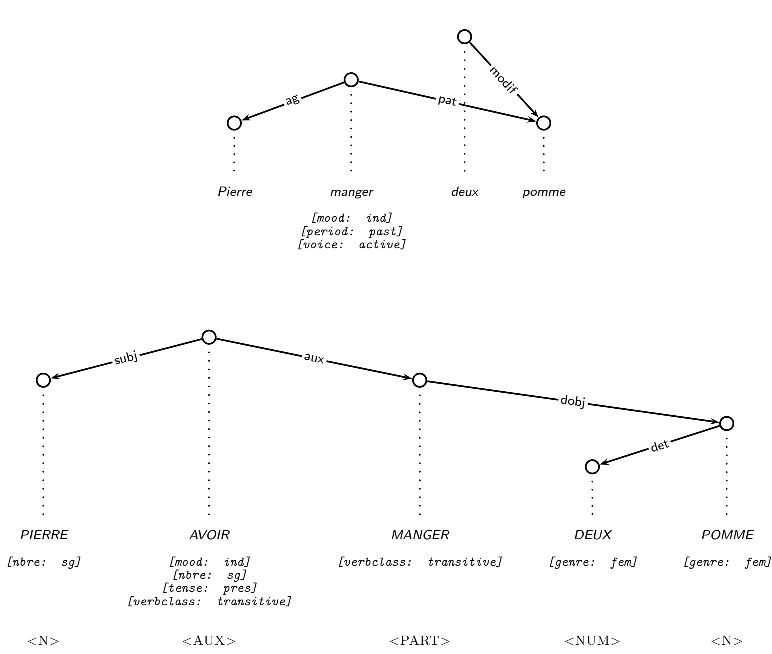


Fig. A.6 – Exemple de résultat généré par notre interface - version LaTeX

A.4 Utilisation de Dia pour la conception de structures GUP

Pour permettre de conçevoir/modifier aisément des structures GUST/GUP, nous avons développé un petit outil graphique, basé sur le logiciel Dia. Son utilisation est très intuitive.

Nous supposons ici que la feuille "GUST" a déjà été installée (voir section "installation"). Pour l'activer, démarrez Dia, cliquez sur le menu déroulant situé au milieu de la fenêtre Dia, et sélectionnez la feuille "GUST". Vous devez alors obtenir une fenêtre semblable à la figure A.7.

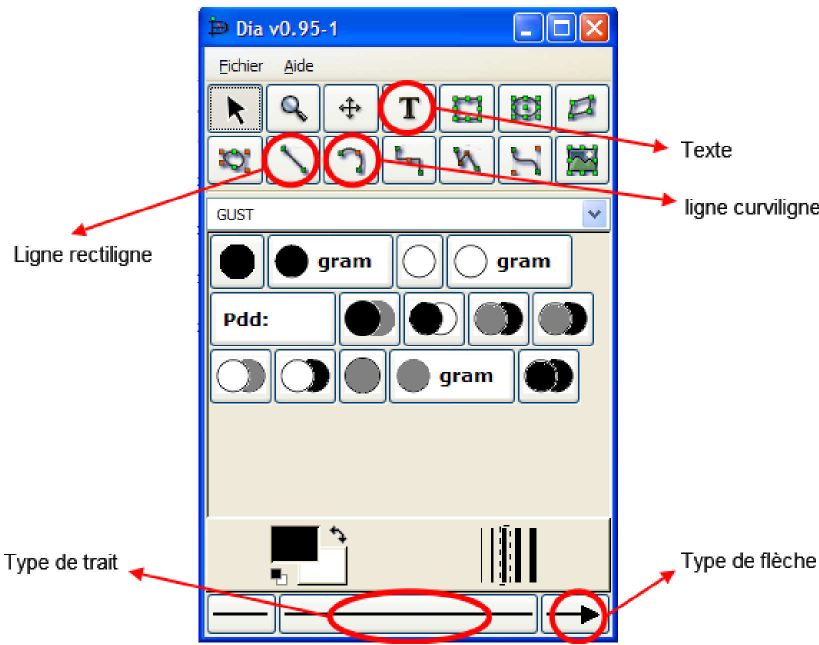


FIG. A.7 – Utilisation de Dia pour la conception de structures GUP

La feuille "GUST" contient au total 14 objets différents, mais seuls les 5 premiers (noeud saturé, grammène saturé, noeud insaturé, grammène insaturé, et partie du discours) nous seront véritablement utiles.

Ajout d'un objet : Pour ajouter un objet (noeud/grammène saturé ou insaturé), cliquez sur l'objet correspondant et ajoutez-le au diagramme. Vous pouvez alors remplir son label.

Ajout d'un arc : Pour relier des objets par des arcs orientés, il faut

1. sélectionner tout en bas de la fenêtre le type de trait et de flèche souhaité ;
2. cliquer sur le bouton "ligne" (rectiligne pour les arcs normaux, curviligne pour les arcs $\mathcal{I}_{sem-synt}$)
3. insérer l'arc dans le diagramme. Les extrémités de l'arc **doivent** être reliées à des objets, il faut pour cela déplacer chaque extrémité de l'arc vers l'objet souhaité, jusqu'à ce que ce dernier s'illumine de rouge indiquant la possibilité de connection. Relâchez alors la souris[2].
4. Il **faut** ensuite étiqueter l'arc. Pour cela, cliquez sur le bouton "texte" dans la fenêtre, déplacez le curseur dans diagramme jusq'au milieu de l'arc, indiqué par une petite croix.

[2]Il est possible de vérifier a posteriori qu'un arc a bien été fixé à deux objets en cliquant dessus : si les deux extrémités sont indiquées en rouge, l'arc est bien fixé, tandis qu'une extrémité notée en vert indique que celle-ci est libre, en d'autres termes non attachée.

Lorsque l'arc s'illumine, relâchez la souris, et tapez le label de l'arc[3].

Il est important de vérifier que chaque arc est bien fixé à ses deux extrémités, et que celui-ci est étiqueté. Dans le cas contraire, sa traduction en structure GUP aboutira à une erreur !

Pour relier un noeud et un grammène, il faut réaliser la même opération. Le *trait* du grammène (par ex. "nbre" pour le grammène "nbre = sg") est indiqué via l'étiquette de l'arc, étiquette qui est également obligatoire.

A chaque type d'arc correspond un type de trait et de flèche particuliers :

1. Arc saturé entre noeuds : ligne rectiligne de type
2. Arc insaturé entre noeuds : ligne rectiligne de type
3. Arc $\mathcal{I}_{sem-synt}$: ligne curviligne de type
4. Arc entre un noeud et un grammène ou une partie du discours : ligne rectiligne de type

Enregistrement : Une fois votre structure construite, il suffit d'enregistrer en cliquant sur le bouton "Enregistrer" dans le menu. Attention, le fichier doit être sauvé sous le format **non compressé**. S'il s'agit d'une règle à insérer dans la grammaire, le nom du fichier doit se terminer en `_gsem`, `_gsynt` ou `_isemsynt`, selon la nature de la règle.

A.4.1 Remarques diverses

- Le convertisseur Dia ⇒ GUST connaît quelques problèmes avec les accents, il est donc préférable d'omettre pour l'instant les accents dans les structures.
- De la même manière, n'insérez pas de tiret '-', cela pose d'inutiles problèmes. Par contre, vous pouvez utilisez à loisir l'underscore '_'.
- Certains mots ne peuvent être utilisés, car il s'agit de mots réservés dans Mozart/Oz : `ref`, `mod`, `true` et `false`.
- Pour signaler au sein d'une règle qu'un arc peut être reproduit un nombre quelconque de fois (par ex. la relation 'epithète' à partir d'un nom), il suffit d'ajouter à l'étiquette de l'arc le symbole '*'.
- Il est possible de remplir le trait "partie du discours" par une liste d'éléments (ce qui signifie que la règle peut s'appliquer pour chacune de ces PdD), séparés par une virgule et un espace entre chacun.
- Lorsque vous concevez un graphe sémantique pour un test, vous **devez** indiquez la racine "communicative" du graphe. Pour cela, ajoutez au noeud communicativement dominant un grammène (à label vide) ayant pour trait "`themeHead`".

A.5 Exemple de grammaire

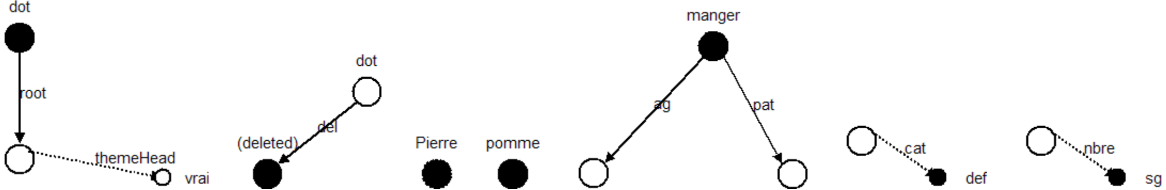
Terminons ce mode d'emploi un exemple complet de grammaire GUST/GUP écrite via le logiciel Dia. Pour que vous puissiez la tester et la modifier vous-même, nous l'avons intégrée à notre logiciel[4] : tous les règles ci-dessous sont placées dans le répertoire `minimal`.

Nous souhaitons attirer votre attention sur les deux premières règles de $\mathcal{G}_{sem}$ et de $\mathcal{G}_{synt}$. Elles sont indispensables car elles spécifient le comportement de deux labels particulier, le label ‹`root`› et le label ‹`del`›. Le premier est utilisé pour indiquer la racine de l'arbre (ou du dag), et le second dénote les noeuds supprimés.

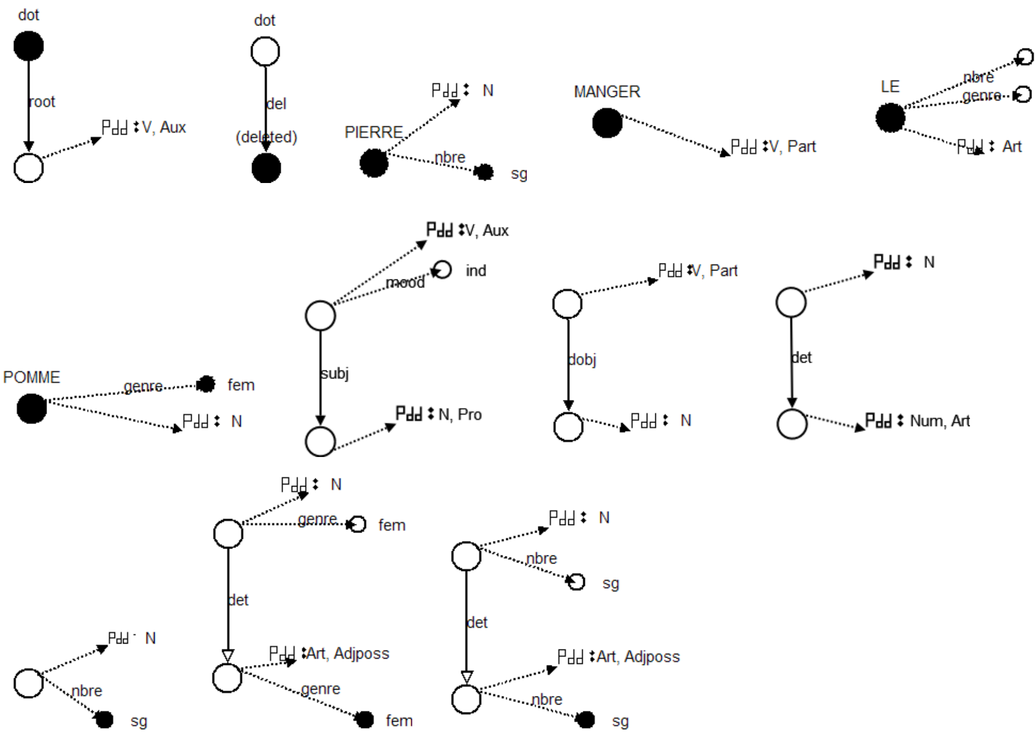
[3]Pour vérifier a posteriori qu'un élément de texte a bien été fixé à l'arc, il suffit de cliquez dessus : un point fixe rouge indique que celui-ci est fixé à l'arc, un point vert que celui-ci ne l'est pas

[4]Ainsi d'ailleurs que la grammaire utilisée pour la validation expérimentale (chap. 7), mais qui est bien sûr infiniment plus compliquée à saisir.

Règles $\mathcal{G}_{sem}$:



Règles $\mathcal{G}_{synt}$:



Règles $\mathcal{I}_{sem-synt}$:

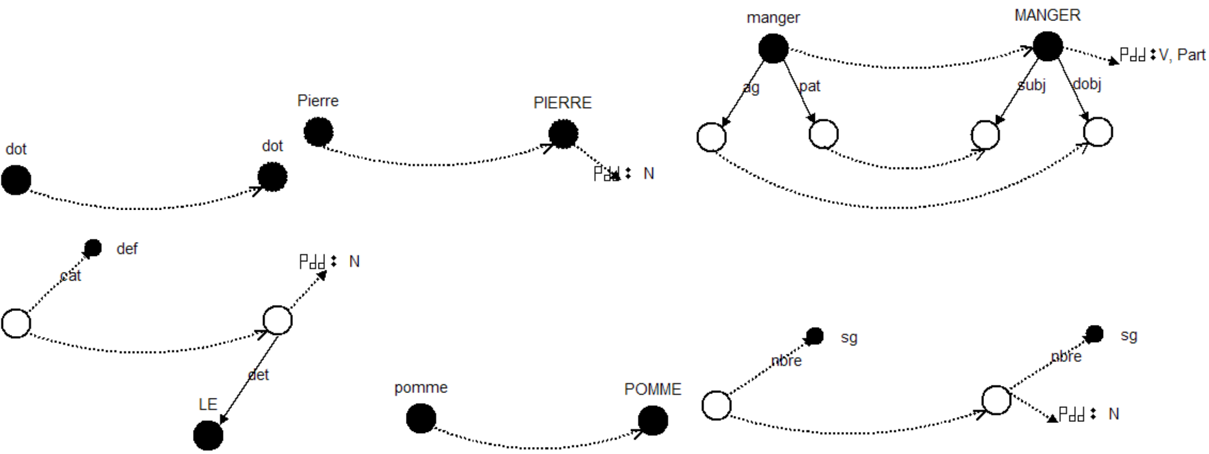


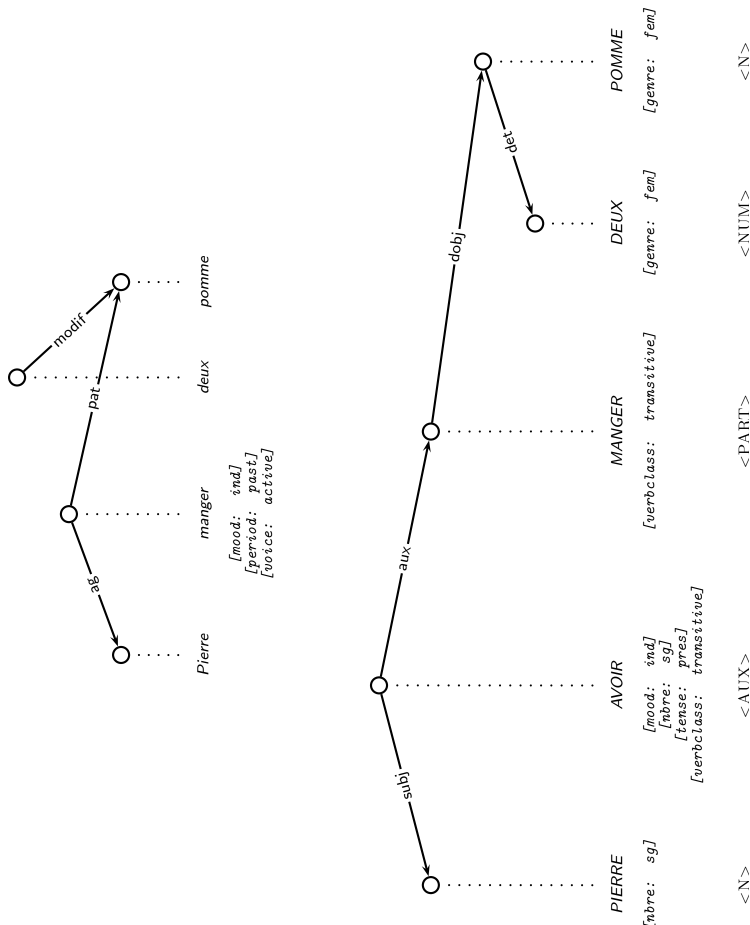
FIG. A.8 – Grammaire “minimale” à tester

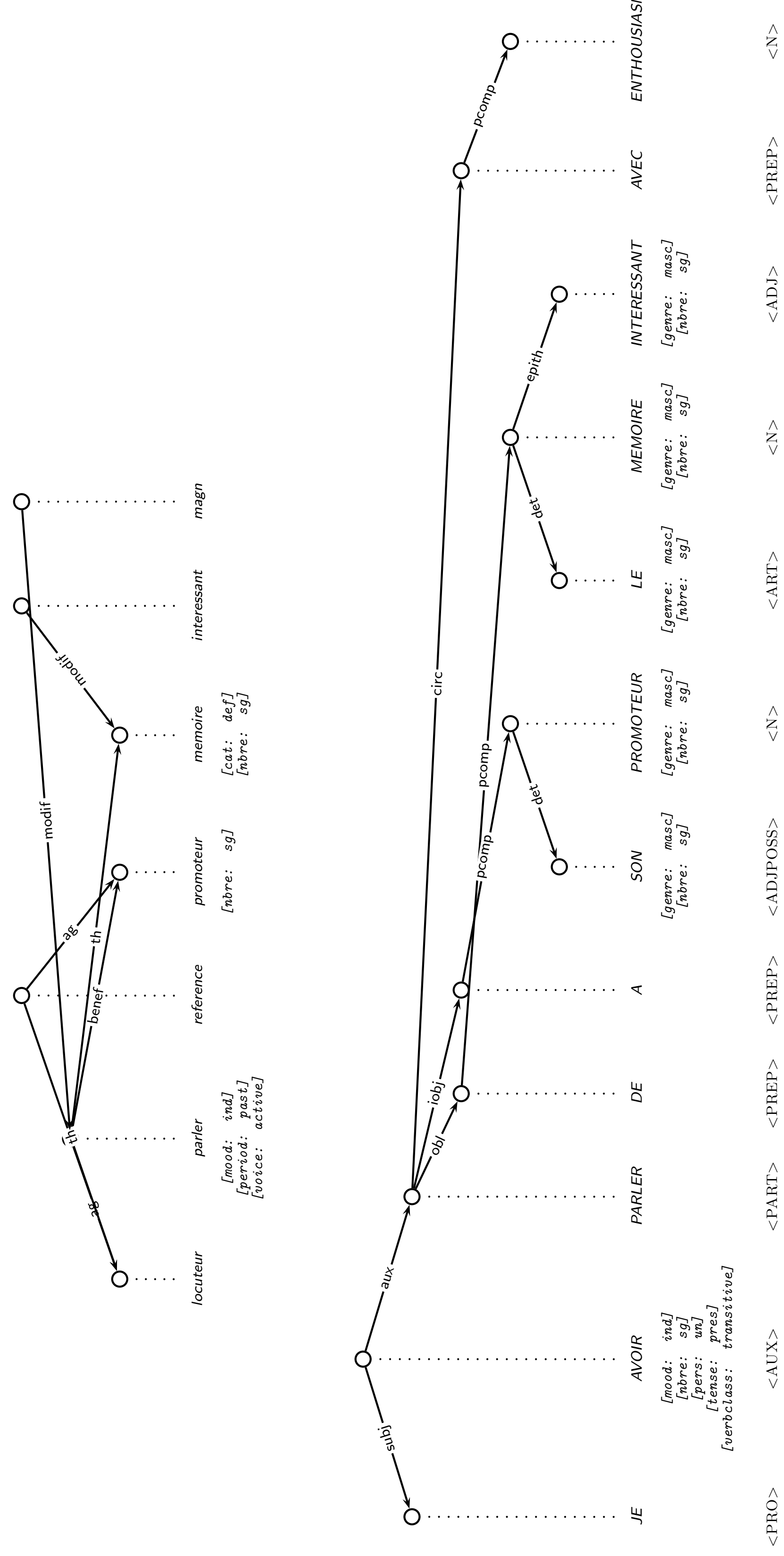
Annexe B

Résultats de la génération de 20 graphes sémantiques

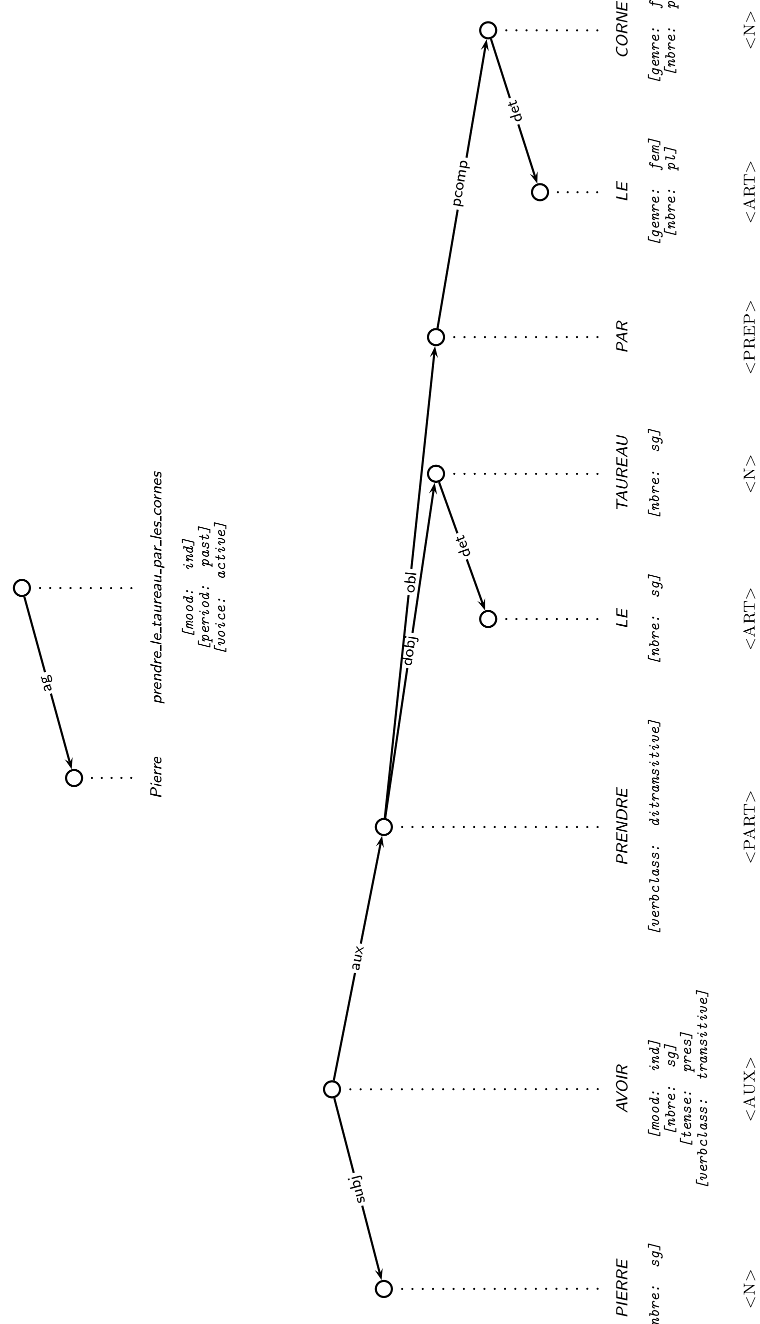
Les graphes ci-dessous ont été générés automatiquement à partir de notre interface. Mise à part la mise en page, ils sont ici présentés “tels quels”, sans aucune modification de notre part.

Rappelons que les arbres ci-dessous ne sont pas ordonnés. Néammoins, nous avons tâché autant que possible de conserver l’ordre canonique des mots, pour plus de lisibilité.

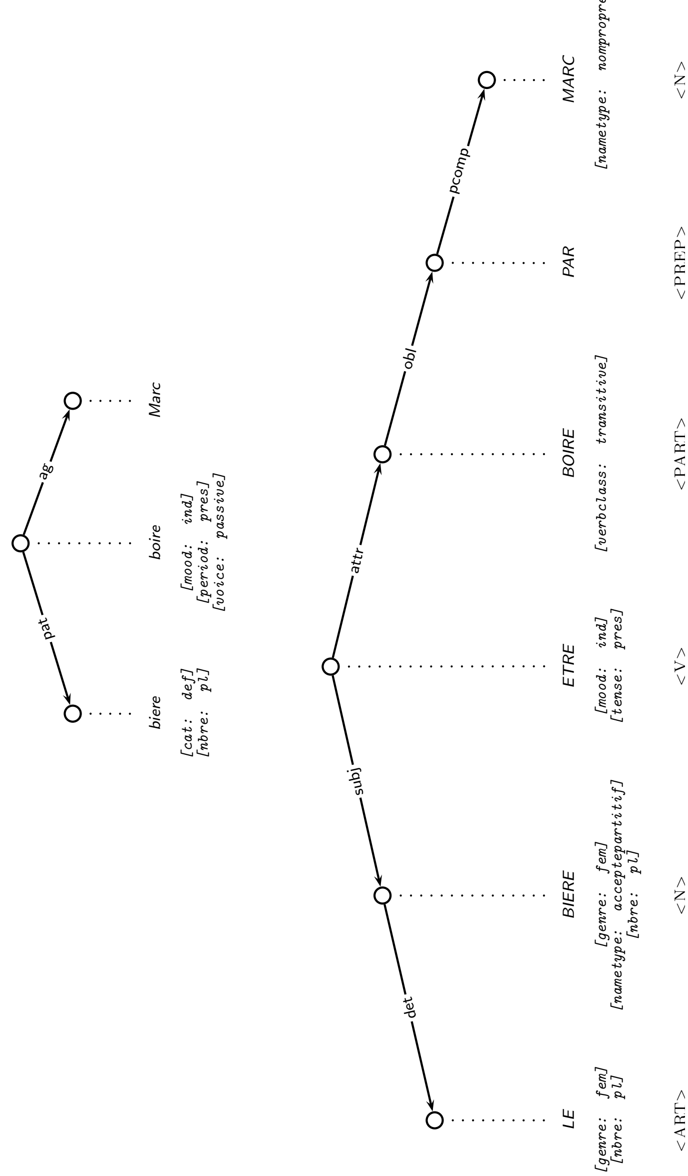


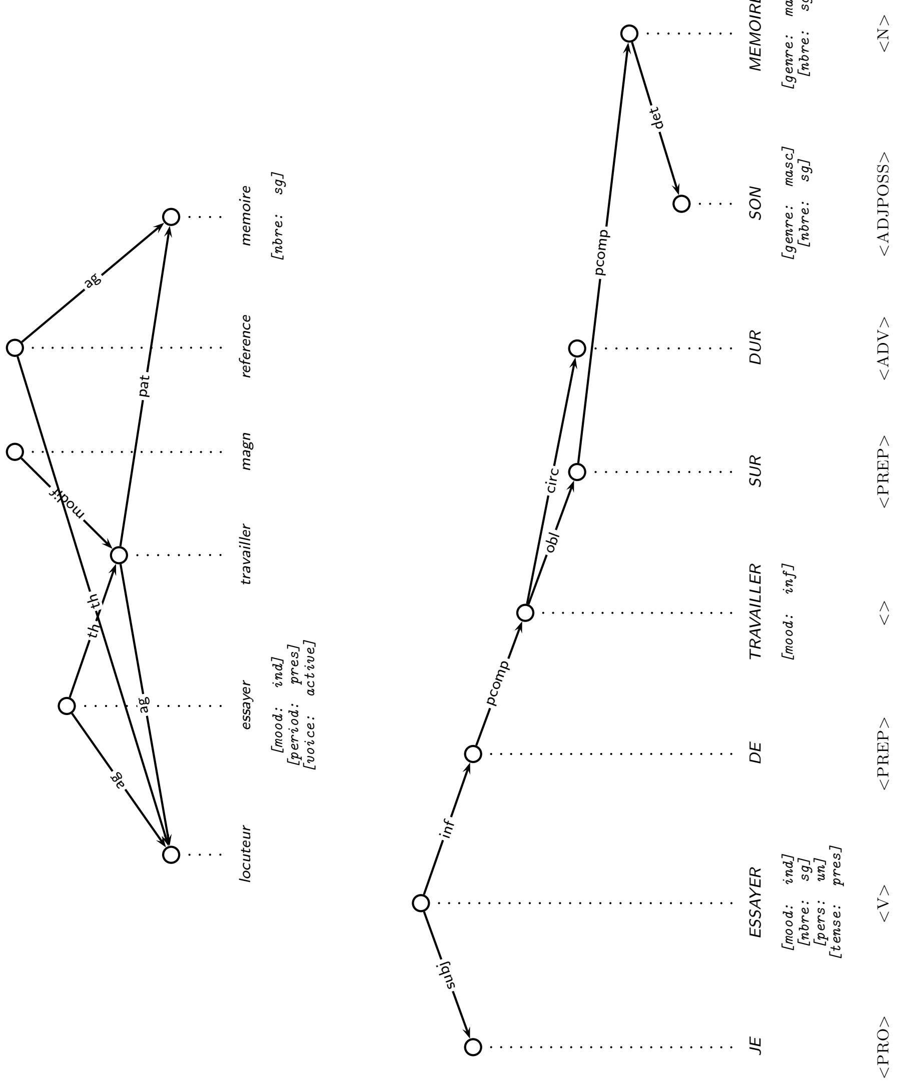


locuteur
parler
[mood: ind]
[period: past]
[voice: active]
reference
promoteur
[nbre: sg]
memoire
[cat: def]
[nbre: sg]
interessant
magn
ag
th
benef
modif
JE
AVOIR
[mood: ind]
[nbre: sg]
[pers: un]
[tense: pres]
[verbclass: transitive]
PARLER
DE
A
SON
PROMOTEUR
LE
MEMOIRE
INTERESSANT
[genre: masc]
[nbre: sg]
AVEC
ENTHOUSIASM
subj
aux
obl
iobj
pcomp
det
epith
circ
<PRO>
<AUX>
<PART>
<PREP>
<PREP>
<ADJPOSS>
<N>
<ART>
<N>
<ADJ>
<PREP>
<N>

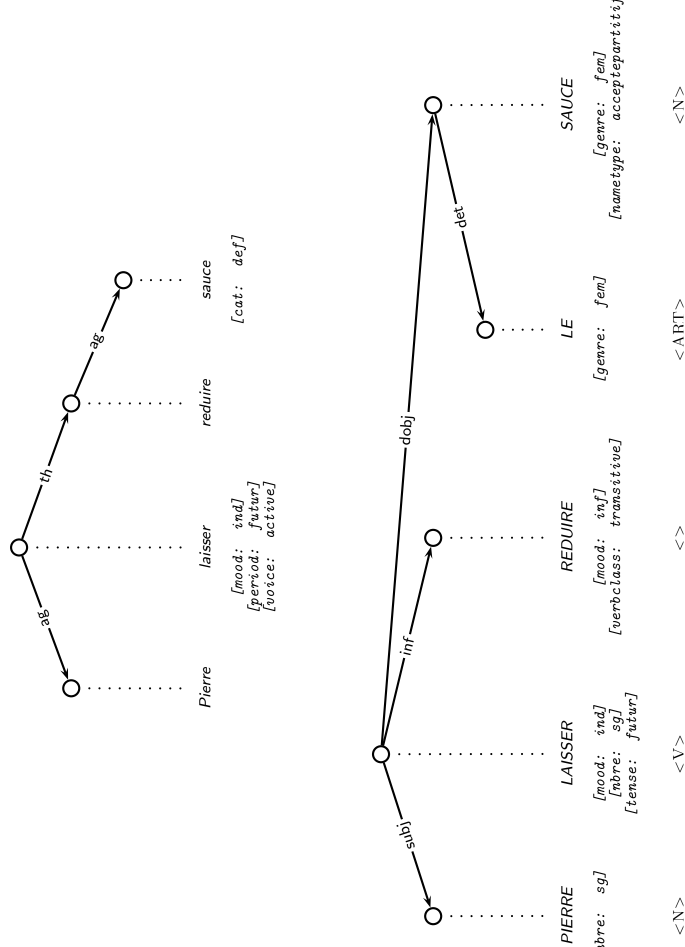


ag
Pierre
prendre_le_taureau_par_les_cornes
[mood: ind]
[period: past]
[voice: active]
subj
aux
dobj
obl
det
pcomp
det
PIERRE
[nbre: sg]
<N>
AVOIR
[mood: ind]
[nbre: sg]
[tense: pres]
[verbclass: transitive]
<AUX>
PRENDRE
[verbclass: ditransitive]
<PART>
LE
[nbre: sg]
<ART>
TAUREAU
[nbre: sg]
<N>
PAR
<PREP>
LE
[genre: fem]
[nbre: pl]
<ART>
CORNE
[genre: fem]
[nbre: pl]
<N>

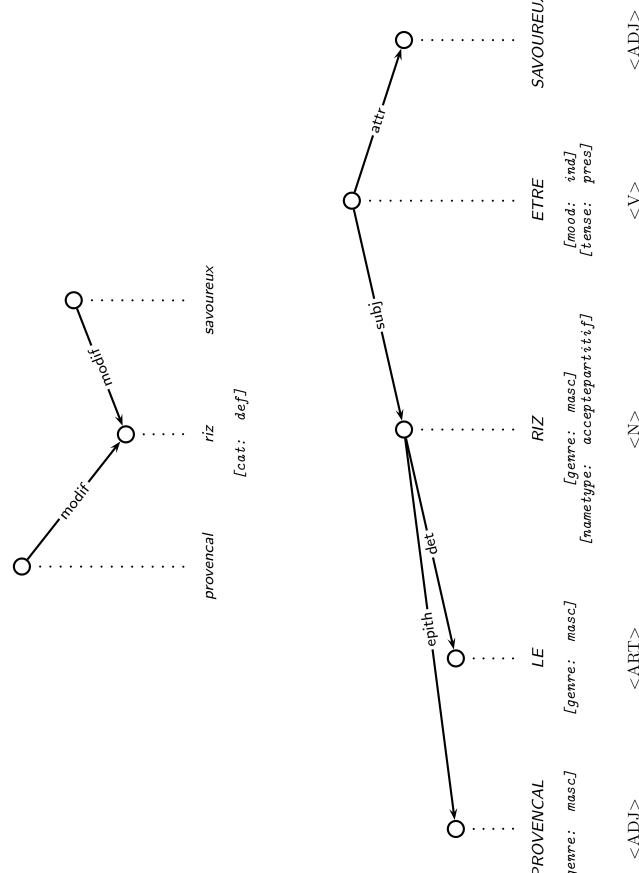

pat
ag
biere
boire
Marc
[cat: def]
[nbre: pl]
[mood: ind]
[period: pres]
[voice: passive]
det
subj
attr
obl
pcomp
LE
BIERE
ETRE
BOIRE
PAR
MARC
[genre: fem]
[nbre: pl]
[genre: fem]
[nametype: accepteparititif]
[nbre: pl]
[mood: ind]
[tense: pres]
[verbclass: transitive]
[nametype: nompropre]
<ART>
<N>
<V>
<PART>
<PREP>
<N>



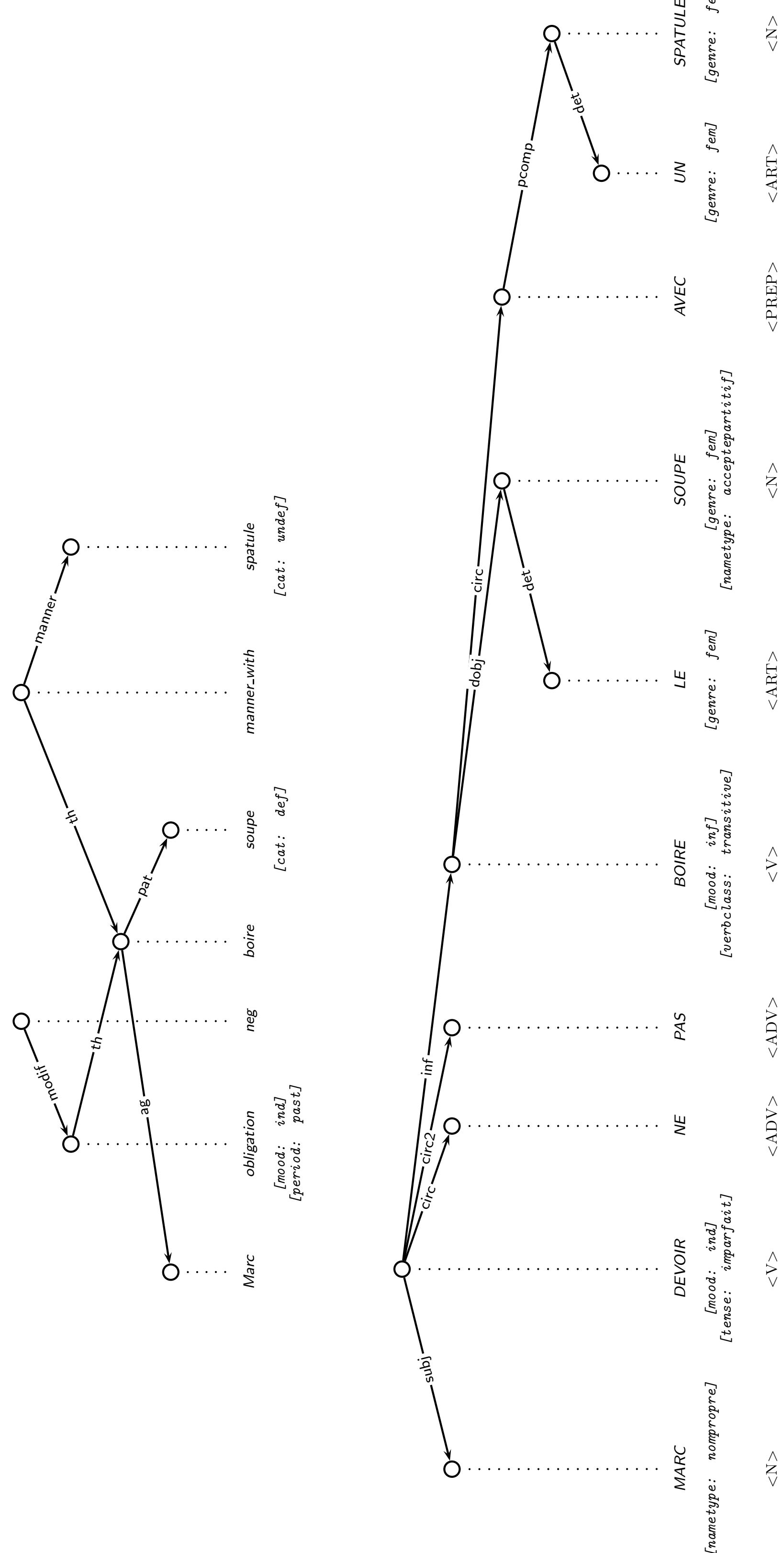
ag
pat
modif
th
ag
locuteur
essayer
[mood: ind]
[period: pres]
[voice: active]
travailler
magn
reference
memoire
[nbre: sg]
subj
inf
pcomp
obl
circ
pcomp
det
JE
ESSAYER
[mood: ind]
[nbre: sg]
[pers: un]
[tense: pres]
DE
TRAVAILLER
[mood: inf]
SUR
DUR
SON
[genre: masc]
[nbre: sg]
MEMOIRE
[genre: masc]
[nbre: sg]
<PRO>
<V>
<PREP>
<>
<PREP>
<ADV>
<ADJPOSS>
<N>

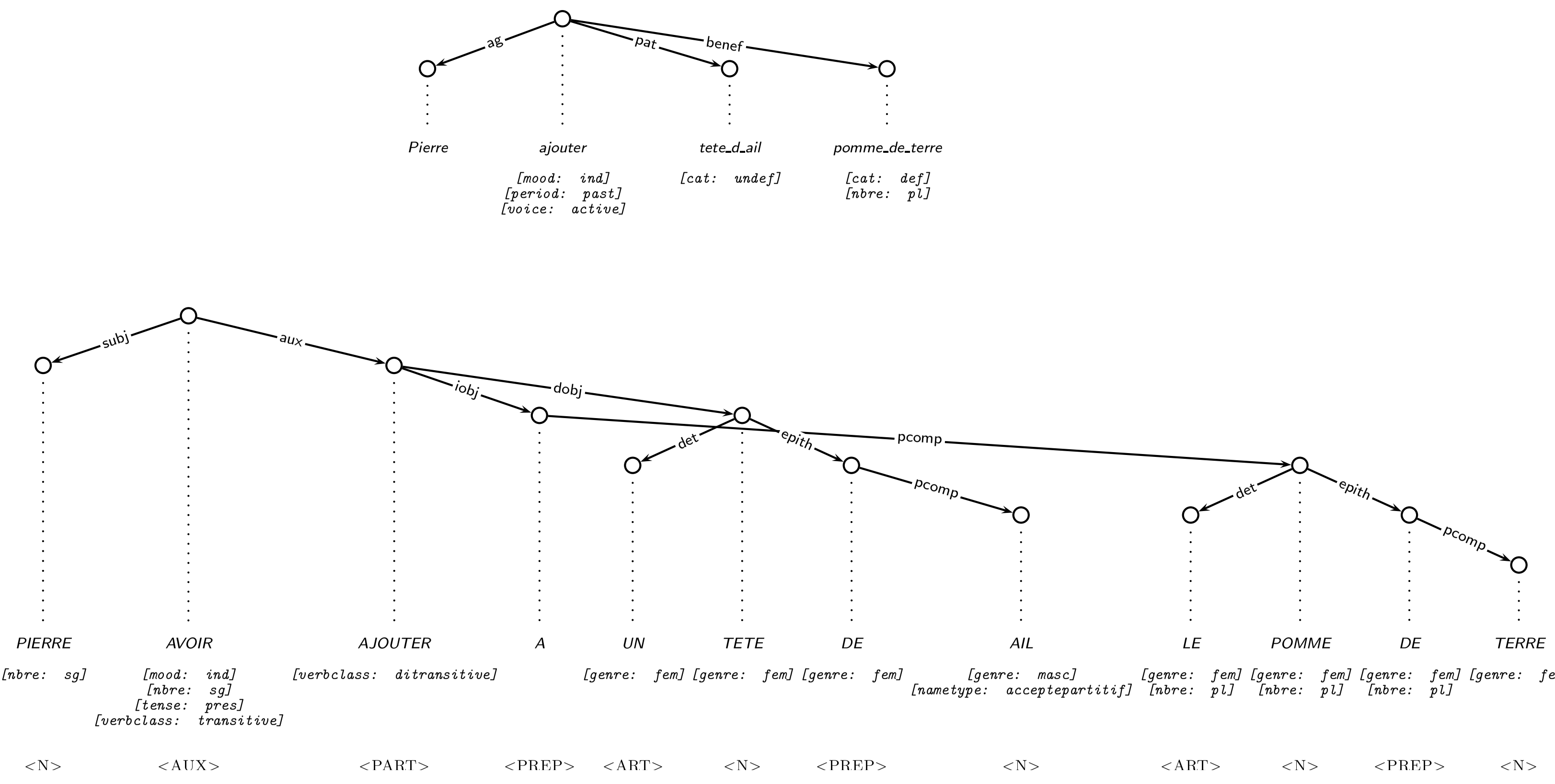


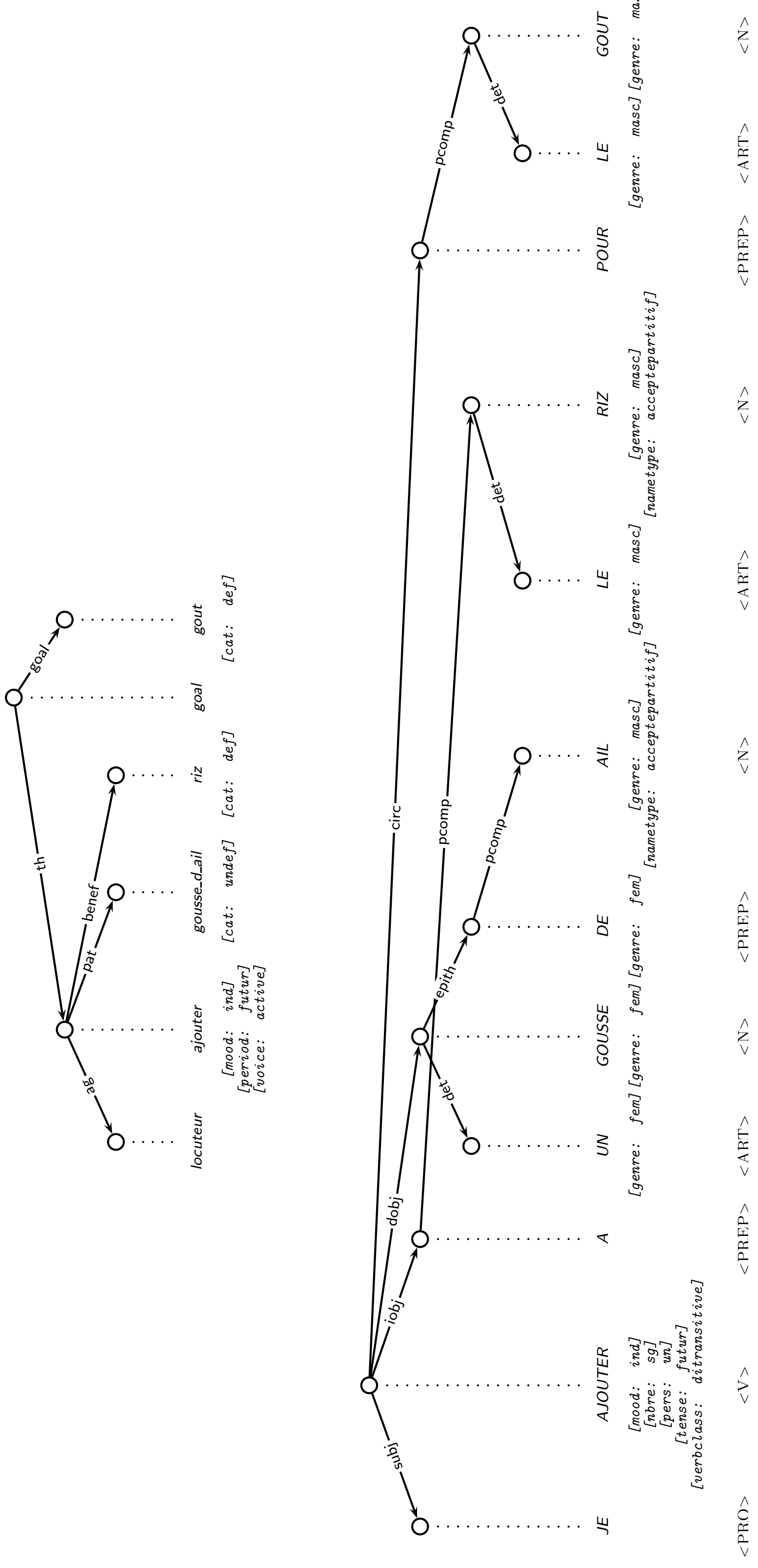
ag
th
ag
Pierre
laisser
[mood: ind]
[period: futur]
[voice: active]
reduire
sauce
[cat: def]
subj
inf
dobj
det
PIERRE
[nbre: sg]
<N>
LAISSER
[mood: ind]
[nbre: sg]
[tense: futur]
<V>
REDUIRE
[mood: inf]
[verbclass: transitive]
<>
LE
[genre: fem]
<ART>
SAUCE
[genre: fem]
[nametype: acceptepartitif]
<N>



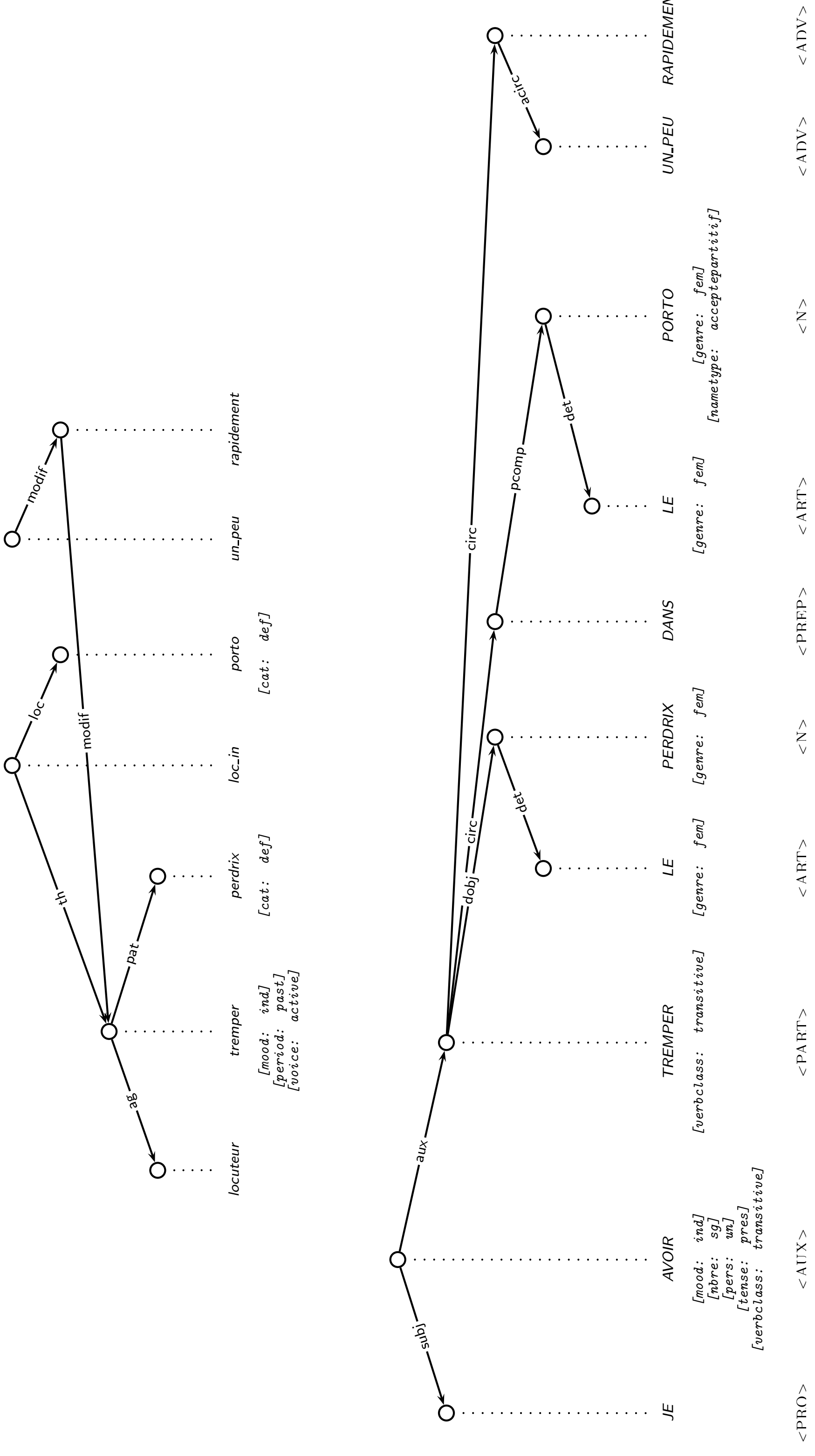
provencal
riz
savoureux
modif
modif
[cat: def]
PROVENCAL
LE
RIZ
ETRE
SAVOUREUX
epith
det
subj
attr
[genre: masc]
[genre: masc]
[genre: masc]
[nametype: acceptepartitif]
[mood: ind]
[tense: pres]
<ADJ>
<ART>
<N>
<V>
<ADJ>

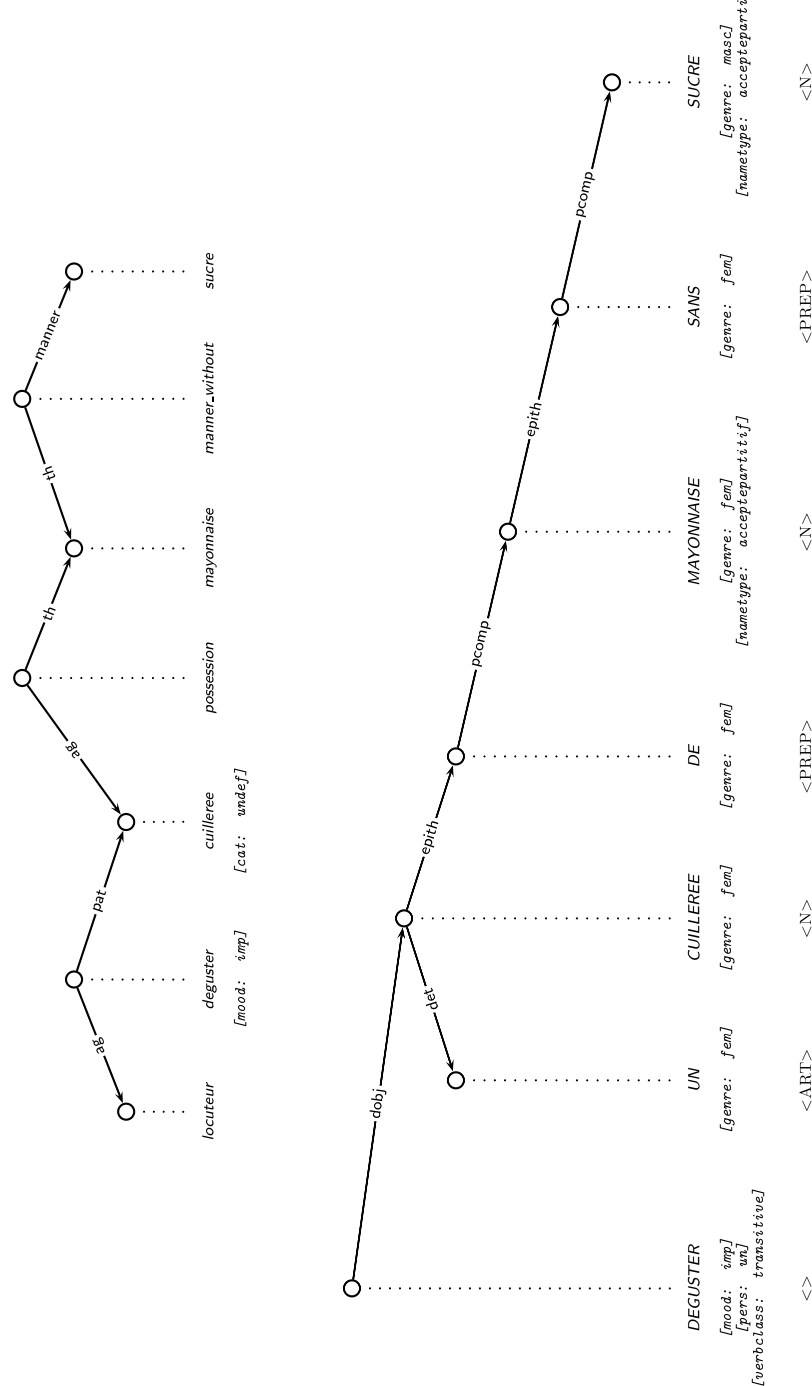

modif
th
ag
pat
th
manner
Marc
obligation
[mood: ind]
[period: past]
neg
boire
soupe
[cat: def]
manner_with
spatule
[cat: undef]
subj
circ
circ2
inf
dobj
circ
det
pcomp
det
MARC
[nametype: nompropre]
<N>
DEVOIR
[mood: ind]
[tense: imparfait]
<V>
NE
<ADV>
PAS
<ADV>
BOIRE
[mood: inf]
[verbclass: transitive]
<V>
LE
[genre: fem]
<ART>
SOUPE
[genre: fem]
[nametype: acceptepartitif]
<N>
AVEC
<PREP>
UN
[genre: fem]
<ART>
SPATULE
[genre: fem]
<N>


ag
pat
benef
Pierre
ajouter
[mood: ind]
[period: past]
[voice: active]
tete_d_ail
[cat: undef]
pomme_de_terre
[cat: def]
[nbre: pl]
subj
aux
iobj
dobj
det
epith
pcomp
pcomp
det
epith
pcomp
PIERRE
[nbre: sg]
<N>
AVOIR
[mood: ind]
[nbre: sg]
[tense: pres]
[verbclass: transitive]
<AUX>
AJOUTER
[verbclass: ditransitive]
<PART>
A
<PREP>
UN
[genre: fem]
<ART>
TETE
[genre: fem]
<N>
DE
[genre: fem]
<PREP>
AIL
[genre: masc]
[nametype: accepteparti tif]
<N>
LE
[genre: fem]
[nbre: pl]
<ART>
POMME
[genre: fem]
[nbre: pl]
<N>
DE
[genre: fem]
[nbre: pl]
<PREP>
TERRE
[genre: fem]
<N>

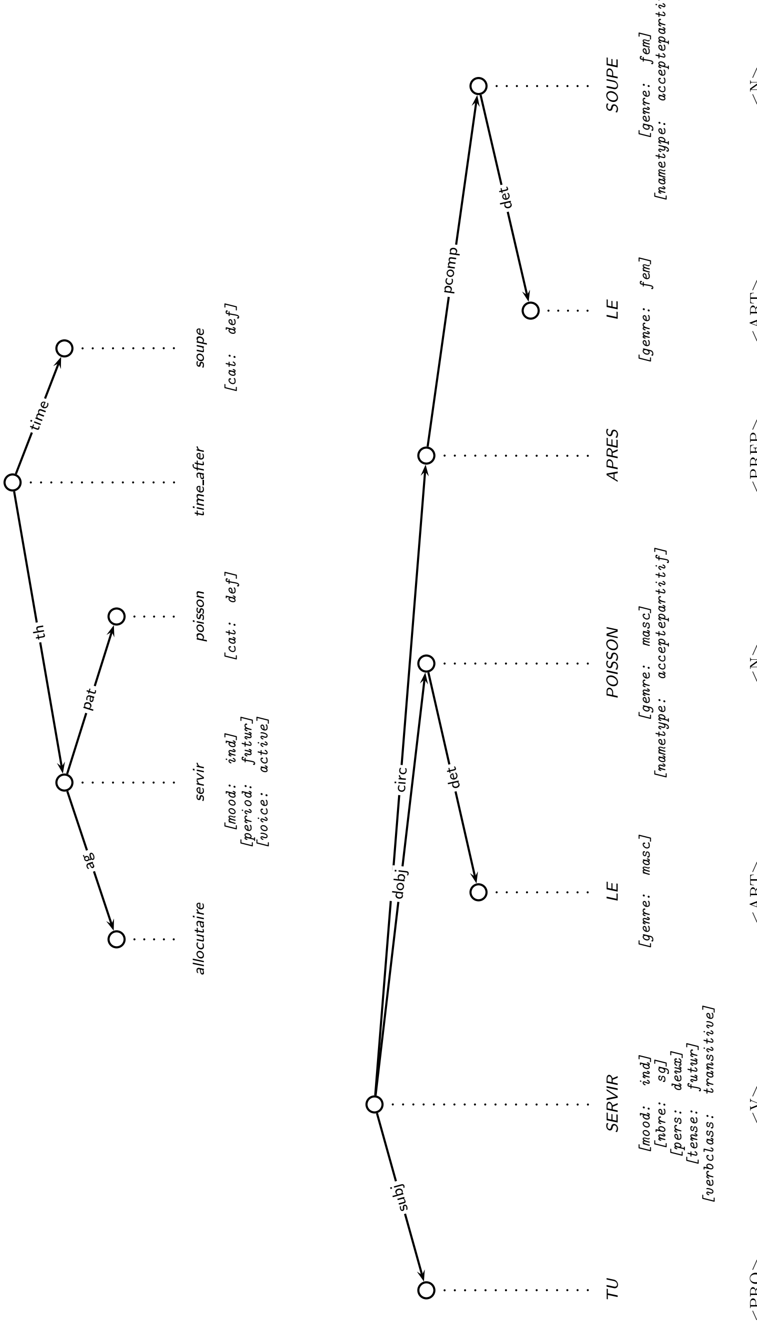


ag
pat
benef
th
goal
locuteur
ajouter
gousse_d_ail
riz
goal
gout
[mood: ind]
[period: futur]
[voice: active]
[cat: undef]
[cat: def]
[cat: def]
subj
iobj
dobj
circ
pcomp
det
epith
pcomp
pcomp
det
pcomp
det
JE
AJOUTER
A
UN
GOUSSE
DE
AIL
LE
RIZ
POUR
LE
GOUT
[mood: ind]
[nbre: sg]
[pers: un]
[tense: futur]
[verbclass: ditransitive]
[genre: fem]
[genre: fem]
[genre: fem]
[genre: masc]
[nametype: accepteparititif]
[genre: masc]
[genre: masc]
[nametype: accepteparititif]
[genre: masc]
[genre: masc]
<PRO>
<V>
<PREP>
<ART>
<N>
<PREP>
<N>
<ART>
<N>
<PREP>
<ART>
<N>

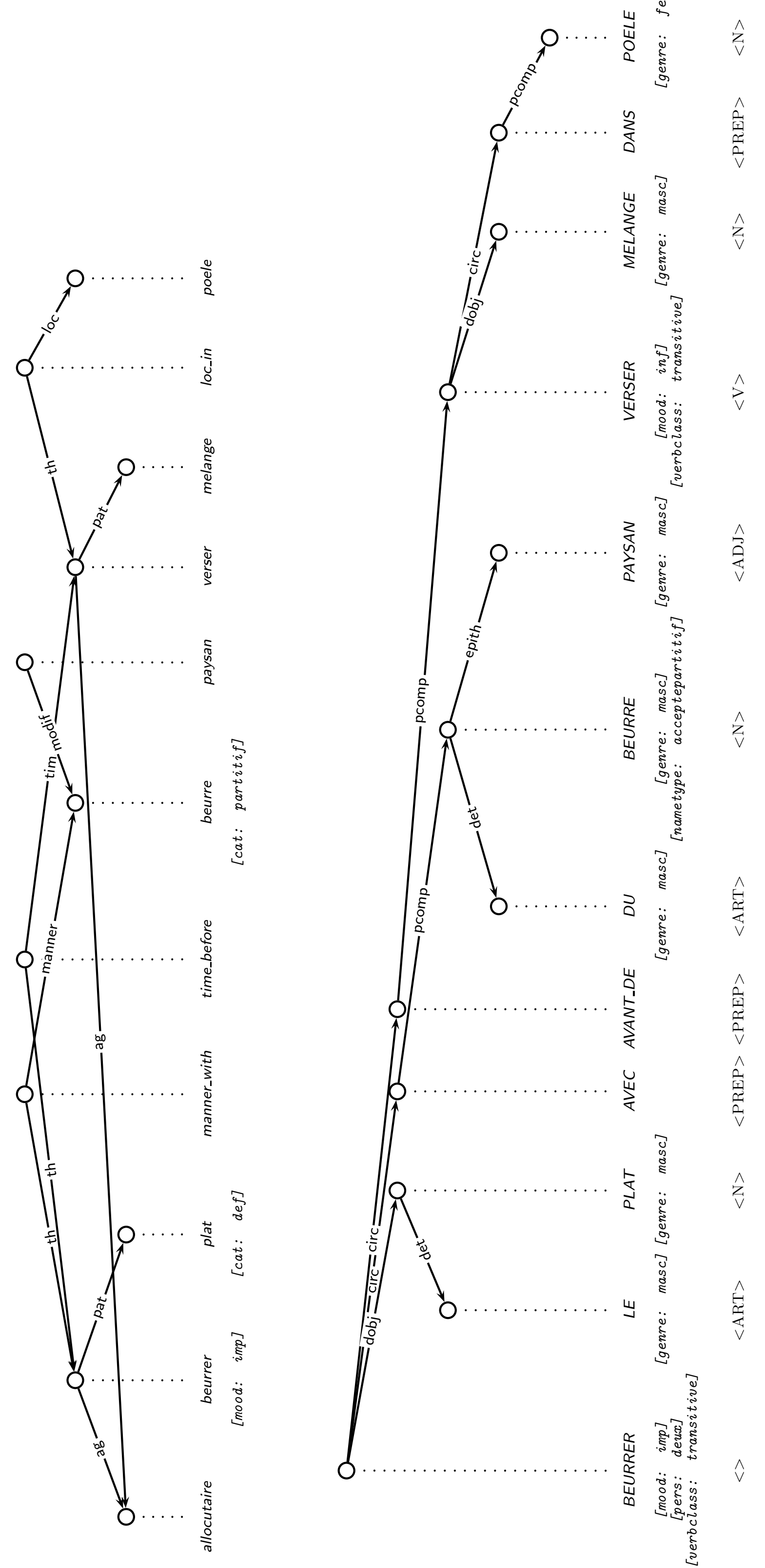

locuteur
tremper
[mood: ind]
[period: past]
[voice: active]
perdrix
[cat: def]
loc_in
porto
[cat: def]
un_peu
rapidement
ag
pat
th
loc
modif
modif
JE
AVOIR
[mood: ind]
[nbre: sg]
[pers: un]
[tense: pres]
[verbclass: transitive]
TREMPER
[verbclass: transitive]
LE
[genre: fem]
PERDRIX
[genre: fem]
DANS
LE
[genre: fem]
PORTO
[genre: fem]
[nametype: acceptepartitif]
UN_PEU
RAPIDEMENT
<PRO>
<AUX>
<PART>
<ART>
<N>
<PREP>
<ART>
<N>
<ADV>
<ADV>
subj
aux
dobj
circ
circ
det
pcomp
det
acirc



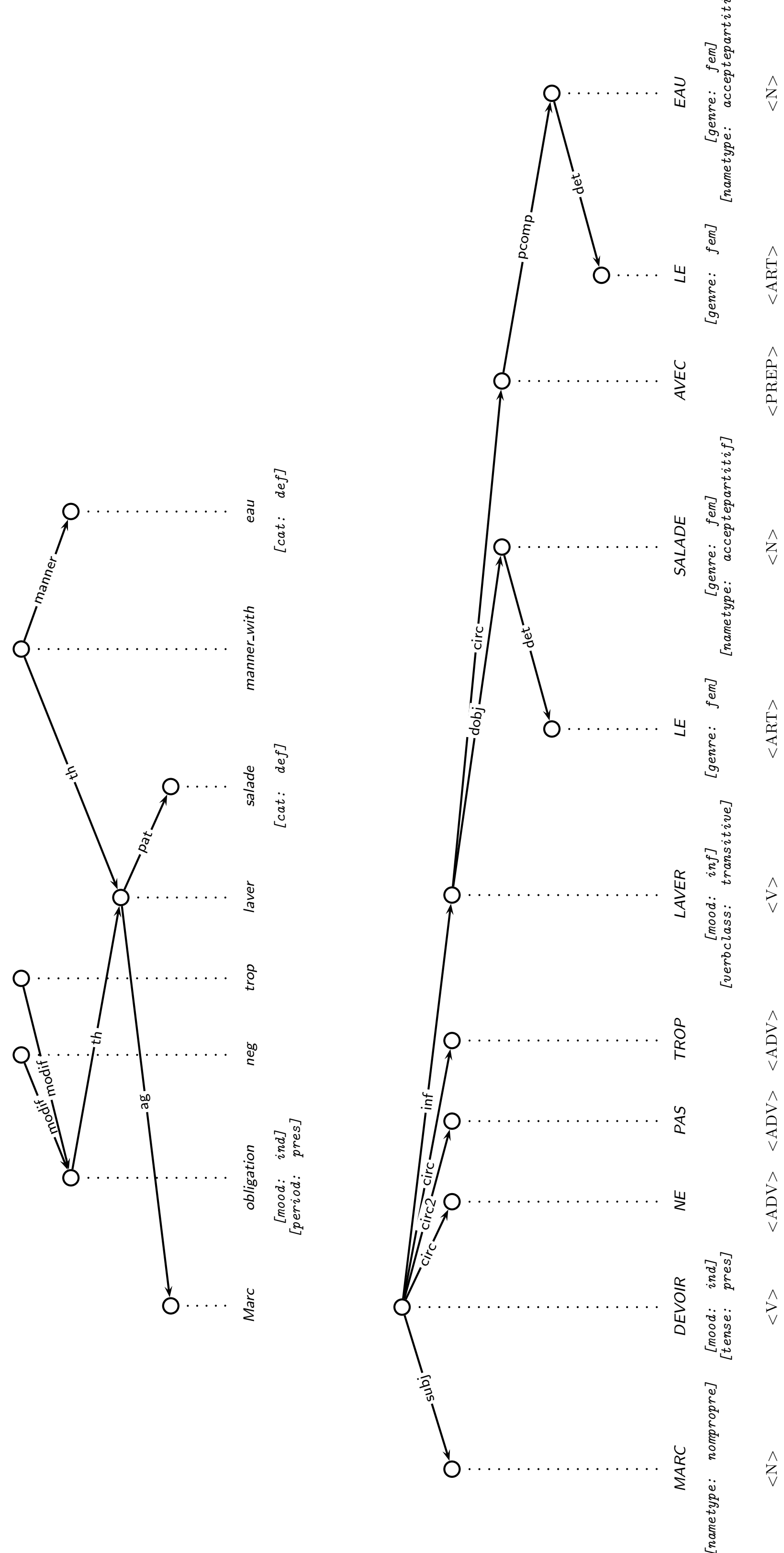
locuteur
deguster
[mood: imp]
cuilleree
[cat: undef]
possession
mayonnaise
manner_without
sucre
ag
pat
ag
th
th
manner
DEGUSTER
[mood: imp]
[pers: un]
[verbclass: transitive]
<>
UN
[genre: fem]
<ART>
CUILLEREE
[genre: fem]
<N>
DE
[genre: fem]
<PREP>
MAYONNAISE
[genre: fem]
[nametype: acceptepartitif]
<N>
SANS
[genre: fem]
<PREP>
SUCRE
[genre: masc]
[nametype: acceptepartitif]
<N>
dobj
det
epith
pcomp
epith
pcomp



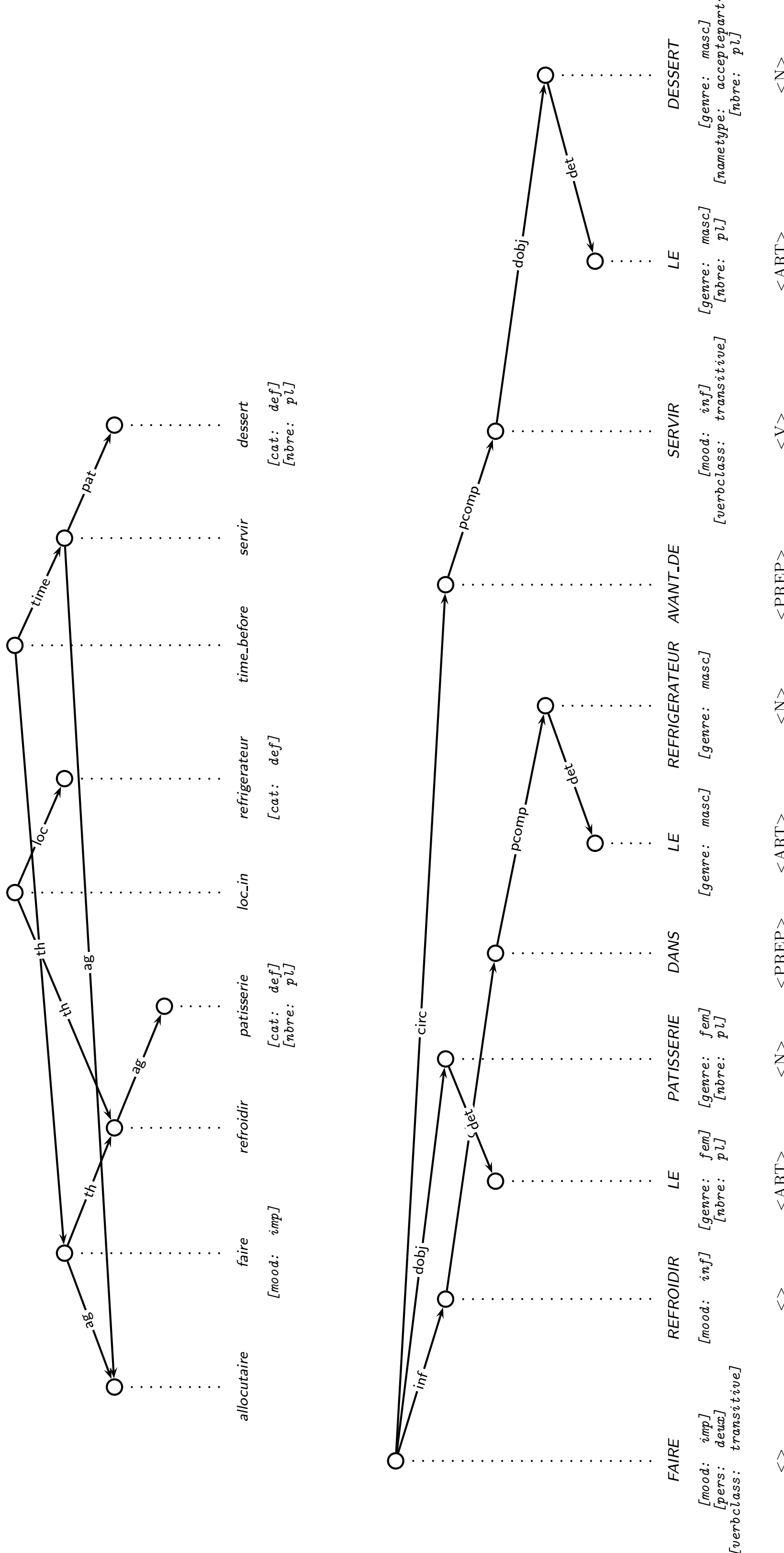
allocutaire
servir
[mood: ind]
[period: futur]
[voice: active]
poisson
[cat: def]
time_after
soupe
[cat: def]
ag
pat
th
time
TU
<PRO>
SERVIR
[mood: ind]
[nbre: sg]
[pers: deux]
[tense: futur]
[verbclass: transitive]
<V>
LE
[genre: masc]
<ART>
POISSON
[genre: masc]
[nametype: accepteparititif]
<N>
APRES
<PREP>
LE
[genre: fem]
<ART>
SOUPE
[genre: fem]
[nametype: accepteparititif]
<N>
subj
dobj
circ
det
pcomp
det

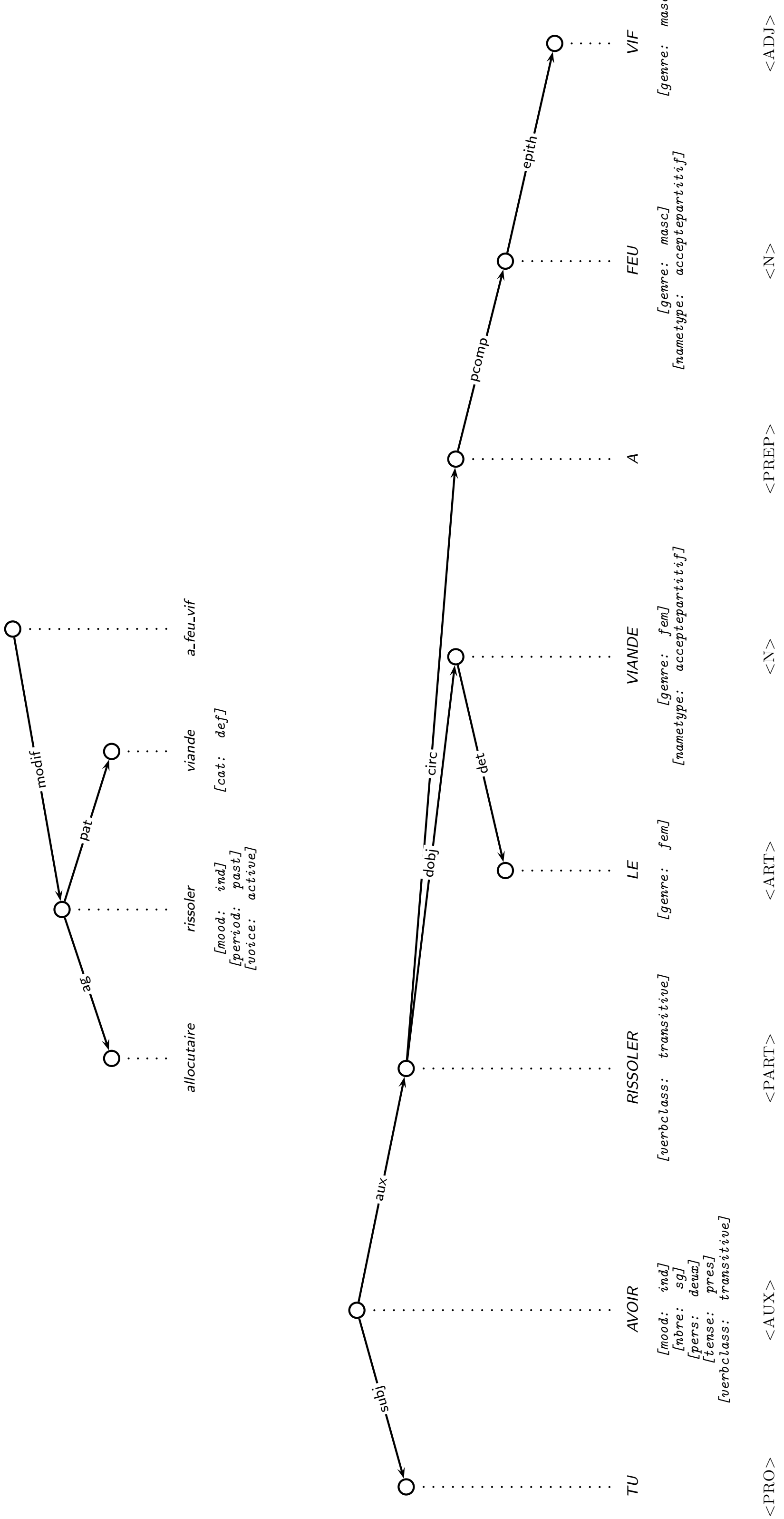


allocutaire
beurrer
[mood: imp]
plat
[cat: def]
manner_with
time_before
beurre
[cat: partitif]
paysan
verser
melange
loc_in
poele
ag
pat
th
manner
tim
modif
loc
BEURRER
[mood: imp]
[pers: deux]
[verbclass: transitive]
LE
[genre: masc]
PLAT
[genre: masc]
AVEC
AVANT_DE
DU
[genre: masc]
BEURRE
[genre: masc]
[nametype: acceptepartitif]
PAYSAN
[genre: masc]
VERSER
[mood: inf]
[verbclass: transitive]
MELANGE
[genre: masc]
DANS
POELE
[genre: fem]
<>
<ART>
<N>
<PREP>
<PREP>
<ART>
<N>
<ADJ>
<V>
<N>
<PREP>
<N>
dobj
circ
det
pcomp
epith

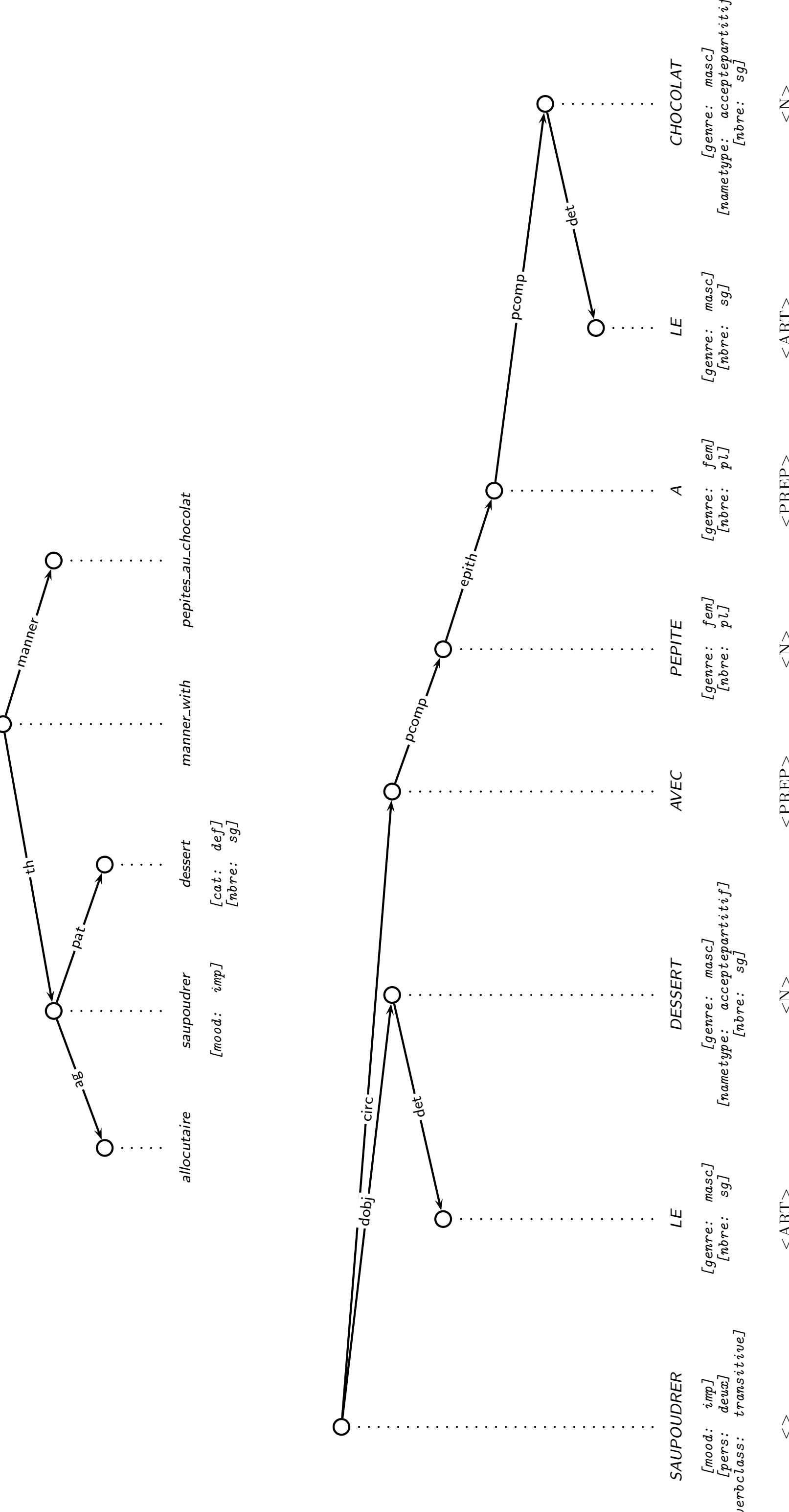


Marc
obligation
[mood: ind]
[period: pres]
neg
trop
laver
salade
[cat: def]
manner_with
eau
[cat: def]
modif
modif
th
ag
pat
th
manner
MARC
[nametype: nompropre]
<N>
DEVOIR
[mood: ind]
[tense: pres]
<V>
NE
<ADV>
PAS
<ADV>
TROP
<ADV>
LAVER
[mood: inf]
[verbclass: transitive]
<V>
LE
[genre: fem]
<ART>
SALADE
[genre: fem]
[nametype: acceptepartitif]
<N>
AVEC
<PREP>
LE
[genre: fem]
<ART>
EAU
[genre: fem]
[nametype: acceptepartitif]
<N>
subj
circ
circ2
circ
inf
dobj
circ
det
pcomp
det

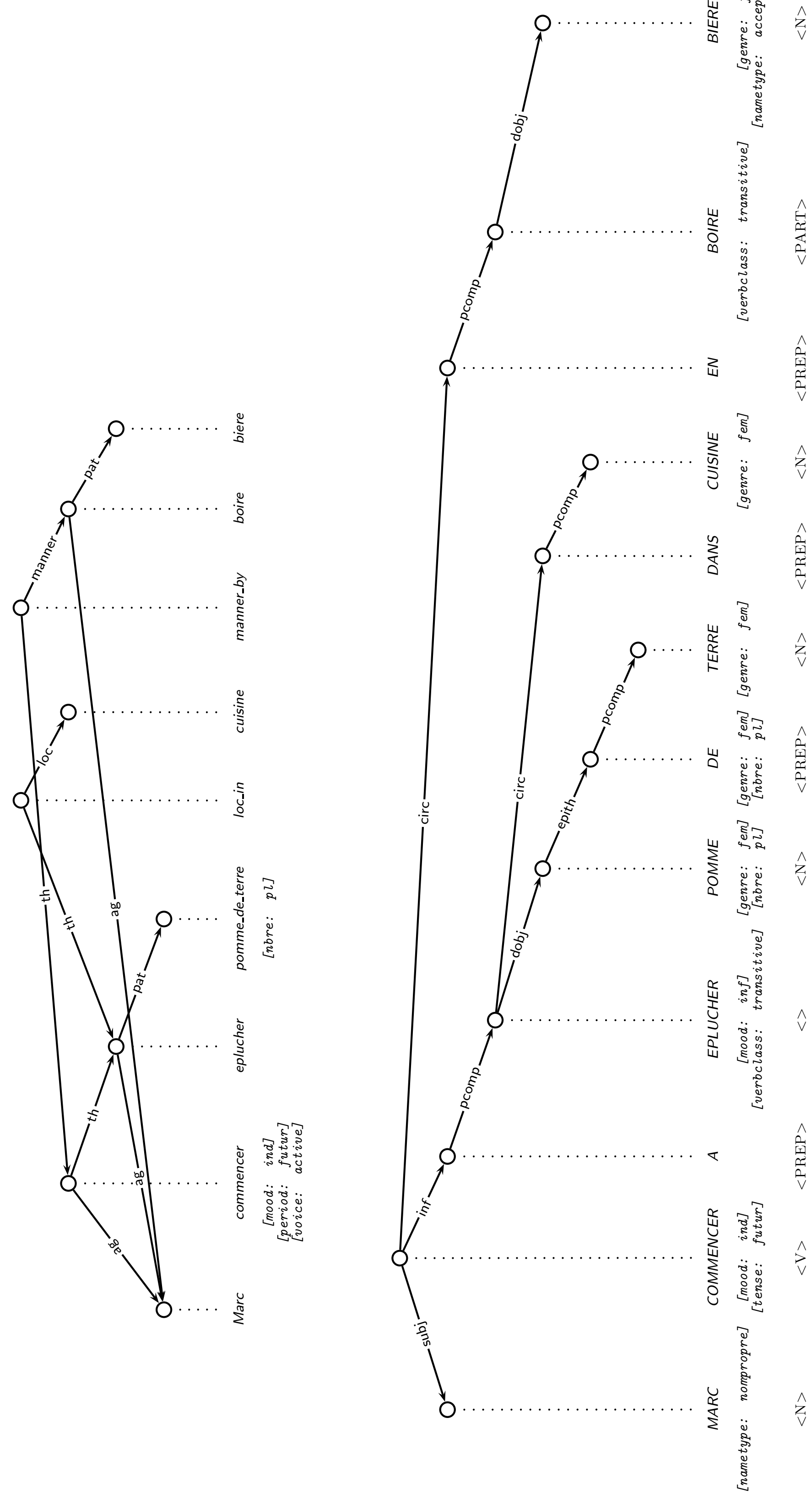

allocutaire
faire
[mood: imp]
refroidir
patisserie
[cat: def]
[nbre: pl]
loc_in
refrigerateur
[cat: def]
time_before
servir
dessert
[cat: def]
[nbre: pl]
ag
th
th
ag
th
loc
ag
time
pat
FAIRE
[mood: imp]
[pers: deux]
[verbclass: transitive]
<>
REFROIDIR
[mood: inf]
<>
LE
[genre: fem]
[nbre: pl]
<ART>
PATISSERIE
[genre: fem]
[nbre: pl]
<N>
DANS
<PREP>
LE
[genre: masc]
<ART>
REFRIGERATEUR
[genre: masc]
<N>
AVANT_DE
<PREP>
SERVIR
[mood: inf]
[verbclass: transitive]
<V>
LE
[genre: masc]
[nbre: pl]
<ART>
DESSERT
[genre: masc]
[nametype: accepteparti
[nbre: pl]
<N>
inf
dobj
circ
det
pcomp
det
pcomp
dobj
det



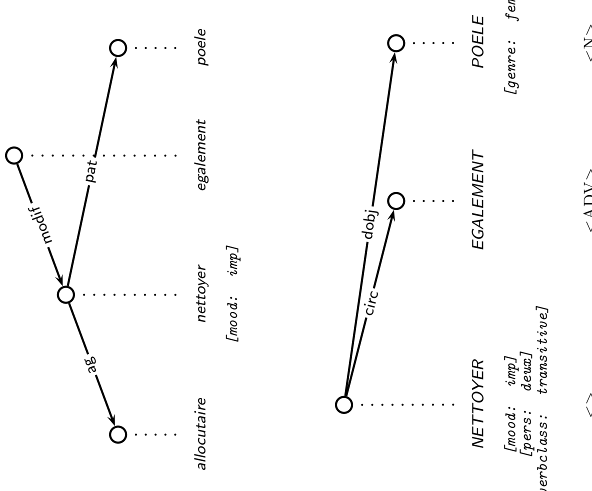
allocutaire
rissoler
[mood: ind]
[period: past]
[voice: active]
viande
[cat: def]
a_feu_vif
ag
pat
modif
TU
<PRO>
subj
AVOIR
[mood: ind]
[nbre: sg]
[pers: deux]
[tense: pres]
[verbclass: transitive]
<AUX>
aux
RISSOLER
[verbclass: transitive]
<PART>
dobj
circ
LE
[genre: fem]
<ART>
det
VIANDE
[genre: fem]
[nametype: accepteparititif]
<N>
A
<PREP>
pcomp
FEU
[genre: masc]
[nametype: accepteparititif]
<N>
epith
VIF
[genre: masc]
<ADJ>



ag
pat
th
manner
allocutaire
saupoudrer
[mood: imp]
dessert
[cat: def]
[nbre: sg]
manner_with
pepites_au_chocolat
dobj
circ
det
pcomp
epith
pcomp
det
SAUPOUDRER
[mood: imp]
[pers: deux]
[verbclass: transitive]
<>
LE
[genre: masc]
[nbre: sg]
<ART>
DESSERT
[genre: masc]
[nametype: acceptepartitif]
[nbre: sg]
<N>
AVEC
<PREP>
PEPITE
[genre: fem]
[nbre: pl]
<N>
A
[genre: fem]
[nbre: pl]
<PREP>
LE
[genre: masc]
[nbre: sg]
<ART>
CHOCOLAT
[genre: masc]
[nametype: acceptepartitif]
[nbre: sg]
<N>



Marc
commencer
[mood: ind]
[period: futur]
[voice: active]
eplucher
pomme_de_terre
[nbre: pl]
loc_in
cuisine
manner_by
boire
biere
ag
th
pat
loc
manner
MARC
[nametype: nompropre]
<N>
COMMENCER
[mood: ind]
[tense: futur]
<V>
A
<PREP>
EPLUCHER
[mood: inf]
[verbclass: transitive]
<>
POMME
[genre: fem]
[nbre: pl]
<N>
DE
[genre: fem]
[nbre: pl]
<PREP>
TERRE
[genre: fem]
<N>
DANS
<PREP>
CUISINE
[genre: fem]
<N>
EN
<PREP>
BOIRE
[verbclass: transitive]
<PART>
BIERE
<N>
subj
inf
pcomp
dobj
epith
circ


modif
pat
ag
allocutaire
nettoyer
[mood: imp]
egalement
poele
circ
dobj
NETTOYER
[mood: imp]
[pers: deux]
[verbclass: transitive]
<>
EGALEMENT
<ADV>
POELE
[genre: fem]
<N>

Annexe C

Extraits du code source

Vu la taille de notre implémentation, il nous est impossible de lister l'entièreté de notre code source (qui totalise plus de 150 pages A4). Nous avons donc choisi de présenter 6 fichiers, dont 3 appartiennent au compilateur (écrit en Python, donc), et 3 font partie des nouvelles contraintes XDG (programmées en Oz) :

1. `PUGParser.py` : Analyseur de type LR utilisé pour extraire d'un fichier texte l'ensemble de notre grammaire GUST/GUP ;
2. `Compiler.py` : Compilateur de grammaires GUST/GUP en grammaires XDG ;
3. `CompileTests.py` : Conversion et compilation de la batterie de tests ;
4. `GUSTSemEdgeConstraints.oz` : Contrainte assurant la saturation sémantique complète des arcs via des règles d'interface ;
5. `GUSTSagittalConstraints.oz` : Contrainte assurant le respect des règles sagittales (i.e. saturation des arcs propres à une dimension) ;
6. `GUSTLinkingConstraints.oz` : Contrainte chargée de traiter les règles d'interface.

Le lecteur intéressé par le détail de notre implémentation pourra évidemment retrouver l'ensemble de notre code source dans notre logiciel, disponible pour rappel à l'adresse suivante : `http://ece.fsa.ucl.ac.be/plison/memoire/auGUSTe.zip`

C.1 PUGParser.py

```
#!/usr/bin/python
# -*- coding: iso8859-15 -*-
#
# LR Parser of PUG specifications (see Pierre Lison's 2006
# M.Sc. thesis for details on this format and its use)
#
# The BNF rules is included in the function definitions.
#
# Important note: this module imports PUGLexer,
# which defines all the admissible lexical components.
#
# Author: Pierre Lison
# Last Modif: 20 August 2006


import os,lex, yacc, sys
from PUGLexer import tokens, buildLexer
from Constants import *

def p_start(p):
```

```
    p[0] = p[1]

def p_tests1(p):
    'tests : tests NAME LBRA definition RBRA'
    dico = {p[2]:p[4]}
    if isinstance(p[1],dict):
        dico2 = p[1]
        dico.update(dico2)
        p[0] = dico
    else:
        p[0] = dico

def p_grammar(p):
    'grammar : NAME dimension NAME dimension NAME dimension'
    global grammar
    p[0] = {p[1]:p[2], p[3]:p[4], p[5]:p[6]}


def p_tests2(p):
    'tests : empty'
    pass

# Error rule for syntax errors
def p_error(p):
    print "Syntax error in input!"
    print p
    sys.exit()

def p_dimension(p):
    'dimension : LBRA rules RBRA'
    p[0] = p[2]

def p_rules_1(p):
    'rules : rules NAME LBRA definition RBRA'
    dico = {p[2]:p[4]}
    if isinstance(p[1],dict):
        dico2 = p[1]
        dico.update(dico2)
        p[0] = dico
    else:
        p[0] = dico


def p_rules_2(p):
    'rules : empty'
    pass

def p_definition(p):
    'definition : part1 part2'
    dico = {}
    for i in p[1]:
     #   print i
        dico[i['id']] = {}
        dico[i['id']]['type']=i['type']
        for j in p[2]:
            if (j['arg'] == i['id']):
                dico[i['id']][j['function']] = j['value']
    p[0] = dico

def p_part1_1(p):
```

```
    'part1 : part1 NUM COLON TYPE'
    val = {}
  # print p[2]
    val['id'] = p[2]
    val['type'] = p[4]
    if isinstance(p[1],list):
        p[1].append(val)
        p[0] = p[1]
    else:
        p[0] = [val]

def p_part1_2(p):
    'part1 : empty'
    pass

def p_part2(p):
    'part2 : part2 NAME LPAR NUM RPAR EQU assign'
    val = {}
    val['function'] = p[2]
    val['arg']=p[4]
    val['value']=p[7]
    if isinstance(p[1],list):
        p[1].append(val)
        p[0] = p[1]
    else:
        p[0]=[val]

def p_part2_2(p):
    'part2 : empty'
    pass

def p_assign(p):
    '''assign : POL
    | STR
    | NUM'''
    p[0] = str(p[1])

def p_assign_list(p):
    '''assign : LBRA list RBRA'''
    p[0] = p[2]

def p_str_list(p):
    '''list : list STR '''
    p[1].append(p[2])
    p[0] = p[1]

def p_str_list2(p):
    'list : empty'
    p[0] = []

def p_empty(p):
    'empty :'
    pass

def parseGrammar (data):
    p_start.__doc__ = 'start : grammar'
    return parse (data)

def parseTests (data):
    p_start.__doc__ = 'start : tests'
```

```
    return parse (data)

def parse(data):
    '''Parse the string data according to the BNF rules,
    and returns the result'''

    print
    print "Begin parse of PUG file..."
    buildLexer (data)
    print "... Lexical analysis OK"
    # Build the parser
    yacc.yacc()
    result = yacc.parse(data)
    print "... Syntactic analysis OK"
    if os.path.exists ("parsetab.py"):
        os.remove ("parsetab.py")
    print "Parsing successful"
    #print result
    return result
```

C.2 Compiler.py

```
#!/usr/bin/python
# -*- coding: iso8859-15 -*-
#
# XDG Compiler of MTUG/PUG grammars
# (see Pierre Lison's 2006 M.Sc. thesis for details)
#
# Author: Pierre Lison
# Last modif: 20th August 2006

import PUGParser
from XDGTextConstants import *
from ClassProcessing import *
from LinkingProcessing import *
from XDGTextGeneration import *
from Utils import *
from NodeProcessing import *
import os, Constants


# A MTUG/ PUG compiler
class Compiler:

    def compileGrammar(self, g):
        ''' Given a grammar g correctly parsed from the PUGParser module,
        returns a dictionnary object containing the compiled grammar,
        made of 9 data structures:
        1)   All the defined classes ;
        2)   All the lexical units ;
        3-4) Their identifiers on each dimension ;
        5-6) Edge labels on each dimension ;
        7-8) Admissible grammenes for each dimension ;
        9)   Part-of-speechs. '''

        # Verify well-formedness
        grammarOK = self.verifyWellFormedness (g)
```

```
    if grammarOK:
        print
        print "Begin compilation of grammar file ..."

        # The lexical classes
        classes = {}

        # Add some basic classes
        classes.update ({'Root-isemsynt':{'dim':'isemsynt'}})
        classes.update({'deleted-gsem':{'dim':'gsem', 'name':'', 'in':'
            del!'}})
        classes.update({'deleted-gsynt':{'dim':'gsynt', 'name':'', 'in'
            :'del!'}})

        # Extract the semantic classes
        sem_classes = extractClasses (g, 'gsem')
        sem_classes = cleanClasses (sem_classes)
        print "... Semantic classes extraction OK"

        # Extract the syntactic classes
        synt_classes = extractClasses (g, 'gsynt')
        synt_classes = cleanClasses (synt_classes)
        print "... Syntactic classes extraction OK"

        # Extract the semantic units
        sem_units = extractUnits(g, 'gsem', sem_classes)
        sem_labels = extractLabels(g, 'gsem')
        print "... Semantic units extraction OK"

        # Extract the syntactic units
        synt_units = extractUnits(g, 'gsynt', synt_classes)
        synt_labels = extractLabels(g, 'gsynt')
        print "... Syntactic units extraction OK"

        # Add the semantic and syntactic classes to the set
        classes.update(sem_classes)
        classes.update(synt_classes)

        # Extract the linking classes and add them to the set
        classes = extractClassesLinking (g, classes, 'gsem', 'gsynt', '
            isemsynt')
        classes = cleanClasses(classes)
        print "... Linking classes extraction OK"

        # Extract the linking of lexical units
        linking = extractLinking (g, 'gsem', 'gsynt', 'isemsynt')
        print "... Linking units extraction OK"

        # Search for deleted nodes on both dimensions
        delNodes_synt = getLinkingDeletedNodes (g, synt_units, 'gsem',
            'gsynt', 'isemsynt')
        delNodes_sem = getLinkingDeletedNodes (g, sem_units, 'gsynt', '
            gsem', 'isemsynt')
        print "... Deleted nodes extraction OK"

        # Extract the grammenes values and part of speech
        sem_grams = extractGramValues (g, 'gsem')
        synt_grams = extractGramValues (g, 'gsynt')
        POS = getAllPOS (g, 'gsynt')
        print "... Grammenes and POS values extraction OK"
```

```
        # Preprocess the lexical Units
        Units = XDGUnitsPreprocessing (sem_units, synt_units, linking,
            [])
        print "... XDG units preprocessing OK"

        for delNode in delNodes_synt + delNodes_sem:
            Units.append (delNode)

        # Get the identifiers (unique id for lexical units)
        (identifiers_gsem, identifiers_gsynt) = getIdentifiers (Units)

        # Gather all the results in one dictionnary object
        result = {'Units': Units, \
                  'classes': classes, \
                  'identifiers_gsem':identifiers_gsem, '
                      identifiers_gsynt':identifiers_gsynt, \
                  'sem_labels': sem_labels, 'synt_labels':synt_labels,
                      \
                  'sem_grams':sem_grams, 'synt_grams':synt_grams, \
                  'POS':POS }
    else:
        print "ERROR: The given grammar is incorrect and has not been
            compiled"
        sys.exit()

    print "Grammar successfully compiled \n"
    return result


####################


def writeGrammar (self, g_compiled, RelativeXDK, XDGGrammarName):
    ''' Given a compiled PUG grammar and a file path, write the
    corresponding XDG grammar to the file in the UL format'''

    print
    print "Begin write of XDG grammar file ..."
    sem_labels = g_compiled['sem_labels']
    synt_labels = g_compiled['synt_labels']
    sem_grams = g_compiled['sem_grams']
    synt_grams = g_compiled['synt_grams']
    identifiers_gsem = g_compiled['identifiers_gsem']
    identifiers_gsynt = g_compiled['identifiers_gsynt']
    POS = g_compiled['POS']
    Units = g_compiled['Units']
    classes = g_compiled['classes']

    # Generate the XDG preamble
    XDGPreamble = generateXDGPreamble(sem_labels, synt_labels,
        sem_grams, synt_grams, POS, \
                                        identifiers_gsem,
                                            identifiers_gsynt)
    print "... Preamble generation OK"

    # Generate the XDG classes specifications
    XDGClasses = generateXDGClasses (classes)
    print "... Classes generation OK"
```

```
        # Generate the XDG lexical units specifications
        XDGLexUnits = generateXDGLexUnits (Units)
        print "... Lexical units generation OK"

        # Write everything to the XDG grammar file
        XDGrammar = XDGPreamble + XDGClasses + XDGLexUnits
        f = open (RelativeXDK + "Grammars%s"%(os.sep) + XDGGrammarName, 'w'
            )
        f.write (XDGrammar)
        f.close
        print "... XDG file writing OK"

        f = open (RelativeXDK + "Grammars%s"%(os.sep) + XDGGrammarName, 'r'
            )
        f.close()

        print "XDG grammar file \'%s\' successfuly written\n"%
            XDGGrammarName


    ####################


    def finalizeCompilation (self, g_compiled, RelativeXDK, XDGGrammarName,
        PUGGrammarName):
        ''' Given a compiled PUG grammar and adequate file paths, performs
        two operations:
        1) create and compile a GUSTAlterGrams.oz constraint file
        describing the admissible grammenes (that is, the features
        and their corresponding feature values)
        2) Run the GUSTEmptyNodes.oz script which is used to count the
        maximal expansion of every semantic unit (in order to handle
        multiword expressions) and write the result to a file'''

        sem_grams = g_compiled['sem_grams']
        synt_grams = g_compiled['synt_grams']

        print "Finalize compilation...."
        f = open (RelativeXDK + Constants.ConstraintLibPath + Constants.
            GUSTAlterGramsFile, 'w')

        # Extract the semantic grammenes values
        semGramsValues = []
        for semFeat in sem_grams.keys():
            for semVal in sem_grams[semFeat]:
                if semVal not in semGramsValues:
                    semGramsValues.append (semVal)

        # Extract the syntactic grammenes values
        syntGramsValues = []
        for syntFeat in synt_grams.keys():
            for syntVal in synt_grams[syntFeat]:
                if syntVal not in syntGramsValues:
                    syntGramsValues.append (syntVal)

        # Generate the GUSTAlterGrams text
        AlterGramsText = GUSTAlterGrams % (str(sem_grams.keys()).replace ('
            \'', '').replace (',', ''), \
                                            str(semGramsValues).replace ('\''
                                                , '').replace (',', ''), \
```

```
                                        str(synt_grams.keys()).replace (
                                            '\'', '').replace (',', ''),
                                        str(syntGramsValues).replace ('
                                            \'', '').replace (',', ''), \
                                        str(synt_grams).replace('\'','')
                                            .replace(',', '').replace('{'
                                            ,'').replace('}',''),\
                                        str(sem_grams).replace('\'', '')
                                            .replace(',', '').replace('{'
                                            ,'').replace('}',''))

    # Write the text to the constraint file
    f.write (AlterGramsText)
    f.close()
    GAGfile = RelativeXDK + Constants.ConstraintLibPath + Constants.
        GUSTAlterGramsFile

    # Compile the constraint file
    os.system ("ozc -c %s -o %s -q"%(GAGfile, GAGfile+"f"))
    print "... Grammenes constraint integration OK"

    # Run the GUSTEmptyNodesCounter script
    Grammar = RelativeXDK + "Grammars%s"%(os.sep) + XDGGrammarName
    Command = "ozengine %s -p \"%s\" -g \"%s\" -o \"%s\""\
              %(Constants.EmptyNodesCounter, \
                RelativeXDK.replace('\\','/'), \
                XDGGrammarName, \
                PUGGrammarName.replace('.txt', '_emptyNodes.txt'))
    os.system(Command)

    print "... Empty nodes counter OK"

    ans = raw_input ("\nDo you want to use some optimization heuristics
         in order to improve the speed of the grammar? [y/n] ")

    while ans != 'y' and ans != 'n':
        print "Wrong choice, please type \'y\' or \'n\'"
        ans = raw_input ("Do you want to use some optimization
            heuristics in order to improve the speed of the grammar? [y/
            n] ")


    if ans =='y':
        self.optimizationHeuristics (PUGGrammarName.replace('.txt', '
            _emptyNodes.txt'))
        print "... Optimization heuristics for empty nodes OK"


    print "Compilation successfully finalized\n"


####################


def optimizationHeuristics(self, emptyNodeFile):
    '''Small heuristics (only usable for the validationGrammar) geared
        at reducing the
    number of empty nodes, in order to improve the efficiency'''
    f1 = file (emptyNodeFile, 'r')
    text = f1.read()
```

```
        f1.close()
        f2 = file (emptyNodeFile, 'w')
        modifiedText = text.replace ("4\t0", "1\t0") # Optimization of
            names with determiners
        modifiedText = modifiedText.replace ("2\t2", "1\t1") # Optimisation
             of transitive verbs
        f2.write (modifiedText)


    ####################


    def verifyWellFormedness (self, g):
        ''' Performs a bare check of the grammar in order
        to verify the wellformedness of the grammar'''
        for dim in g.keys():
            dim_data = g[dim]
            for rule in dim_data.keys():
                rule_data = dim_data[rule]
                for obj in rule_data.keys():
                    obj_data = rule_data[obj]
                    if not obj_data.has_key('p'):
                        print ("ERROR: object %s of rule %s of dimension %s
                             has no polarity") \
                        % (obj, rule, dim)
                        return False
        return True


    ####################


    def printing(self, sem_units, sem_labels, sem_classes, synt_units,
       synt_labels, synt_classes, linking):
        '''Print various results about the grammar'''

        print "\n——— Résultats de la compilation de la grammaire de test
           ———"
        #print g
        print "\nSEMANTIQUE:"
        print "Sémantèmes:",
        print sem_units.keys()
        print "Rôles Sémantiques",
        print sem_labels
        print "Classes:",
        print sem_classes.keys()
        print "\nSYNTAXE:"
        print "Lexèmes",
        print synt_units.keys()
        print "Fonctions Syntaxiques:",
        print synt_labels
        print "Classes::",
        print synt_classes.keys()

        print "\nINTERFACE SEMANTIQUE SYNTAXE"
        print linking
        print '\n\n————————————'
```

C.3 CompileTests.py

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
#
# Converter and compiler for PUG test suites
#
# Author: P. Lison
# Last modif: 20th August

import PUGParser, Dia2GUST_gsem, os
import sys, Constants
from Utils import *
from XDGTextConstants import *

###########################

def getDaughters (graph, node_id):
    'Returns the daughters of a node in a given graph'
    daughters = []
    for obj in graph.keys():
        if 'edge' in graph[obj]['type']:
            if graph[obj]['source'] == node_id:
                daughters.append ((graph[obj]['label'], graph[obj]['target'
                    ]))
    return daughters

###########################

def getMothers (graph, node_id):
    'Returns the mothers of a node in a given graph'
    mothers = []
    for obj in graph.keys():
        if 'edge' in graph[obj]['type']:
            if graph[obj]['target'] == node_id:
                mothers.append ((graph[obj]['label'], graph[obj]['source'])
                    )
    return mothers

###########################

def getLabels (graph):
    'Returns all the labels used in a given graph'
    labels = []
    for obj in graph.keys():
        if 'edge' in graph[obj]['type']:
            labels.append (graph[obj]['label'].strip("'"))
    return labels

def getNodes (graph):
    'Returns all the nodes in a given graph'
    nodes = []
    for obj in graph.keys():
        if 'node' in graph[obj]['type']:
            nodes.append (obj)
    return nodes

###########################

def getPreamble ():
```

```
    'Return the preamble used in the input graph constraints'

    text1 = """
%% Copyright 2001-2006
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%    Denys Duchier <duchier@ps.uni-sb.de> (LORIA, Nancy)
%%
%% Principle written by Pierre Lison <plison@agora.eu.org>
%% on 26/02/06
functor
import
    FS
    GUSTAlterGrams at '../GUSTAlterGrams.ozf'

export
    fixInputGraph: FixInputGraph
define
    proc {FixInputGraph Nodes}
        for Node in Nodes do
"""
    return text1

##########################

def listStrip(list):
    list2 = str(list).replace("[","")
    list2 = list2.replace("]","")
    list2 = list2.replace(",","")
    list2 = list2.replace("\'","")
    return list2

##########################

def compactConstraintStructure (constraints):
    'Returns a compacted version of the constraint structure'

    hasBeenModified = False
    compactedConstraints = constraints[:]
    for c1 in compactedConstraints:
        compactedConstraintsWithC1Excluded = compactedConstraints[:]
        compactedConstraintsWithC1Excluded.remove (c1)
        for c2 in compactedConstraintsWithC1Excluded:
            if c1[0] == c2[0] and c1[1] == c2[1] and c1[2] == c2[2]:
                if c1 in compactedConstraints and c2 in
                    compactedConstraints:
                     compactedConstraints.remove (c1)
                     compactedConstraints.remove (c2)
                     compactedConstraints.append ((c1[0], c1[1], c1[2], c1
                         [3] + c2[3]))
                     hasBeenModified = True
    if hasBeenModified:
        return compactConstraintStructure (compactedConstraints)
    else:
        return compactedConstraints


##########################

def getStructureConstraints (graph, themeHeadNode):
    'Returns the structural constraints of a graph'
```

```
    nodes = getNodes (graph)

    # We first gather all the edge constraints
    constraints = []
    for n in nodes:
        daughters = getDaughters(graph, n)
        for d in daughters:
            constraint1 = (n, 'out', d[0], [d[1]])
            constraint2 = (d[1], 'in', d[0], [n])
            constraints.append (constraint1)
            constraints.append (constraint2)

    if themeHeadNode <> -1:
        constraints.append (('dot', 'out', 'root', [str(themeHeadNode)]))
        constraints.append ((str(themeHeadNode), 'in', 'root', ['dot']))

    # We compact the constraints
    compactedConstraints = compactConstraintStructure (constraints)

    constraintsLoops = {}
    for c in compactedConstraints:
        if constraintsLoops.has_key (c[0]):
            if constraintsLoops[c[0]].has_key (c[1]):
                constraintsLoops[c[0]][c[1]].append ((c[2], c[3]))
            else:
                constraintsLoops[c[0]][c[1]] = [(c[2], c[3])]
        else:
            constraintsLoops[c[0]] = {}
            constraintsLoops[c[0]][c[1]] = [(c[2], c[3])]

    # We finally build the oz constraints
    constraintsText = ""
    for node  in getNodes(graph):
        constraintsText = constraintsText + "            if Node.index == %
           s then \n" %graph[node]['linearPos']

        if constraintsLoops.has_key(node):
            # In Valency
            if constraintsLoops[node].has_key('in'):
                ListRestrictionsLeft = ""
                ListRestrictionsRight = ""
                for c in constraintsLoops[node]['in']:
                    if len (c[1]) ==1:
                        constraintsText = constraintsText + \
                                          "                  {FS.value.singl
                                             %i Node.gsem.model.mothersL.%s
                                             } \n" \
                                          % (graph[c[1][0]]['linearPos'],c
                                             [0])
                    else:
                        indexList = []
                        for i in c[1]:
                            indexList.append(graph[i]['linearPos'])
                        constraintsText = constraintsText + \
                                          "                  {FS.reified.
                                             equal {FS.value.make [%s]}
                                             Node.gsem.model.mothersL.%s}
                                             =: 1 \n" \
                                          % (listStrip(indexList), c[0])
```

```
                ListRestrictionsLeft = ListRestrictionsLeft + "{List.
                    subtract "
                ListRestrictionsRight = ListRestrictionsRight + " %s}"
                    % c[0]


            constraintsText = constraintsText + """
            for Val in %s{Arity Node.gsem.model.mothersL}%s do
                {FS.subset Node.gsem.model.mothersL.Val FS.value.empty}
            end \n""" %(ListRestrictionsLeft,ListRestrictionsRight)

        else:
            constraintsText = constraintsText + """
            for Val in  {Arity Node.gsem.model.mothersL} do
                {FS.subset Node.gsem.model.mothersL.Val FS.value.empty}
            end \n"""

        # Out Valency
        if constraintsLoops[node].has_key('out'):
            ListRestrictionsLeft = ""
            ListRestrictionsRight = ""
            for c in constraintsLoops[node]['out']:
                if len (c[1]) ==1:
                    constraintsText = constraintsText + \
                                      "                {FS.value.singl
                                         %i Node.gsem.model.daughtersL
                                         .%s} \n" \
                                      % (graph[c[1][0]]['linearPos'],c
                                         [0])
                else:
                    indexList = []
                    for i in c[1]:
                        indexList.append(graph[i]['linearPos'])
                    constraintsText = constraintsText + \
                                      "                {FS.reified.
                                         equal {FS.value.make [%s]}
                                         Node.gsem.model.daughtersL.%s}
                                          =: 1 \n" \
                                      % (listStrip(indexList), c[0])
                ListRestrictionsLeft = ListRestrictionsLeft + "{List.
                    subtract "
                ListRestrictionsRight = ListRestrictionsRight + " %s}"
                    % c[0]


            constraintsText = constraintsText + """
            for Val in %s{Arity Node.gsem.model.daughtersL}%s do
                {FS.subset Node.gsem.model.daughtersL.Val FS.value.
                    empty}
            end \n""" %(ListRestrictionsLeft,ListRestrictionsRight)

        else:
            constraintsText = constraintsText +"""
            for Val in {Arity Node.gsem.model.daughtersL} do
            {FS.subset Node.gsem.model.daughtersL.Val FS.value.empty}
            end \n"""

    else:
        constraintsText = constraintsText + "skip"
```

```
        constraintsText = constraintsText + "            end \n\n"

    return constraintsText

##########################

def getGramsConstraints (graph):
    'Return the grammenes constraints of the given graph'

    themeHeadNode = -1
    gramsText = ""
    for n in getNodes(graph):
        grams =  getGrams (graph, n)
        if grams <> []:
            gramText = """
         if Node.index == %i then
            {FS.reified.equal  {FS.value.make ["""%graph[n]['linearPos']
            for gram in grams:
                if gram[1] <> 'themeHead':
                    gramText = gramText + "{GUSTAlterGrams.getSemIndex %s %
                        s} "%(gram[1], gram[2])
                else:
                    themeHeadNode = n
                    if len (grams) == 1 :
                        gramsText = gramsText + """
         if Node.index == %i then
            {FS.subset Node.gsem.attrs.grams FS.value.empty}
         end\n\n"""%graph[n]['linearPos']
            gramText = gramText + "]} Node.gsem.attrs.grams} =: 1\n
                end\n\n"

            if '[]' not in gramText:
                gramsText = gramsText + gramText

        else:
            gramsText = gramsText + """
         if Node.index == %i then
            {FS.subset Node.gsem.attrs.grams FS.value.empty}
         end\n\n"""%graph[n]['linearPos']

    return (gramsText, themeHeadNode)

##########################

def getClosing ():
    'Returns the closing of the constraint file'

    return """
        end
    end
end
    """

##########################

def getNumberOfEmptyNodes (nodeLabel):
    '''Returns the number of empty nodes to add in the
    surroundings of the given node

    Precondition: the grammar must already be compiled'''
```

```
    GUSTEmptyNodes = GrammarName.replace('.txt', '_emptyNodes.txt')
    if os.path.exists (GUSTEmptyNodes):
        f = open (GUSTEmptyNodes, 'r')
    else:
        print "ERROR: the file %s does not exists, and is necessary for the
             compilation of the test suite. Please recompile your grammar"%
           GUSTEmptyNodes
        sys.exit()

    EmptyNodes = {}
    data = f.readlines()
    for line in data:
        lineSplit = line.strip('\n').split('\t')
        EmptyNodes[lineSplit[0]] = (int(lineSplit[1]), int(lineSplit[2]))

    if EmptyNodes.has_key(nodeLabel):
        return (EmptyNodes[nodeLabel])
    else:
        print "ERROR: the word %s is not present in the file describing
           number of empty nodes for generation"%nodeLabel
        sys.exit()

##########################

def getSortedNodes (graph):
    'Returns the sorted nodes list of the graph'

    sortedLinearPoss = []
    for o in graph.keys():
        if graph[o]['type'] == 'node.gsem' \
               and graph[o]['label'] <> '*' \
               and graph[o]['label'] <> '.':
            sortedLinearPoss.append (int(graph[o]['linearPos']))
    sortedLinearPoss.sort()

    sortedNodes = sortedLinearPoss[:]
    for o in graph.keys():
        if graph[o]['type'] == 'node.gsem' \
               and graph[o]['label'] <> '*' \
               and graph[o]['label'] <> '.':
            if int(graph[o]['linearPos']) in sortedLinearPoss:
                sortedNodes[sortedLinearPoss.index(int(graph[o]['linearPos'
                   ]))] = o

    return sortedNodes

##########################

def addEmptyNodes (graph):
    '''Add empty nodes in the graph according to the specifications of the
       grammar

    Precondition: the grammar must have already been compiled'''

    indexIncrement = 0
    sortedNodes = getSortedNodes (graph)

    for node in sortedNodes:
```

```
        (nb_before, nb_after) = getNumberOfEmptyNodes (graph[node]['label'
            ])
        graph[node]['linearPos'] = int(graph[node]['linearPos']) +
            indexIncrement + nb_before
        indexIncrement = indexIncrement + nb_before + nb_after

        for emptyNodeIndex in range(nb_before):
            nodeName = str(node)+'b'+str(emptyNodeIndex+1)
            edgeName = 'del-'+ nodeName
            graph[nodeName] = {'label':'*', 'type':'node.gsem', 'p':'+', \
                               'linearPos':int(graph[node]['linearPos'])+
                                   emptyNodeIndex - nb_before}
            graph[edgeName] = {'label':'del', 'source':'dot', 'type':'edge.
                gsem', 'target':nodeName, 'p':'+'}

        for emptyNodeIndex in range(nb_after):
            nodeName = str(node)+'a'+str(emptyNodeIndex+1)
            edgeName = 'del-'+ nodeName
            graph[nodeName] = {'label':'*', 'type':'node.gsem', 'p':'+', \
                               'linearPos':int(graph[node]['linearPos'])+
                                   emptyNodeIndex+ 1}
            graph[edgeName] = {'label':'del', 'source':'dot', 'type':'edge.
                gsem', 'target':nodeName, 'p':'+'}

    biggestIndex = 0
    for node in getNodes(graph):
        if graph[node]['linearPos'] > biggestIndex:
            biggestIndex = graph[node]['linearPos']
    graph['dot'] = {'label':'.', 'type':'node.gsem', 'p':'+', 'linearPos':
        biggestIndex+1}
    return graph

##########################

def extractInputSentence (graph):
    'Returns the input sentence of the graph'

    nodes = getNodes (graph)
    sortedNodes = getSortedNodes (graph)

    for n in nodes:
        if not n.isdigit():
            if len(n.split('b')) == 2:

                index = sortedNodes.index (n.split('b')[0])
                sortedNodes.insert(index,n)
            elif len(n.split('a')) == 2:
                index = sortedNodes.index (n.split('a')[0]) + 1
                sortedNodes.insert(index,n)
            else:
                sortedNodes.append(n)

    inputSentence = []
    for n in sortedNodes:
        inputSentence.append ("%s"%graph[n]['label'])

    return str(inputSentence).replace(",","")

##########################
```

```
def generateInputConstraint(graph):
    '''Generate the whole constraint file
    (i.e. structural constraints + grammenes constraints)'''

    labels = getLabels(graph)
    nodes = getNodes(graph)
    graph2 = addEmptyNodes (graph)

    preamble = getPreamble ()

    (gramsConstraints, themeHeadNode) = getGramsConstraints (graph2)

    structureConstraints = getStructureConstraints (graph2, themeHeadNode)


    closing = getClosing()

    text = preamble + structureConstraints + gramsConstraints + closing

    return text

##########################

def __getTabs (string):
    if len(string) < 16:
        return " ...\t\t\t"
    elif len(string) < 24:
        return " ...\t\t"
    else:
        return " ...\t"

##########################

def convertDiaTestFiles():
    'Convert the Dia Test files in the directory to a unique text file'

    print
    answer = raw_input( "Path of the directory to be parsed [default: %s]:
       "%Constants.DiaTestFiles)
    while not os.path.exists (answer) and answer != "":
        print "Sorry, wrong path ! \n"
        answer = raw_input( "Path of the directory to be parsed [default: %
            s]: "%Constants.DiaTestFiles)
    if answer == "":
        answer = Constants.DiaTestFiles

    total = 0
    directory = answer
    directoryList = os.listdir(directory)
    for f in directoryList:
        if ".dia" in f:
            total = total + 1

    if total == 0:
        print "Sorry, no Dia file in this directory!"
        convertDiaTestFiles()

    print "Begin parsing of Dia files directory.... (total of %i files)"%(
       total)
```

```
    # Parse the directory
    GUSTTests = ""
    increment = 0
    incrPercentage = 0
    for i in directoryList:
        if ".dia" in i and 'autosave' not in i and '~' not in i:
            increment = increment + 1
            incrPercentage = incrPercentage + 1
            print i.replace('.dia','') + __getTabs (i),
            gustData = Dia2GUST_gsem.process (directory+os.sep+i)
            GUSTTests = GUSTTests + i.replace(".dia", "") + "\t{\n" +
                gustData + "\n\t}\n\n"
            if float(incrPercentage) >= float(total) / 10:
                print "\t\t\t\t\t%u %%"%(int(float(increment *100)/float(
                    total)))
                incrPercentage = 0

    print
    answer = raw_input( "Filename of the PUG test suite [default: %s]: "%
        Constants.GUSTTestsFile)
    if answer == "":
        answer = Constants.GUSTTestsFile

    # Write the test suite
    f = file (answer, 'w')
    f.write (GUSTTests)
    f.close()
    print "Writing of PUG test suite successful"


##########################


def IntegrateTests ():
    'Integrate the test suite to the XDK constraints'

    print
    answer = raw_input( "Path and filename of the PUG test suite [default:
        %s]: "%Constants.GUSTTestsFile)
    while not os.path.exists (answer) and answer != "":
        print "Sorry, wrong path ! \n"
        answer = raw_input( "Path and filename of the PUG test suite [
            default: %s]: "%Constants.GUSTTestsFile)
    if answer == "":
        answer = Constants.GUSTTestsFile

    f = file (answer, 'r')
    Tests = f.read ()
    f.close()
    testSuite = PUGParser.parseTests (Tests)

    print
    answer = raw_input( "Path and filename of the PUG grammar [default: %s
        ]: "%Constants.GUSTGrammarFile)
    while not os.path.exists (answer) and answer != "":
        print "Sorry, wrong path ! \n"
        answer = raw_input( "Path and filename of the PUG grammar [default:
            %s]: "%Constants.GUSTGrammarFile)
    if answer == "":
        answer = Constants.GUSTGrammarFile
```

```
global GrammarName
GrammarName = answer

RelativeXDK = Constants.RelativeXDK


inputSentences = []
TestsData = {}
GUSTGeneration2 = ""
GUSTGeneration4 = ""

print
print "Begin integration of test constraints into the XDK..."

# Iterate on each test of the test suite
for testNb in testSuite.keys():

    # Generate and write the constraint file
    print testNb + __getTabs (testNb),
    XDKConstraints = generateInputConstraint(testSuite[testNb])
    i3 = "InputGraph" + testNb.replace('test', '') + ".oz"
    f = file (RelativeXDK + Constants.InputGraphsPath + i3, 'w')
    f.write (XDKConstraints)
    f.close()

    # Extract the input sentence
    inputSentence = extractInputSentence (testSuite[testNb])
    inputSentences.append (listStrip(inputSentence)+'\n')

    # Add the test to the GUSTGeneration file
    GUSTGeneration2 = GUSTGeneration2 +      "%s at \'InputGraphs/%sf\'\
        n" \
                    %(i3.rstrip('.oz'), i3)

    GUSTGeneration4 = GUSTGeneration4 + """
    if Sentence == %s then
    {%s.fixInputGraph Nodes}
    FoundSentence = true
    end\n\n"""%(inputSentence, i3.rstrip('.oz'))

    # Compile the constraint
    os.system ("ozc -q -c %s -o %s"%(RelativeXDK + Constants.
        InputGraphsPath + i3, RelativeXDK + Constants.InputGraphsPath +
        i3+ 'f'))
    print "...OK"

# Write and compile the GUSTGeneration file
d = file (RelativeXDK + Constants.GUSTGenerationFile, 'w')
d.write (GUSTGeneration1 + GUSTGeneration2 + GUSTGeneration3 +
    GUSTGeneration4 + GUSTGeneration5)
d.close()
os.system ("ozc -q -c %s -o %s"%(RelativeXDK + Constants.
    GUSTGenerationFile, RelativeXDK + Constants.GUSTGenerationFile+ 'f')
    )
print "Constraint functors integration and compilation successful"

if '.txt' in GrammarName:
    XDGExamplesName = GrammarName.lstrip('.').replace("%s"%os.sep, '')
else:
```

```
        XDGExamplesName = GrammarName.lstrip('.').replace("%s"%os.sep, '')
            + ".txt"

    # Write the input sentences file
    print
    print "Begin Input sentences writing..."
    d = file (RelativeXDK + "Grammars%s"%os.sep + XDGExamplesName, 'w')
    for iS in inputSentences:
        d.write (iS)
    d.close()
    print "Input sentences writing sucessful"
    print
    print "Test suite successfully integrated"
```

C.4 GUSTSemEdgeConstraints.oz

```
%% Copyright 2001-2006
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%Denys Duchier <duchier@ps.uni-sb.de> (LORIA, Nancy)
%%
%% Constraint ensuring that every semantic edge is saturated by
%% an interface rule
%%
%% Principle written by Pierre Lison <plison@agora.eu.org>
%% on 26/02/06
functor
import
%    System
   FS
   Helpers(checkModel) at 'Helpers.ozf'
export
   Constraint
define
   fun {Constraint Nodes G Principle}

      Proc =
      proc {$ Node1 Node2 LA}

         % All the semantic edges must be satured by a Isemsynt rule
         if {And LA \= root LA \= del} then
            if {And {FS.reflect.cardOf.lowerBound Node1.isemsynt.attrs.
                semOut.LA} < 1
                 {FS.reflect.cardOf.lowerBound Node2.isemsynt.attrs.semIn.LA
                     } < 1} then

                {FS.card Node1.gsem.model.daughtersL.LA} =: 0
                {FS.card Node2.gsem.model.mothersL.LA} =: 0

         % Linguistic hypothesis: we never have more than 3 edges with the
             same label
         % attached to a node
            else
                {FS.card Node1.gsem.model.daughtersL.LA} =<: 3
                {FS.card Node2.gsem.model.mothersL.LA} =<: 3
            end
         end
```

```
      end
   in
      Proc
   end
end
```

C.5 GUSTSagittalConstraints.oz

```
%% Copyright 2001-2006
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%Denys Duchier <duchier@ps.uni-sb.de> (LORIA, Nancy)
%%
%% Constraint in charge of processing the "sagit" rules
%% and ensuring that every syntactic edge is saturated on Gsynt
%%
%% Principle written by Pierre Lison <plison@agora.eu.org>
%% on 26/02/06

functor
import
 %  System
   FS
   Helpers(checkModel) at 'Helpers.ozf'
export
   Constraint
define
   proc {Constraint Nodes G Principle}

      % Iterate on Nodes
      for Node in {List.reverse Nodes} do
         RulesList = Node.gsynt.entry.sagit
      in

         % Just in case
         if {Not {Value.isDet RulesList}} then
            {Wait RulesList}
         end

         if {Value.isDet RulesList} then

            % Iterate on Rules
            for Rule in RulesList do

               % Verify the gram conditions of a rule
               fun {VerifyGramsSagit Grams}
                  case Grams
                  of nil then true
                  [] Gram|T then
                     if {List.member Gram {FS.reflect.lowerBoundList Node.
                        gsynt.attrs.grams}} then
                        {VerifyGramsSagit T}
                     else Result in
                        for Node2 in Nodes do
                           for Val in {Arity Node2.isemsynt.attrs.
                              syntInGrams} do
                              if {List.member Gram
```

```
                        {FS.reflect.lowerBoundList Node2.isemsynt
                            .attrs.syntInGrams.Val}}
                       andthen {Value.isFree Result} then
                       Result = true
                    end
                 end
                 for Val in {Arity Node2.isemsynt.attrs.
                    syntOutGrams} do
                    if {List.member Gram
                        {FS.reflect.lowerBoundList Node2.isemsynt
                            .attrs.syntOutGrams.Val}}
                       andthen {Value.isFree Result} then
                       Result = true
                    end
                 end
              end
              if {Value.isDet Result} then Result else false end
           end
        end
     end

     % Verify the POS condition of a rule
     fun {VerifyPOS POS} Result in
        if {FS.reflect.lowerBoundList POS} \= nil then
           for Pos in {FS.reflect.lowerBoundList POS} do
              if {List.member Pos {FS.reflect.upperBoundList Node
                  .gsynt.entry.pos}}
                 andthen {Value.isFree Result} then
                 Result = true
              end
           end
           if {Value.isDet Result} then Result else  false end
        else
           true
        end
     end

     AllOK

     % Appply the verification of the gram conditions
     if {Value.isFree AllOK} then
        if {Not {VerifyGramsSagit {FS.reflect.lowerBoundList Rule
            .sagitConds.gramsSagit}}} then
           AllOK = false end
     end

     % Apply the verification of the POS conditions
     if {Value.isFree AllOK} then
        if {Not {VerifyPOS Rule.sagitConds.posSagit}} then
           AllOK = false end
     end

  in
     % if the rule can be applied
     if {Value.isFree AllOK} then

        % Saturation of the edge
        % NB: indeed, if Rule.sagitModifs.(in or out).Val == 1,
        % the constraint below states that 1 must be in Node.
            gsynt.attrs.(in or out).Val,
```

```
                    % which means the saturation of the edge

                    for Val in {Arity Node.gsynt.attrs.out} do
                       {FS.subset Rule.sagitModifs.out.Val Node.gsynt.attrs.
                          out.Val}
                    end
                    for Val in {Arity Node.gsynt.attrs.'in'} do
                       {FS.subset Rule.sagitModifs.'in'.Val Node.gsynt.attrs.
                          'in'.Val}
                    end
                 end

              end
           end

        end


        % Iterate on all edges to verify that they are well saturated
        for Node in Nodes do
           for Val in {Arity Node.gsynt.attrs.'in'} do
              if {Not {Value.isKinded Node.gsynt.entry.'in'.Val}} then
                 {FS.subset Node.gsynt.attrs.'in'.Val
                  {FS.value.make {FS.reflect.lowerBound Node.gsynt.attrs.'in'
                     .Val}}}
              else
                 {FS.subset Node.gsynt.attrs.'in'.Val Node.gsynt.entry.'in'.
                    Val}
              end
              if {Not {Value.isKinded Node.gsynt.entry.out.Val}} then
                 {FS.subset Node.gsynt.attrs.out.Val
                  {FS.value.make {FS.reflect.lowerBound Node.gsynt.attrs.out.
                     Val}}}
              else
                 {FS.subset Node.gsynt.attrs.out.Val Node.gsynt.entry.out.Val
                    }
              end
           end
        end
     end

end
```

C.6 GUSTLinkingConstraints.oz

```
%% Copyright 2001-2006
%% by Ralph Debusmann <rade@ps.uni-sb.de> (Saarland University) and
%%Denys Duchier <duchier@ps.uni-sb.de> (LORIA, Nancy)
%%
%% Constraint in charge of processing the "link" rules:
%% 1) Verify the preconditions
%% 2) Saturate the semantic edges (and if several saturations are possible,
     distribute the rules)
%% 3) Saturate the syntactic edges
%% 4) Constrain the geometry of the structures with the linking constraints
%% 5) "Generate" new nodes (with empty semantic content) for multiword
    expressions
```

```
%%
%% Principle written by Pierre Lison <plison@agora.eu.org>
%% on 26/02/06
functor
import
  % System
   FD
   FS

   Helpers(checkModel) at 'Helpers.ozf'
export
   Constraint
define
   proc {Constraint Nodes G Principle}

      % Verify if two rules are mutually compatible
      % Rec1 and Rec2 are records describing the semantic parts of the two
          rules
      fun {IsCompatibleSemSaturation Rec1 Rec2} Result in

         if Rec1.semGrams \= FS.value.empty andthen Rec1.semGrams \= FS.
             value.empty then
            for Gram in {FS.reflect.lowerBoundList Rec1.semGrams} do
               if {List.member Gram {FS.reflect.lowerBoundList Rec2.
                   semGrams}} then
                  Result = false
               end
            end
         end

         for Val in {Arity Rec1.semIn} do

            if Rec1.semIn.Val \= {FS.value.make [0]}
               andthen Rec1.semIn.Val == Rec2.semIn.Val
               andthen {Value.isFree Result} then
               Result = false
            end
            if Rec1.semOut.Val \= {FS.value.make [0]}
               andthen Rec1.semOut.Val == Rec2.semOut.Val
               andthen {Value.isFree Result} then
               Result = false
            end

            if Rec1.semOutGrams.Val \= FS.value.empty
               andthen Rec1.semOutGrams.Val == Rec2.semOutGrams.Val
               andthen {Value.isFree Result} then
               Result = false
            end
         end

         for Feat in [semInIn semInOut semOutIn semOutOut] do
            if Rec1.Feat \= nil andthen Rec2.Feat \= nil then  FVal11
                FVal21 FVal12 FVal22 in
               for SemXX in Rec1.Feat do
                  Valencies1 = {List.nth SemXX 1}
                  Valencies2 = {List.nth SemXX 2}
               in
                  for Val in {Arity Valencies1} do
                     if {FS.int.min Valencies1.Val} == 1 then FVal11 = Val
                           end
```

```
            end
            for Val in {Arity Valencies2} do
               if {FS.int.min Valencies2.Val} == 1 then FVal21 = Val
                  end
            end
         end
         for SemXX in Rec2.Feat do
            Valencies1 = {List.nth SemXX 1}
            Valencies2 = {List.nth SemXX 2}
         in
            for Val in {Arity Valencies1} do
               if {FS.int.min Valencies1.Val} == 1 then FVal12 = Val
                        end
            end
            for Val in {Arity Valencies2} do
               if {FS.int.min Valencies2.Val} == 1 then FVal22 = Val
                  end
            end
         end

         if {Value.isFree Result} andthen FVal11 == FVal12 andthen
            FVal21 == FVal22 then
            Result = false
         end
      end
   end

   if {Value.isFree Result} then
      Result = true
   end
   Result
end

% Verify is a the semantic part of a rule is empty (i.e. no
    saturation)
fun {IsEmptySemRecord Rec} Result in

   if Rec.semGrams \= FS.value.empty then
      Result = false
   end

   for Feat in [semOutGrams semInGrams] do
      for Val in {Arity Rec.Feat} do
         if Rec.Feat.Val \= FS.value.empty then
            if {Value.isFree Result} then Result = false end
         end
      end
   end
   for Feat in [semIn semOut] do
      for Val in {Arity Rec.Feat} do
         if Rec.Feat.Val \= {FS.value.make [0]} then
            if {Value.isFree Result} then Result = false end
         end
      end
   end
   if {Value.isDet Result} then Result else true end
end

% Returns all the compatible rules
% and distribute those who are incompatible
```

```
fun {GetCompatibleRulesList Links}
   HasConflict
   ConflictList = {Cell.new nil}
   IndexList = {List.number 1 {List.length Links} 1}
   ExcludedList1 = {Cell.new nil}
   ExcludedList2 = {Cell.new nil}

in
   for Index1 in IndexList do
      for Index2 in {List.drop IndexList Index1} do

         % if we have a conflict ....
         if {Not {IsCompatibleSemSaturation {List.nth Links Index1}.
            sem {List.nth Links Index2}.sem}}
            andthen {Not {IsEmptySemRecord {List.nth Links Index1}.
               sem}}
            andthen {Not {IsEmptySemRecord {List.nth Links Index2}.
               sem}}
            andthen {Value.isFree HasConflict} then

            {Cell.assign ConflictList {List.append {Cell.access
               ConflictList} [Index1]}}
            {Cell.assign ConflictList {List.append {Cell.access
               ConflictList} [Index2]}}

            {Cell.assign ExcludedList1 {List.append {Cell.access
               ExcludedList1} [Index2]}}
            {Cell.assign ExcludedList1 {List.append {Cell.access
               ExcludedList1} [Index1]}}
            for Index3 in {List.drop {List.drop IndexList Index1}
               Index2} do
               if {Not {IsCompatibleSemSaturation {List.nth Links
                  Index1}.sem {List.nth Links Index3}.sem}}
                  andthen {Not {IsEmptySemRecord {List.nth Links
                     Index3}.sem}} then
                  {Cell.assign ExcludedList1 {List.append {Cell.
                     access ExcludedList1} [Index3]}}
               end
            end

            {Cell.assign ExcludedList2 {List.append {Cell.access
               ExcludedList2} [Index1]}}
            {Cell.assign ExcludedList2 {List.append {Cell.access
               ExcludedList2} [Index2]}}
            for Index3 in {List.drop {List.drop IndexList Index1}
               Index2} do
               if {Not {IsCompatibleSemSaturation {List.nth Links
                  Index2}.sem {List.nth Links Index3}.sem}}
                  andthen {Not {IsEmptySemRecord {List.nth Links
                     Index3}.sem}} then
                  {Cell.assign ExcludedList2 {List.append {Cell.
                     access ExcludedList2} [Index3]}}
               end
            end
            HasConflict = true
         end
      end
   end

   % Apply the distribution in case of conflict
```

```
      if {Cell.access ConflictList} \= nil then
         Split = {FS.var.upperBound {Cell.access ConflictList}}
         OthersList1 = {Cell.new Links}
         OthersList2 = {Cell.new Links}
      in
         {FS.card Split} =: 1

         for El in {Cell.access ExcludedList1} do
            {Cell.assign OthersList1 {List.subtract {Cell.access
               OthersList1} {List.nth Links El}}}
         end
         for El in {Cell.access ExcludedList2} do
            {Cell.assign OthersList2 {List.subtract {Cell.access
               OthersList2} {List.nth Links El}}}
         end

         {FS.distribute naive [Split]}

         if {FS.int.min Split} == 1 then
            {GetCompatibleRulesList {List.append [{List.nth Links {FS.
               int.min Split}}] {Cell.access OthersList1}}}
         else
            {GetCompatibleRulesList {List.append [{List.nth Links {FS.
               int.min Split}}] {Cell.access OthersList2}}}
         end

      else
         Links
      end
   end

in

   % Iterate on nodes
   for Node in Nodes do

      if {Value.isDet Node.isemsynt.entry.link} then AdequateRulesList
          RulesList in

         AdequateRulesList = {Cell.new nil}

         % Iterate on rules
         for Rule in Node.isemsynt.entry.link do

            % Verify the sem grams specifications
            fun {VerifySemGrams Grams}
               case Grams
               of nil then true
               [] Gram|T then
                  if {List.member Gram {FS.reflect.lowerBoundList Node.
                     gsem.attrs.grams}} then
                     {VerifySemGrams T}
                  else
                     false
                  end
               end
            end

            % Verify the sem 'in' edges specifications
            fun {VerifySemIn InVals} OKSemIn in
```

```
    for Val in {Arity InVals} do
       if {FS.int.min InVals.Val} == 1 then
          if {FS.reflect.cardOf.lowerBound Node.gsem.model.
             mothersL.Val} < 1 then
             OKSemIn = false
          end
       end
    end
    if {Value.isFree OKSemIn} then
       OKSemIn = true
    end
    OKSemIn
end

% Verify the sem 'out' edges specifications
fun {VerifySemOut OutVals} OKSemOut in
    for Val in {Arity OutVals} do
       if {FS.int.min OutVals.Val} == 1 then
          if {FS.reflect.cardOf.lowerBound Node.gsem.model.
             daughtersL.Val} < 1 then
             OKSemOut = false
          end
       end
    end
    if {Value.isFree OKSemOut} then
       OKSemOut = true
    end
    OKSemOut
end

% Verify the sem 'in' labels (i.e. constraint on the id of
    the above node)
fun {VerifySemInLabels InLabels} OKSemInLabels in
    for Val in {Arity InLabels} do
       if {FS.reflect.cardOf.lowerBound InLabels.Val} == 1
          then UpperNodeIndex in
          UpperNodeIndex = {FS.int.min Node.gsem.model.
             mothersL.Val}
          for Node2 in Nodes do
             if Node2.index == UpperNodeIndex then
                if {FS.int.min InLabels.Val} \= Node2.gsem.
                   entry.id then
                   OKSemInLabels = false
                end
             end
          end
       end
    end
    if {Value.isFree OKSemInLabels} then
       OKSemInLabels = true
    end
    OKSemInLabels
end

% Verify the sem 'out' labels (i.e. constraint on the id of
    the below node)
fun {VerifySemOutLabels OutLabels} OKSemOutLabels
in
    for Val in {Arity OutLabels} do
```

```
            if {FS.reflect.cardOf.lowerBound OutLabels.Val} == 1
               then DownNodeIndex in
               DownNodeIndex = {FS.int.min Node.gsem.model.
                  daughtersL.Val}
               for Node2 in Nodes do
                  if Node2.index == DownNodeIndex then
                     if {FS.int.min OutLabels.Val} \= Node2.gsem.
                        entry.id
                        andthen {Value.isFree OKSemOutLabels} then
                        OKSemOutLabels = false
                     end
                  end
               end
            end
         end
         if {Value.isFree OKSemOutLabels} then
            OKSemOutLabels = true
         end
         OKSemOutLabels
      end

      % Verify the synt grams preconditions
      fun {VerifySyntGramsPreconditions Grams}
         case Grams
         of nil then true
         [] Gram|T then
            if {List.member Gram {FS.reflect.lowerBoundList Node.
               gsynt.attrs.grams}} then
               {VerifySyntGramsPreconditions T}
            else
               false
            end
         end
      end


      % Verify the synt POS preconditions
      fun {VerifySyntPOSPreconditions SyntPOSPreconditions}
          Result in
         if {FS.reflect.lowerBoundList SyntPOSPreconditions} \=
             nil then
            for Pos in {FS.reflect.lowerBoundList
                SyntPOSPreconditions} do
               if {List.member Pos {FS.reflect.upperBoundList Node
                   .gsynt.entry.pos}}
                  andthen {Value.isFree Result} then
                  Result = true
               end
            end
            if {Value.isDet Result} then Result else  false end
         else
            true
         end
      end

      AllOK

      % Apply the verification of sem grams
      OKSemGrams = {VerifySemGrams {FS.reflect.lowerBoundList Rule
          .preconditions.semGrams}}
```

```
if {Not OKSemGrams} then AllOK = false end

OKSemGrams2 = {VerifySemGrams {FS.reflect.lowerBoundList
    Rule.sem.semGrams}}
if {Not OKSemGrams2} then AllOK = false end

% Apply the verification of sem 'in' edges
if {Value.isFree AllOK} then
   if {Not {VerifySemIn Rule.sem.semIn}} then AllOK = false
      end
end

% Apply the verification of sem 'out' edges
if {Value.isFree AllOK} then
   if {Not {VerifySemOut Rule.sem.semOut}} then AllOK =
      false end
end

% Apply the verification of sem 'in' edges (unsaturated)
if {Value.isFree AllOK} then
   if {Not {VerifySemIn Rule.preconditions.semIn}} then
      AllOK = false end
end

% Apply the verification of sem 'out' edges (unsaturated)
if {Value.isFree AllOK} then
   if {Not {VerifySemOut Rule.preconditions.semOut}} then
      AllOK = false end
end

% Apply the verification of sem 'in' labels
if {Value.isFree AllOK} then
   if {Not {VerifySemInLabels Rule.preconditions.semInLabels
      }} then AllOK = false end
end

% Apply the verification of sem 'out' labels
if {Value.isFree AllOK} then
   if {Not {VerifySemOutLabels Rule.preconditions.
      semOutLabels}} then
      AllOK = false
   end
end

% Apply the verification of synt grams preconditions
if {Value.isFree AllOK} then
   if {Not {VerifySyntGramsPreconditions
            {FS.reflect.lowerBoundList Rule.preconditions.
               syntGrams}}} then AllOK = false end
end

% Apply the verification of synt POS preconditions
if {Value.isFree AllOK} then
   if {Not {VerifySyntPOSPreconditions Rule.preconditions.
      syntPOS}} then AllOK = false end
end

if {Value.isFree AllOK} then
   AllOK = true
end
```

```
        in
           if AllOK then
              {Cell.assign AdequateRulesList Rule|{Cell.access
                 AdequateRulesList}}
           else
              % in case the rule is lexicalised and the conditions are
                 not met
              % declare the CSP to be unsolved
              if Rule.lexicalised == 2 then
                 {FS.include Node.gsynt.model.labels FS.value.empty}
              end
           end
        end

        % Get the compatible rules list
        RulesList = {GetCompatibleRulesList {Cell.access
           AdequateRulesList}}

        % Iterate on rules
        for Rule in RulesList do

           % saturate sem grams
           {FS.subset Rule.sem.semGrams Node.isemsynt.attrs.semGrams}

           % satured synt grams
           {FS.subset Rule.synt.syntGrams Node.gsynt.attrs.grams}

           % constrain POS
           if {FS.reflect.cardOf.lowerBound Rule.preconditions.syntPOS}
               == 1 then
              {FS.reified.equal Rule.preconditions.syntPOS Node.gsynt.
                 attrs.pos} =: 1
           end

           % saturate sem edges
           for Val in {Arity Node.gsem.model.daughtersL} do

              if {FS.int.min Rule.sem.semOut.Val} >= 1 then
                 {FS.include
                  {FS.int.min Rule.sem.semOut.Val}
                  Node.isemsynt.attrs.semOut.Val}
              end

              if {FS.int.min Rule.sem.semIn.Val} >= 1 then
                 {FS.include
                  {FS.int.min Rule.sem.semIn.Val}
                  Node.isemsynt.attrs.semIn.Val}
              end
           end

           for Val in {Arity Node.gsynt.model.daughtersL} do

              % Saturation of synt 'out' edges
              if {FS.int.min Rule.synt.syntOut.Val} >= 1 then
                 {FS.include
                  {FS.int.min Rule.synt.syntOut.Val}
                  Node.isemsynt.attrs.syntOut.Val}
                 {FS.card Node.gsynt.model.daughtersL.Val} >=: 1
              end
```

```
            % Saturation of synt 'in' edges
            if {FS.int.min Rule.synt.syntIn.Val} >= 1 then
               {FS.include
                {FS.int.min Rule.synt.syntIn.Val}
                Node.isemsynt.attrs.syntIn.Val}
               {FS.card Node.gsynt.model.mothersL.Val} >=: 1
            end

            % Constraint on node ids
            if {FS.reflect.cardOf.lowerBound Rule.synt.syntOutLabels.
               Val} == 1 then
               {FS.reified.equal Rule.synt.syntOutLabels.Val Node.
                  isemsynt.attrs.syntOutLabels.Val} =: 1
            end
            if {FS.reflect.cardOf.lowerBound Rule.synt.syntInLabels.
               Val} == 1 then
               {FS.reified.equal Rule.synt.syntInLabels.Val Node.
                  isemsynt.attrs.syntInLabels.Val} =: 1
            end

            % Saturation of 'out' synt grams
            if {FS.reflect.card Rule.synt.syntOutGrams.Val} > 0 then
               {FS.subset Rule.synt.syntOutGrams.Val Node.isemsynt.
                  attrs.syntOutGrams.Val}
            end

            % Saturation of 'in' synt grams
            if {FS.reflect.card Rule.synt.syntInGrams.Val} > 0 then
               {FS.subset Rule.synt.syntInGrams.Val Node.isemsynt.
                  attrs.syntInGrams.Val}
            end

            % constraint on 'out' POS
            if {FS.reflect.card Rule.synt.syntOutPOS.Val} > 0 then
               {FS.subset Rule.synt.syntOutPOS.Val Node.isemsynt.
                  attrs.syntOutPOS.Val}
            end

            % constraint on 'in' POS
            if {FS.reflect.card Rule.synt.syntInPOS.Val} > 0 then
               {FS.subset Rule.synt.syntInPOS.Val Node.isemsynt.attrs
                  .syntInPOS.Val}
            end
         end

         % Saturation of 'double' sem edges
         % (4 types: semOutOut, semOutIn, semInOut and semInIn)
         for Feat in [     o(featName:semOutOut var1:daughtersL var2:
            semOut)
                           o(featName:semOutIn var1:daughtersL var2:
                              semIn)
                           o(featName:semInOut var1:mothersL var2:
                              semOut)
                           o(featName:semInIn var1:daughtersL var2:
                              semIn)] do
            FeatName = Feat.featName
            Var1 = Feat.var1
            Var2 = Feat.var2
         in
            if Rule.sem.FeatName \= nil then
```

```
            for El in Rule.sem.FeatName do FVal1 FVal2
               Valencies1 = {List.nth El 1}
               Valencies2 = {List.nth El 2}
            in
               for Val in {Arity Valencies1} do
                  if {FS.int.min Valencies1.Val} == 1 then
                     FVal1 = Val
                  end
               end
               for Val in {Arity Valencies2} do
                  if {FS.int.min Valencies2.Val} == 1 then
                     FVal2 = Val
                  end
               end
               for Node2 in Nodes do
                  if Node2.index == {FS.int.min Node.gsem.model.
                     Var1.FVal1} then
                     {FS.value.singl 1 Node2.isemsynt.attrs.Var2.
                        FVal2}
                  end
               end
            end
         end
      end


      % Saturation of 'double' synt edges
      % (4 types: syntOutOut, syntOutIn, syntInOut and syntInIn)
      for Feat in [     o(featName:syntOutOut var1:daughtersL var2:
          syntOut var3:syntOutLabels)
                        o(featName:syntOutIn var1:mothersL var2:
                           syntIn var3:syntOutLabels)
                        o(featName:syntInOut var1:daughtersL var2:
                           syntOut var3:syntInLabels)
                        o(featName:syntInIn var1:mothersL var2:
                           syntIn var3: syntInLabels)] do
         FeatName = Feat.featName
         Var1 = Feat.var1
         Var2 = Feat.var2
         Var3 = Feat.var3
      in
         if Rule.synt.FeatName \= nil then
            for El in Rule.synt.FeatName do Result FVal1 FVal2
               Valencies1 = {List.nth El 1}
               Valencies2 = {List.nth El 2}
            in
               for Val in {Arity Valencies1} do
                  if {FS.int.min Valencies1.Val} == 1 then
                     FVal1 = Val
                  end
               end
               for Val in {Arity Valencies2} do
                  if {FS.int.min Valencies2.Val} == 1 then
                     FVal2 = Val
                  end
               end

               if {FS.reflect.card Rule.synt.Var3.FVal1} == 1 then
                  for Node2 in Nodes do ID in
                     ID = Node2.gsynt.entry.id
```

```
                    if {Value.isKinded ID} then
                       if {List.member {FS.int.min Rule.synt.Var3
                          .FVal1}
                          {FD.reflect.domList ID}} andthen {
                             Value.isFree Result} then

                          {FD.reified.int [{FS.int.min Rule.synt.
                             Var3.FVal1}] ID} =<:
                          {FS.reified.include 1 Node2.isemsynt.
                             attrs.Var2.FVal2}

                          {FD.reified.int [{FS.int.min Rule.synt.
                             Var3.FVal1}] ID} =<:
                          {FS.card Node2.gsynt.model.Var1.FVal2}
                       end
                    else
                       if ID == {FS.int.min Rule.synt.Var3.FVal1}
                          andthen {Not {FS.var.is Result}} then
                          Result = Node2.isemsynt.attrs.Var2.
                             FVal2
                       end
                    end
                 end
                 if {FS.var.is Result} then {FS.value.singl 1
                    Result} end
              else
                 for Node2 in Nodes do
                    {FS.include 1 Node2.isemsynt.attrs.Var2.FVal2
                       }
                 end
              end
           end
        end
     end


     % Saturation of 'triple' syntactic edges
     if Rule.synt.syntOutOutOut \= nil then
        for SyntOutOutOut in Rule.synt.syntOutOutOut do  Result
           FVal1 FVal2 FVal3 SOOL
           Valencies1 = {List.nth SyntOutOutOut 1}
           Valencies2 = {List.nth SyntOutOutOut 2}
           Valencies3 = {List.nth SyntOutOutOut 3}
        in
           for Val in {Arity Valencies1} do
              if {FS.int.min Valencies1.Val} == 1 then
                 FVal1 = Val
              end
           end
           for Val in {Arity Valencies2} do
              if {FS.int.min Valencies2.Val} == 1 then
                 FVal2 = Val
              end
           end
           for Val in {Arity Valencies3} do
              if {FS.int.min Valencies3.Val} == 1 then
                 FVal3 = Val
              end
           end
```

```
      for SyntOutOutLabels in Rule.synt.syntOutOutLabels do
         for Val in {Arity {List.nth SyntOutOutLabels 1}} do
            if {FS.int.min {List.nth SyntOutOutLabels 1}.Val
               } == 1 andthen
               Val == FVal1 then
               SOOL = {List.nth SyntOutOutLabels 2}
            end
         end
      end

      for Node2 in Nodes do ID in
         ID = Node2.gsynt.entry.id
         if {Value.isKinded ID} then
            if {List.member {FS.int.min Rule.synt.
               syntOutLabels.FVal1}
                {FD.reflect.domList ID}} then
               for Node3 in Nodes do ID2 in
                  ID2 = Node3.gsynt.entry.id
                  if {Value.isKinded ID2} then
                     if {List.member {FS.int.min SOOL.FVal2}
                         {FD.reflect.domList ID2}}  then
                        {FD.reified.int
                         [{FS.int.min SOOL.FVal2}] ID2} =<:
                        {FS.reified.include 1 Node3.isemsynt
                           .attrs.syntOut.FVal3}
                        {FD.reified.int
                         [{FS.int.min SOOL.FVal2}] ID2} =<:
                        {FS.card Node3.gsynt.model.
                           daughtersL.FVal3}
                     end
                  else
                     if Node3.gsynt.entry.id ==
                        {FS.int.min SOOL.FVal2}
                        andthen {Not {FS.var.is Result}}
                           then
                        Result = Node3.isemsynt.attrs.
                           syntOut.FVal3
                     end
                  end
               end
            end
         else
            if ID == {FS.int.min Rule.synt.syntOutLabels.
               FVal1} then
               for Node3 in Nodes do ID2 in
                  ID2 = Node3.gsynt.entry.id
                  if {Value.isKinded ID2} then
                     if {List.member {FS.int.min Rule.synt.
                        syntOutOutLabels.FVal1.FVal2}
                         {FD.reflect.domList ID2}}  then
                        {FD.reified.int
                         [{FS.int.min Rule.synt.
                            syntOutOutLabels.FVal1.FVal2}]
                            ID2} =<:
                        {FS.reified.include 1 Node3.isemsynt
                           .attrs.syntOut.FVal3}
                        {FD.reified.int
                         [{FS.int.min Rule.synt.
                            syntOutOutLabels.FVal1.FVal2}]
                            ID2} =<:
```

```
                              {FS.card Node3.gsynt.model.
                                 daughtersL.FVal3}
                           end
                        else
                           if Node3.gsynt.entry.id ==
                              {FS.int.min Rule.synt.
                                 syntOutOutLabels.FVal1.FVal2}
                              andthen {Not {FS.var.is Result}}
                                 then
                              Result = Node3.isemsynt.attrs.
                                 syntOut.FVal3
                           end
                        end
                     end
                  end
               end
            end
            if {FS.var.is Result} then {FS.value.singl 1 Result}
                 end
         end
      end

      % Constrain the node ids of nodes situated two edges below
      if Rule.synt.syntOutOutLabels \= nil then NextBound
          PreviousBound   in

         for SyntOutOutLabels in Rule.synt.syntOutOutLabels do
             Result FVal1 FVal2
             Valencies1 = {List.nth SyntOutOutLabels 1}
             Valencies2 = {List.nth SyntOutOutLabels 2}
         in
            for Val in {Arity Valencies1} do
               if {FS.int.min Valencies1.Val} == 1 then
                  FVal1 = Val
               end
            end
            for Val in {Arity Valencies2} do
               if {FS.reflect.card Valencies2.Val} == 1 then
                  FVal2 = Val
               end
            end

            for Node2 in {List.reverse Nodes} do
               if Node2.index < Node.index andthen Node2.word \= '
                  *'
                  andthen {Value.isFree PreviousBound} then
                  PreviousBound = Node2.index
               end
            end
            if {Value.isFree PreviousBound} then
               PreviousBound = 1
            end

            for Node2 in Nodes do
               if Node2.index > Node.index andthen Node2.word \= '
                  *'
                  andthen {Value.isFree NextBound} then
                  NextBound = Node2.index
               end
            end
```

```
            if {Value.isFree NextBound} then
               NextBound = {List.length Nodes}
            end

            for Node2 in Nodes do ID in
               if Node2.index == Node.index + 1 then
                  ID = Node2.gsynt.entry.id
                  if {Value.isKinded ID} then
                     if {List.member {FS.int.min Rule.synt.
                        syntOutLabels.FVal1}
                         {FD.reflect.domList ID}} andthen {Not {FS
                            .var.is Result}} then
                        skip
                        {FD.reified.int [{FS.int.min Rule.synt.
                           syntOutLabels.FVal1}] ID} =<:
                        {FS.reified.equal Valencies2.FVal2
                         Node2.isemsynt.attrs.syntOutLabels.FVal2}
                     end
                  else
                     if ID == {FS.int.min Rule.synt.syntOutLabels.
                        FVal1}
                        andthen {Not {FS.var.is Result}} then
                        Result = Node2.isemsynt.attrs.
                           syntOutLabels.FVal2
                     end
                  end
               end
            end
            if {FS.var.is Result} then
               {FS.reified.equal Valencies2.FVal2 Result} =: 1
            end
         end
      end

      % Saturated the synt grams of nodes situated two edges below
      if Rule.synt.syntOutOutGrams \= nil then
         for SyntOutOutGrams in Rule.synt.syntOutOutGrams do
             Result FVal1 FVal2
            Valencies1 = {List.nth SyntOutOutGrams 1}
            Valencies2 = {List.nth SyntOutOutGrams 2}
         in
            for Val in {Arity Valencies1} do
               if {FS.int.min Valencies1.Val} == 1 then
                  FVal1 = Val
               end
            end
            for Val in {Arity Valencies2} do
               if {FS.reflect.card Valencies2.Val} == 1 then
                  FVal2 = Val
               end
            end
            for Node2 in Nodes do ID in
               ID = Node2.gsynt.entry.id
               if {Value.isKinded ID} then
                  if {List.member {FS.int.min Rule.synt.
                     syntOutLabels.FVal1}
                      {FD.reflect.domList ID}} andthen {Not {FS.
                         var.is Result}} then
                     {FD.reified.int [{FS.int.min Rule.synt.
                        syntOutLabels.FVal1}] ID} =<:
```

```
                    {FS.reified.equal Valencies2.FVal2
                     Node2.isemsynt.attrs.syntOutGrams.FVal2}
                  end
               else
                  if ID == {FS.int.min Rule.synt.syntOutLabels.
                     FVal1}
                     andthen {Not {FS.var.is Result}} then
                     Result = Node2.isemsynt.attrs.syntOutGrams.
                        FVal2
                  end
               end
            end
            if {FS.var.is Result} then
               {FS.reified.equal Valencies2.FVal2 Result} =: 1
            end
         end
      end

      % Linking sisters constraint
      if  Rule.linking.sisters \= FS.value.empty then
         {FS.reified.equal Rule.linking.sisters Node.isemsynt.
            attrs.sisters} =: 1
      end

      for Val in {Arity Node.gsem.model.daughtersL} do

         % Hypothesis: we do not have contradictory rules in
            linking

         % Subcategorization (LinkingDaughterEnd)
         if  Rule.linking.subcats.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.subcats.Val
             Node.isemsynt.attrs.subcats.Val} =: 1
         end

         % LinkingBelowStartEnd
         if  Rule.linking.subcats_start.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.subcats_start.Val
             Node.isemsynt.attrs.subcats_start.Val} =: 1
         end

         if  Rule.linking.subcats_end.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.subcats_end.Val
             Node.isemsynt.attrs.subcats_end.Val}  =: 1
         end

         % LinkingAboveStartEnd
         if  Rule.linking.mod_start.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.mod_start.Val
             Node.isemsynt.attrs.mod_start.Val} =: 1
         end

         if  Rule.linking.mod_end.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.mod_end.Val
             Node.isemsynt.attrs.mod_end.Val}  =: 1
         end

         % LinkingMotherEnd
         if  Rule.linking.'mod'.Val \= FS.value.empty then
            {FS.reified.equal Rule.linking.'mod'.Val
```

```
                    Node.isemsynt.attrs.'mod'.Val} =: 1
                end
            end

            % LinkingBelowStartEnd (reverse)
            for Val in {Arity Node.gsynt.model.daughtersL} do

                if  Rule.linking.reverseSubcats_start.Val \= FS.value.
                    empty then
                   {FS.reified.equal Rule.linking.reverseSubcats_start.
                       Val
                    Node.isemsynt.attrs.reverseSubcats_start.Val} =: 1
                end
                if  Rule.linking.reverseSubcats_end.Val \= FS.value.empty
                     then
                   {FS.reified.equal Rule.linking.reverseSubcats_end.Val
                    Node.isemsynt.attrs.reverseSubcats_end.Val}  =: 1
                end
            end

            % Handling of empty nodes
            if Rule.synt.group.before \= nil then
                if {Value.isFree Node.isemsynt.attrs.group.before} then
                   Node.isemsynt.attrs.group.before = {Cell.new Rule.synt
                       .group.before.1}
                else
                   {Cell.assign Node.isemsynt.attrs.group.before
                    {List.flatten
                     {Cell.access Node.isemsynt.attrs.group.before}|
                     Rule.synt.group.before.1}}
                end
            end

            if Rule.synt.group.after \= nil then

                if {Value.isFree Node.isemsynt.attrs.group.after} then
                   Node.isemsynt.attrs.group.after = {Cell.new Rule.synt.
                       group.after.1}
                else
                   {Cell.assign Node.isemsynt.attrs.group.after
                    {List.flatten
                     {Cell.access Node.isemsynt.attrs.group.after}|
                     Rule.synt.group.after.1}}
                end
            end

         end
      end
   end

 %   We verify that all semantic grams are saturated
   for Node in Nodes do
      {FS.reified.equal Node.isemsynt.attrs.semGrams
       {FS.value.make {FS.reflect.lowerBoundList Node.isemsynt.attrs.
          semGrams}}} =: 1
      {FS.subset Node.gsem.attrs.grams Node.isemsynt.attrs.semGrams}
   end
 end
end
```

Annexe D

Glossaire

Actant : Constituants syntaxiques imposés par la valence de certaines classes lexicales.

Acte de langage : (ou de parole) Phénomène pragmatique, étudié notamment par Austin et Searle, par lequel une expression linguistique permet d'influencer la réalité : demandes, ordres, promesses, attitudes liées à un comportement social.

Affixe : Morphe non autonome, qui est destiné à se combiner avec d'autres signes morphologiques au sein d'un mot-forme.

Ambiguité : Propriété d'une expression linguistique qui peut être associée à plus d'un sens. Il existe des ambiguités lexicales et des ambiguités syntaxiques.

Antonyme : Deux lexies L_1 et L_2 appartenant à la même partie du discours sont antonymes si les sens 'L_1' et 'L_2' se distinguent par la négation ou, plus généralement, la mise en opposition d'une de leurs composantes.

Apposition : Figure de style où deux éléments sont placés côte-à-côte, le second élément servant à définir ou modifier le premier.

Clivage : Mise en relief d'un élément en l'encadrant de formules telles que "`C'est ... que`".

Clitique : Désigne une forme linguistique s'employant seulement dans une position non-accentuée (i.e. est prosodiquement regroupée avec le mot précédent ou suivant).

Collocation : L'expression AB (ou BA), formée des lexies A et B, est une collocation si, pour produire cette expression, le locuteur sélectionne A (appelé base de la collocation) librement d'après son sens 'A', alors qu'il sélectionne B (appelé collocatif) pour exprimer un sens 'C' en fonction de A. Exemple : "$\texttt{pleuvoir}_{(=A)}$ $\texttt{des cordes}_{(=B)}$".

Connotation : Conenu informationnel associé à une lexie qui, contrairement au sens, n'est pas nécessairement exprimé quand cette lexie est utilisée. Ex : tigre connote en français la férocité.

Copule : Mot qui lie un attribut au sujet d'une proposition. Le verbe "`être`" est la copule la plus fréquente.

Corpus : Collection de données linguistiques écrites ou orales utilisées pour étudier et décrire divers phénomènes linguistiques ou pour vérifier des hypothèses sur le langage.

Déictique : Expressions dont le sens ne peut se décrire qu'en mentionnant une entité impliquée dans la situation de communication langagière (ex : "ici", "moi", "demain", etc.)

Dénotation : (d'une expression linguistique) Relation entre l'expression et la classe des entités que ce mot représente. Cette relation est stable et indépendante de l'usage du mot.

Dépendance : Relation syntaxique asymétrique entre deux mots d'une phrase. Caractérisation théorique possible de (Mel'čuk, 1988), basée sur la constituance : La tête syntaxique d'un constituant est l'élément qui détermine la valence passive de ce constituant, c'est-à-dire l'élément qui contrôle la distribution de ce constituant.

Dérivation : Mécanisme morphologique qui consiste en la combinaison d'un radical et d'un affixe - appelé affixe dérivationnel, ayant les trois propriétés suivantes : (1) son signifié est moins général et moins abstrait que celui d'un affixe flexionnel - il s'apparente au signifié d'une lexie ; (2) l'expression de son signifié correspond normalement à un choix libre du locuteur, qui décide de communiquer le signifié en question ; (3) Sa combinaison avec le radical d'une lexie donne un mot-forme qui est associé à une autre lexie.

Diachronique : Qui concerne l'évolution historique d'un élément linguistique.

Diathèse : Correspondance entre les arguments sémantiques et les actants syntaxiques d'un verbe.

Flexion : Mécanisme morphologique qui consiste en la combinaison d'un radical et d'un affixe - appelé affixe flexionnel - ayant les trois propriétés suivantes : (1) son signifié est très général, plutôt abstrait et appartient nécessairement à un petit ensemble de signifiés mutuellement exclusifs appelé catégorie flexionnelle - par exemple, la catégorie flexionnelle de nombre en français, qui regroupe les deux signifiés "singulier - pluriel" ; (2) L'expression de sa catégorie flexionnelle est imposée par la langue - par ex. tout nom frnaçais doit être employé soit au singulier soit au pluriel : cela fait de la flexion un mécanisme régulier ; (3) Sa combinaison avec le radical d'une lexie donne un mot-forme associé à la même lexie.

Fonction lexicale f : décrit une relation entre une lexie L - l'argument de f - et un ensemble de lexies ou d'expressions figées appelé la valeur de l'application de f à la lexie L. La fonction lexicale f est telle que : (1) l'expression $f(L)$ représente l'applicaton de f à la lexie L ; (2) chaque élément de la valeur de $f(L)$ est lié à L de la même façon. Il existe autant de fonctions lexicales qu'il existe de type de liens lexicaux et chaque fonction lexicale est identifiée par un nom particulier. On distingue encore les FL paradigmatiques, comme Syn ou $Syn_{\subset}$, et les FL syntagmatiques, comme $Magn$ ou Bon.

Grammaires catégorielles : Famille de formalismes syntaxiques basés sur le principe de compositionnalité et considérant les constituants syntaxiques comme une combinaison de fonctions.

Grammène : Information grammaticale associée à une unité lexicale.

Head-Driven Phrase Structure Grammar : Théorie linguistique générative non-dérivationnelle, successeur de GPSG (Generalised Phrase Structure Grammar). HPSG est un formalisme lexicalisé et basé sur les contraintes. Les structures de traits typées sont au coeur du formalisme.

Homonymie : Situation où deux lexies distinctes sont associées au même signifiant alors qu'elles n'entretiennent aucune relation de sens.

Hyperonyme : On dit que la lexie L_{hyper} est un hyperonyme de la lexie L_{hypo} lorsque la relation sémantique qui les unit possède les deux caractéristiques suivantes : (1) le sens 'L_{hyper}' est inclus dans le sens 'L_{hypo}' ; (2) 'L_{hypo}' peut être considéré comme un cas particulier de 'L_{hyper}'. La lexie L_{hypo} est quant à elle appelée hyponyme de L_{hyper}.

Intransitif : Non transitif

Langue : Une langue est un système de signes conventionnels et de règles de combinaison de ces signes, qui forment un tout complexe et structuré.

Langage : 'Faculté humaine de communiquer des idées au moyen de la langue.

Lexème : Lexie regroupant des mots-formes qui ne se distinguent que par la flexion.

Lexical Functional Grammar : Formalisme linguistique issu de la grammaire générative. LFG considère le langage comme étant constitué de dimensions relationnelles multiples. Chacune de ces dimensions est représentée comme une structure distincte avec ses propres règles, concepts, et forme. Les deux plus importantes sont la f-structure (fonctions grammaticales) et la c-structure (constituants syntaxiques).

Lexicologie : Branche de la linguistique qui étudie les propriétés des unités lexicales de la langue, appelées lexies.

Lexie : (ou unité lexicale) est un regroupement (1) de mots-formes ou (2) de constructions linguistiques que seule distingue la flexion. Dans le premier cas, il s'agit de lexèmes, dans le second cas, de locutions. Chaque lexie (lexème ou locution) est associée à un sens donné, que l'on retrouve dans le signifié de chacun des signes (mots-formes ou constructions linguistiques) auxquels elle correspond.

Lexie pleine : Lexie possédant un sens propre.

Lexie vide : Lexie ne possédant pas de sens par elle-même (par ex., une préposition).

Lexique : Entité théorique qui correspond à l'ensemble des lexies d'une langue.

Liens paradigmatiques : connectent les lexies à l'intérieur du lexique par des relations sémantiques, éventuellement accompagnées de relations morphologiques.

Liens syntagmatiques : connectent les lexies à l'intérieur de la phrase par des relations de combinatoire.

Linguistique Cognitive : Ecole linguistique considérant l'essence du langage comme basée sur des facultés mentales spécialisées et développées au fil de l'évolution humaine, et insistant sur le fait que le langage fait partie intégrante de la cognition. Les principes théoriques doivent être fondées sur l'expérience humaine et la manière dont les êtres humaines perçoivent et conceptualisent le monde qui les entoure.

Locution : Lexie regroupant des expressions linguistiques complexes que seule distingue la flexion.

Mode : (d'un verbe) Trait grammatical dénotant la manière dont le locutre précent le procès.

Modifieur : Elément (optionnel) chargé de modifier le sens d'un autre élément, par exemple un adjectif pour un nom, ou un adverbe pour un verbe.

Mot-forme : Signe linguistique ayant les deux propriétés suivantes : (1) Autonomie de fonctionnement ; (2) Cohésion interne.

Morphe : Signe linguistique ayant les deux propriétés suivantes : (1) il possède un signifiant qui est un segment de la chaîne parlée, (2) c'est un signe élémentaire, il ne peut être décomposé en plusieurs autres signes de la langue.

Morphème : Regroupement de morphes “alternatifs” ayant le même signifié.

Morphologie : Etude de la structure interne des mots-formes.

Oblique : Objet du verbe introduit par une préposition.

Paraphrases : Propriété de deux expressions linguistiques ayant approx. le même sens.

Parties du discours : Classes générales à l'intérieur desquelles sont regroupées les lexies de la langue en fonction de leurs propriétés grammaticales. On distingue typiquement les classes ouvertes, contenant un nombre indéfini d'éléments : verbes, substantifs, adjectifs, adverbes ; et les classes fermées : auxiliaires, pronoms, déterminants, conjonctions, prépositions.

Parole : Actualisation des langues dans des actes de communication qui impliquent un locuteur et un destinataire.

Partitif : Cas qui exprime la partie d'un tout.

Phone : Son vocal.

Phonétique : Etude de la production, la transmission et la réception des sons vocaux.

Phonologie : l'étude des phonèmes - les unités minimales distinctives d'une langue -, du point de vue de leur fonction dans une langue donnée et des relations d'opposition et de contraste qu'ils ont dans le système des sons de cette langue.

Polysémie : Propriété d'un vocable donné de contenir plus d'une lexie.

Procès : Ce que le verbe peut affirmer du sujet (état, devenir, action).

Radical : (d'une lexie) est son support morphologique. C'est l'élément morphologique qui porte le signifié associé en propre à cette lexie.

Rection : Désigne la propriété qu'ont certains unités linguistiques d'être accompagnées d'un complément, obligatoire ou optionnel.

Référent : (d'une expression linguistique) Élément du "monde", de la réalité que cette expression permet de désigner dans un context donné de parole (i.e. d'utilisation de la langue)

Régime : (=sous-catégorisation) Mot, groupe de mots régi par un autre (par ex. le fait qu'un verbe requiert un objet direct, indirect, etc)

Saillance : Importance relative ou proéminence d'un signe.

Sémantème : Unité sémantique.

Sens : (d'une expression linguistique) Propriété qu'elle partage avec toutes ses paraphrases.

Sémantique : Etude des sens et de leur organisation au sein des messages que l'on peut exprimer dans une langue.

Signe : Au sens large, association entre une idée (le contenu du signe) et une forme.

Signe linguistique : Association indissoluble ("entité à deux faces") entre un contenu, appelé signifié, et une forme, appelée signifiant. (de Saussure, 1916) définit plusieurs propriétés de celui-ci : arbitrarité du signe, linéarité du signifiant, etc.

Signifiant : Une des facettes du signe linguistique, représentant l'image acoustique de celui-ci.

Signifié : Une des facettes du signe linguistique, représentant le concept, le sens de celui-ci.

Structure communicative : (angl. *information structure*) Spécification de la façon dont le locuteur veut présenter l'information qu'il communique.

Synchronique : Par opposition à diachronique, concerne l'utilisation d'un élément à un moment donné, fixe de son évolution (très souvent, son utilisation actuelle)

Synonyme : Deux lexies L_1 et L_2 appartenant à la même partie du discours sont synonymes si elle ont approximativement le même sens $'L_1' \approx' L_2'$.

Syntaxe : Etude de la structure des phrases.

Topicalisation : Action de déplacer le thème (ou le "topic") au début de la phrase.

Transitif : Se dit d'un verbe suivi d'un complément d'objet direct

Tree Adjoining grammar : Formalisme grammatical, proche des grammaires de réécriture hors-contexte, mais dont l'unité élémentaire de réécriture est l'arbre plutôt que le symbole. Les TAG possèdent deux opérations : l'adjonction et la substitution.

Valence : Capacité d'un verbe à prendre un nombre et type spécifique d'arguments.

Verbes de contrôle : Un verbe de contrôle est un verbe dont la représentation sémantique est un prédicat à au moins deux arguments, dont l'un est un verbe et l'autre est à la fois argument du verbe de contrôle et du verbe "contrôlé". Exemple : "`essayer`" dans "`Pierre essaye de dormir`".

Verbes de montée : Sémantème verbal s'exprimant par un prédicat dont l'un des arguments est un verbe dont un des arguments va "monter" en position de sujet syntaxique du verbe de montée. Exemple prototypique : "`sembler`" dans "`Pierre semble dormir`".

Verbe support : Collocatif verbal sémantiquement vide dont la fonction linguistique est de "verbaliser" une base nominale, i.e. de la faire fonctionner dans la phrase comme si elle était elle-même un verbe. Ex : "`éprouver`" avec "`regret`".

Vocable : Regroupement de lexies ayant les deux propriétés suivantes : (1) Elle sont associées aux mêmes signifiants ; (2) Elles présentent un lien sémantique évident.

Vocabulaire : (d'un texte) Ensemble des lexies utilisées dans un texte.

Voix : (d'un verbe) : Trait grammatical décrivant commen s'organisent les rôles sémantiques dévolus aux actants par rapport au procès verbal.

Sources : (Crystal, 1991; Degand, 2006; Fairon, 2004; Polguère, 2003; Tesnière, 1959; O'Grady *et al.*, 1996; Larousse, 1995)

Bibliographie

AJDUKIEWICZ K. (1935). « Die syntaktische Konnexität ». In *Studia Philosophica*, 1, 1–27.

BADER R., FOELDESI C., PFEIFFER U. et STEIGNER J. (2004). *Modellierung grammatischer Phänomene der deutschen Sprache mit Topologischer Dependenzgrammatik.* Rapport interne, Saarland University. Softwareprojekt.

BECH G. (1955). *Studien über das deutsche Verbum infinitum.* Linguistische Arbeiten 139, Niemeyer, Tübingen, 2e 1983 edition.

BLANCHE-BENVENISTE C. (1991). *Le français parlé : études grammaticales.* Editions du CNRS, Paris.

BOHNET B., LANGJAHR A. et WANNER L. (2000). « A development Environment for an MTT-Based Sentence Generator ». In *Proceedings of the First International Natural Language Generation Conference.*

BOJAR O. (2004). « Problems of Inducing Large Coverage Constraint-Based Dependency Grammar ». In *Proceedings of the International Workshop on Constraint Solving and Language Processing*, Roskilde/DEN.

BONFANTE G., GUILLAUME B. et PERRIER G. (2003). « Analyse syntaxique électrostatique ». In *Revue TAL*, Vol. 44, No. 3.

BONFANTE G., GUILLAUME B. et PERRIER G. (2004). « Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation ». In *Actes CoLing*, Genève. p. 303–309.

J. Bresnan, éd. (1982). *The mental representation of Grammatical Relations.* MIT Press, Cambridge.

CANDITO M.-H. (1999). *Organisation modulaire et paramétrable de grammaires électroniques lexicalisées. Application au français et à l'italien.* PhD thesis, Université Paris 7.

CHOMSKY N. (1957). *Syntactic Structure.* MIT Press, Cambridge.

CHOMSKY N. (1965). *Aspects of the Theory of Syntax.* MIT Press, Cambridge.

CHOMSKY N. (1981). *Lectures on government and Binding : the Pisa lectures.* Foris Publications.

CLARK S. et CURRAN J. (2004). « The importance of supertagging for wide-coverage CCG parsing ». In *Proceedings of COLING 2004*. p. 282–288.

CRYSTAL D. (1991). *A dictionary of linguistics and phonetics.* Basil Blackwell, Cambridge, 3e edition.

DE SAUSSURE F. (1916). *Cours de Linguistique Générale.* Payot, Paris, 3e edition.

DEBUSMANN R. (2001). « A Declarative Grammar Formalism For Dependency Grammar ». Master's thesis, Saarland University.

DEBUSMANN R. (2004). « Multiword expressions as dependency subgraphs ». In *Proceedings of the ACL 2004 Workshop on Multiword Expressions : Integrating Processing*, Barcelona/ESP.

DEBUSMANN R. (2006). *Extensible Dependency Grammar : A Modular Grammar Formalism Based On Multigraph Description.* PhD thesis, Saarland University.

DEBUSMANN R. et DUCHIER D. (2005). *Manual of the XDG Development Kit.*

DEBUSMANN R., DUCHIER D., KOLLER A., KUHLMANN M., SMOLKA G. et THATER S. (2004a). « A Relational Syntax-Semantics Interface Based on Dependency Grammar ». In *Proceedings of the COLING 2004 Conference*, Geneva/SUI.

DEBUSMANN R., DUCHIER D. et KRUIJFF G.-J. M. (2004b). « Extensible Dependency Grammar : A New Methodology ». In *Proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar*, Geneva/SUI.

DEBUSMANN R., DUCHIER D. et KUHLMANN M. (2004c). « Multi-dimensional Graph Configuration for Natural Language Processing ». In *Proceedings of the International Workshop on Constraint Solving and Language Processing*, Roskilde/DEN : Springer.

DEBUSMANN R., DUCHIER D., KUHLMANN M. et THATER S. (2004d). « TAG Parsing as Model Enumeration ». In *Proceedings of the TAG+7 Workshop*, Vancouver/CAN.

DEBUSMANN R., DUCHIER D. et NIEHREN J. (2004e). « The XDG Grammar Development Kit ». In *Proceedings of the MOZ04 Conference*, volume 3389 of *Lecture Notes in Computer Science*, Charleroi/BEL : Springer, p. 190–201.

DEBUSMANN R., DUCHIER D. et ROSSBERG A. (2005a). « Modular Grammar Design with Typed Parametric Principles ». In *Proceedings of FG-MOL 2005*, Edinburgh/UK.

DEBUSMANN R., POSTOLACHE O. et TRAAT M. (2005b). « A Modular Account of Information Structure in Extensible Dependency Grammar ». In *Proceedings of the CICLING 2005 Conference*, Mexico City/MEX : Springer.

DEBUSMANN R. et SMOLKA G. (2006). « Multi-dimensional Dependency Grammar as Multigraph Description ». In *Proceedings of FLAIRS-19*, Melbourne Beach/US : AAAI.

DEGAND L. (2006). « Notes du cours "Introduction à la Sémantique" ». Université Catholique de Louvain.

DIENES P., KOLLER A. et KUHLMANN M. (2003). « Statistical A-Star Dependency Parsing ». In D. Duchier, éd., *Prospects and Advances in the Syntax/Semantics Interface*, Nancy/FRA : LORIA, p. 85–89.

DOWTY D. R. (1989). « On the semantic content of the notion of "thematic role" ». In G. Chierchia, B. H. Partee et R. Turner, éds., *Properties, Types and Meaning*, volume 2, Dortrecht : Kluwer, p. 69–129.

DRACH E. (1937). *Grundgedanken der deutschen Satzlehre*. Diesterweg, Frankfurt.

DUCHIER D. (1999). « Axiomatizing Dependency Parsing Using Set Constraints ». In *Proceedings of MOL6*, Orlando/USA. p. 115–126.

DUCHIER D. (2003). « Configuration Of Labeled Trees Under Lexicalized Constraints And Principles ». In *Journal of Research on Language and Computation*.

DUCHIER D., GARDENT C. et NIEHREN J. (1998). *Concurrent Constraint Programming in Oz for Natural Language Processing*. Programming Systems Lab, Universität des Saarlandes, Germany. Available at http ://www.ps.uni-sb.de/Papers.

DUCHIER D., ROUX J. L. et PARMENTIER Y. (2004). « The Metagrammar Compiler : An NLP Application with a Multi-Paradigm Architecture ». In P. V. Roy, éd., *MOZ*, volume 3389 of *Lecture Notes in Computer Science* : Springer, p. 175–187.

DUCHIER D. et THATER S. (1999). « Parsing with tree descriptions : a constraint-based approach ». In *NLULP 1999 (Natural Language Understanding and Logic Programming), Las Cruces, NM.*

ENGEL U. (1988). *Deutsche Grammatik*. Groos, Heidelberg.

FAIRON C. (2004). « Notes du cours "Introduction au Traitement Automatique du Langage" ». Université Catholique de Louvain.

GAIFMAN H. (1965). « Dependency systems and phrase-structure systems ». In *Information and Control*, 18, 304–337. RM-2315.

GERDES K. et KAHANE S. (2001). « Word order in German : a formal dependency grammar using a topological hierarchy ». In *ACL 2001*, Toulouse.

GERDES K. et KAHANE S. (2004). « L'amas verbal au coeur d'une modélisation topologique du français ». In *Actes Journées de la syntaxe - Ordre des mots dans la phrase française, positions et topologie*, Bordeaux. p. 8 p.

GERDES K. et KAHANE S. (2006). « Phrasing it differently ». In L. Wanner, éd., *Papers in Meaning-Text Theory in honour of Igor Mel'čuk* : Benjamins. (sous presse).

GOLDBERG A. (1995). *A Construction Grammar Approach to Argument Structure.* University of Chicago Press.

GROSS G. (1996). *Les expressions figées en français.* Ophrys.

GROSS M. (1975). *Méthodes en syntaxe.* Hermann, Paris.

HAGÈGE C. (1986). *Homme des paroles.* Fayard, Paris.

HAYS D. (1960). *Grouping and dependency theories.* Rapport interne, Rand Corporation. RM-2646.

HUDSON R. (1990). *English Word Grammar.* Blackwell, Oxford.

JACKENDOFF R. (2002). *Foundations of Language.* Oxford University Press.

JESPERSEN O. (1924). *Philosophy of Grammar.* Allen & Unwin, Londres.

JOSHI A. (1987). « Introduction to Tree Adjoining Grammar ». In R. Manaster, éd., *The Mathematics of Language*, Amsterdam : Benjamins, p. 87–114.

JURAFSKY D. et MARTIN J. H. (2000). *Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall.

KAHANE S. (1997). « Bubble trees and syntactic representations ». In Becker et Krieger, éds., *Proc. 5th Meeting of the Mathematics of Language (MOL5)*, Saarbrücken : DFKI. p. 70–76.

KAHANE S. (2001). « Grammaires de dépendance formelles et Théorie Sens-Texte ». In *TALN 2001, Tours.*

KAHANE S. (2002). *Grammaire d'Unification Sens-Texte : vers un modèle mathématique articulé de la langue.* Habilitation à diriger des recherches, Université Paris 7.

KAHANE S. (2003a). « On the status of the deep-syntactic structure ». In *Proc. of the 1st Int. Conf. on Meaning-Text Theory*, Paris.

KAHANE S. (2003b). « What signs for the semantics-syntax interface ? ». In D. Duchier, éd., *Workshop Prospects and Advances in the Syntax/Semantics Interface*, Nancy/FRA : LORIA.

KAHANE S. (2004). « Grammaires d'Unification Polarisées ». In *Actes TALN, Fès*, p. 233–242.

KAHANE S. (2005). « Structure des représentations logiques, polarisation et sous-spécification ». In *Actes TALN*, Dourdan.

KAHANE S. et LAREAU F. (2005). « Grammaire d'Unification Sens-Texte : modularité et polarisation ». In *Actes TALN*, Dourdan. p. 23–32.

KAHANE S., NASR A. et RAMBOW O. (1998). « Pseudo-projectivity : a polynomially parsable non-projective dependency grammar ». In *Proc. COLING-ACL'98*, Montréal. p. 646–652.

KAY M. (1979). « Functional Grammars ». In *Proc. 5th Annual Meeting of the Berkeley Linguistic Society*, Berkeley. p. 142–158.

KLAVANS J. et RESNIK P. (1996). *The balancing act : combining symbolic and statistical approaches to language.* MIT Press.

KOLLER A., DEBUSMANN R., GABSDIL M. et STRIEGNITZ K. (2004). « Put my galakmid coin into the dispenser and kick it : Computational Linguistics and Theorem Proving in a Computer Game ». In *Journal of Logic, Language and Information*, 13 (2), 187–206.

KOLLER A. et STRIEGNITZ K. (2002). « Generation as Dependency Parsing ». In *Proceedings of the 40th ACL*, Philadelphia/USA.

KORTHALS C. et DEBUSMANN R. (2002). « Linking syntactic and semantic arguments in a dependency-based formalism ». In *Proceedings of he COLING 2002 Conference*, Taipei/TW.

LAREAU F. (2004). *Vers un modèle formel de la conjugaison française dans le cadre des grammaires d'unification sens-texte polarisées.* Document pour l'examen général de synthèse, Université de Montréal.

Larousse, éd. (1995). *Le Petit Larousse Illustré.* Larousse.

MEL'ČUK I. (1970). « Towards a Functioning Model of Language ». In M. Bierwisch et K. E. Heidolph, éds., *Progress in Linguistics*. The Hague, Paris : Mouton.

MEL'ČUK I. (1993). *Cours de Morphologie Générale*, volume 1 : le mot. Presses de l'Université de Montréal.

MEL'ČUK I. (1997). « Vers une linguistique sens-texte ». In *Lecon inaugurale au Collège de France. Chaire internationale.*

MEL'ČUK I. et PERTSOV N. (1987). *Surface Syntax of English.* John Benjamins, Amsterdam.

MEL'ČUK I. A. (1974). *Opyt teorii lingvističeskix modelej "Smysl — Tekst".* Nauka, Moscow.

MEL'ČUK I. A. (1988). *Dependency Syntax : Theory and Practice.* State University of New York Press, Albany.

MEL'ČUK I. A. (1995). *Introduction à la lexicologie explicative et combinatoire.* Duculot, Paris.

MEL'ČUK I. A., ARBATCHEWSKY-JUMARIE N., ELNITSKY L. et LESSARD A. (1984). *Dictionnaire explicatif et combinatoire du francais contemporain.* Presses de l'Université de Montréal, Montréal, Canada. Volume 1.

MEL'ČUK I. A., ARBATCHEWSKY-JUMARIE N., ELNITSKY L. et LESSARD A. (1988). *Dictionnaire explicatif et combinatoire du francais contemporain.* Presses de l'Université de Montréal, Montréal, Canada. Volume 2.

MEL'ČUK I. A., ARBATCHEWSKY-JUMARIE N., ELNITSKY L. et LESSARD A. (1992). *Dictionnaire explicatif et combinatoire du francais contemporain.* Presses de l'Université de Montréal, Montréal, Canada. Volume 3.

MEL'ČUK I. A. et POLGUÈRE A. (1987). « A Formal Lexicon in the Meaning-Text Theory (or How to Do Lexica with Words) ». In *Computational Linguistics*, 13 (3-4), 276–289.

MEL'ČUK, IGOR A. et ŽHOLKOVSKIJ A. K. (1970). « Towards a Functioning "Meaning-Text" Model of Language ». In *Linguistics*, 57, 10–47.

MONTAGUE R. (1974). « The proper treatment of quantification in ordinary english ». In *Formal Philosophy. Selected Papers of Richard Montague*, New Haven and London : Yale University Press, p. 247–271.

NARENDRANATH R. (2004). *Evaluation of the Stochastic Extension of a Constraint-Based Dependency Parser.* Rapport interne, Saarland University. Bachelorarbeit.

NASR A. (1995). « A formalism and a parser for lexicalised dependency grammars ». In *4th Int. Workshop on Parsing Technologies* : State University of New York Press.

O'GRADY W., DOBROVOLSKY M. et KATAMBA F. (1996). *Contemporary Linguistics : An Introduction.* Longman, 3e edition.

OWENS J. (1988). *Foundations of Grammar : An Introduction to Mediaeval Arabic Grammatical Theory.* Benjamins, Amsterdam.

PELIZZONI J. et DAS GRACAS VOLPE NUNES M. (2005). « N :M Mapping in XDG - The Case for Upgrading Groups ». In *Proceedings of the International Workshop on Constraint Solving and Language Processing*, Sitges/ES.

PERRIER G. (1999). « Description d'arbres avec polarités : les Grammaires d'Interaction ». In *TALN 2002, Nancy.*

POLGUÈRE A. (1998). « La théorie Sens-Texte ». In *Dialangue*, 8-9.

POLGUÈRE A. (2003). *Lexicologie et sémantique lexicale.* Presses de l'Université de Montréal.

POLLARD C. et SAG I. (1994). *Head-Driven Phrase Structure Grammar.* Stanford CSLI.

PULLUM G. K. et SCHOLZ B. C. (2001). « On the distinction between model-theoretic and generative-enumerative syntactic frameworks ». In P. de Groote, G. Morril et C. Retoré, éds., *Logical Aspects of Computional Linguistics : 4th International Conference*, Berlin : Springer, p. 17–43.

Robinson J. (1970). « Dependency structures and transformational rules ». In *Language*, 46, 259–285.

Rogers J. (1998). « A descriptive characterization of tree-adjoining languages ». In *Proceedings of COLING/ACL 1998*, Montréal.

Russell S. et Norvig P. (2003). *Artificial Intelligence : A Modern Approach.* Prentice Hall.

Schulte C. et Smolka G. (1999). *Finite Domain Constraint Programming in Oz.*

Schulte C. et Stuckey P. (2004). « Speeding up constraint propagation ». In *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258, Toronto : Springer-Verlag, p. 619–633.

Sgall P., Hajicová E. et Panenová J. (1986). *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects.* Reidel, Dordrecht.

Sommerville I. (2004). *Software Engineering.* Pearson - Addison Wesley, 7e edition.

Steele S. M. (1978). « Word Order Variation : A typological study ». In J. Greenberg, éd., *Universals of Human Language*, Stanford : Stanford University Press, p. 585–624.

Tesnière L. (1934). « Comment construire une syntaxe ». In *Bulletin de la Faculté des Lettres de Strasbourg*, Vol. 7, 12e année, 219–229.

Tesnière L. (1959). *Eléments de syntaxe structurale.* Klincksieck, 2e edition.

Uzkoreit H. (2004). « What is Computational Linguistics ? ». Court texte présentant le domaine et ses applications. *URL* : `http ://www.coli.uni-saarland.de/ hansu/what_is_cl.html`.

Žolkovskij A. K. et Mel'čuk I. A. (1965). « O vozmožnom metode i instrumentax semantičeskogo sinteza ». In *Naučno-texničeskaja informacija.*

Žolkovskij A. K. et Mel'čuk I. A. (1967). « O semantičeskom sinteze ». In *Problemy kibernetiki*, 19, 177–238.

Woods W. (1975). « What's in a link : Foundations for semantic networks ». In *Representation and Understanding-Studies in Cognitive Science*, p. 55–82.