Formalisation, implémentation et exploitation d'une hiérarchie objet intono-syntaxique

"Étude sur un treebank de français oral spontané"

Julie Belião

Mémoire d'Ingénierie Linguistique

Sous la direction de Sylvain Kahane

Mémoire de Master Recherche en Phonétique Expérimentale

Sous la direction conjointe de Cédric Gendrot et de Martine Adda-Decker

Résumé

Ce mémoire se concentre sur la problématique que soulève le traitement de corpus oraux. Ces travaux ont été réalisés dans le cadre du projet Rhapsodie qui vise à étudier l'intono-syntaxe du français . Le corpus est constitué de 57 échantillons et correspondant à 3h de français spontané.

Sur l'ensemble du corpus ont été effectuées des annotations syntaxiques et prosodiques qui reposent chacune sur une structure hiérarchique différentes. De manière à pouvoir les intégrer et à effectuer des études intono-syntaxiques, l'introduction d'un formalisme spécifique est nécessaire.

Dans un premier temps, je présente mes efforts en vue de produire une structure hiérarchique objet enrichie, appelée multi-arbre, de l'ensemble des données syntaxiques et prosodiques du corpus. Chaque échantillon est modélisé comme une interaction d'objets de différents types. Sa transcription tout d'abord est considérée comme une suite d'objets "*mots*". L'annotation syntaxique consiste à associer ces "*mots*" à des unités syntaxiques. Un nœud est caractérisée par son type, comme "*noyau*", "*unité illocutoire*", etc. et entretient avec les autres nœuds des relations hiérarchiques. Par exemple, un nœud de type "*noyau*" pourra avoir pour père un nœud de type "*unité illocutoire*". L'annotation prosodique se concentre sur un modèle hiérarchique où un échantillon, objet de type *"TextGrid"*, est constitué de plusieurs objets de type *"Tier"*. Chaque tier contient contient différents *"Intervals",* qui contiennent des informations d'un type prosodique particulier, comme "*le ralentissement*", "*les proéminences*", etc. Je réunis ces hiérarchies en associant à chaque mot de la transcription ses différents objets Intervals. Pour construire l'arborescence de ces objets, il faut analyser le langage de balisage utilisé pour l'annotation syntaxique. Ce travail a donné lieu à la création d'un analyseur ad-hoc, capable de générer un multi-arbre à partir de la transcription annotée. Muni de cette structure de données, on peut générer des segments analysables par un parseur syntaxique automatique. On récupère dès lors des objets "*tokens*" et *"lexèmes"* ainsi que des liens de dépendances syntaxiques entre eux. Ces objets sont intégrés au multi-arbre, de manière à offrir une architecture faisant interagir les TextGrids, leur Tiers et Intervals, avec les unités syntaxiques de l'annotation et les Token et Lexemes issus de l'analyse automatique. Ma contribution consiste en la création du multi-arbre comportant les différents niveaux d'analyse syntaxique et prosodique.

Dans un deuxième temps je propose une étude intono-syntaxique mettant à profit ces structures de données. Le principe d'une étude intono-syntaxique est d'étudier certains phénomènes syntaxiques par le biais de leur propriétés prosodiques et vice versa, on peut ainsi étudier la f_0 au sein chaque unité illocutoire par exemple. Le phénomène syntaxique étudié est l'entassement. L'entassement est une dimension orthogonale à la rection, on dit qu'un segment Y s'entasse sur un segment X si Y vient occuper la même position régie que X, il peut également s'agir de coordinations comme des reformulations ou des disfluences. Par exemple si on considère l'unité illocutoire *"^mais il y avait des gens comme { Gide | Claudel | Valéry | Malraux } //"*, les mots entassés *Gide*, *Claudel*, *Valéry* et *Malraux* sont tous substituables les uns aux autres car ils peuvent occuper la même place régie. Mes travaux se concentrent sur les segments constitutifs des entassements et leurs caractéristiques prosodiques. L'unité minimale choisie est la syllabe et les critères prosodiques retenus pour l'étude sont les valeurs de f_0, les mesure normalisées d'allongement syllabique et l'annotation des proéminences. Je montre ainsi que l'allongement syllabique est plus important aux frontières droites des constituants de l'entassement et va souvent de pair avec une augmentation de la f_0. Enfin par des mesures différentielles des critères prosodiques retenus, je mets en évidence l'existence de transitions marquées entre les différents constituants des entassements. Cette étude est rendue possible par l'exploitation de structure objet en multi-arbre présentée avant et par un ensemble de requêtes statistiques effectuées sur des matrices extraites du multi-arbre.

Remerciements

Je remercie en premier lieu Paola Pietrandrea et Sylvain Kahane, qui par leur patience et leur disponibilité m'ont maintes fois relue et conseillée, merci pour les relectures du dernier moment, du dimanche et de la presque nuit. Je remercie également Cédric Gendrot qui m'a permis de faire ce master, m'a encadrée pour la partie prosodie.

Un merci chaleureux à toute l'équipe de Rhapsodie, à Kim Gerdes qui m'a fait confiance en me permettant de travailler pour le projet dès la licence 3, à Anne Lacheret la directrice du projet et enfin à toute l'équipe syntaxe qui m'a plus appris au cours des fameuses"réunions syntaxe" que je n'avais appris jusque là en cours, merci Christophe, Anne, Noalig, Frédérique, Nathalie, Jeanne-Marie, José, Julie, Bernard, Florence et ceux que j'oublie (mille excuses).

Enfin un merci tout particulier à Antoine qui m'a en tout premier lieu supportée, mais qui m'a aussi enseigné l'algorithmique et qui m'a permis d'avancer en toutes circonstances par son aide et ses encouragements.

Table des matières

Table des matières **4**

Table des figures **8**

I Rhapsodie : *"un treebank de français oral spontané"* **13**

1 Contexte et objectifs **15**
1.1 Un défi pour l'analyse syntaxique 15
1.2 Une analyse prosodique complexe 15

2 Analyse syntaxique de corpus : Phénomènes micro et macro syntaxiques **17**
2.1 Unités rectionnelles 17
2.2 Unités Illocutoires 17
2.3 Phénomènes annotés manuellement et balises associées 18
2.3.1 Balisage des UI 18
2.3.2 Balisage des enchâssements et parenthèses 18
2.3.3 Balisage des marqueurs d'UR et de Composante illocutoire 19
Pré-noyau 19
Post-noyau 20
In-noyau 20
Introducteurs 20
2.3.4 Balisage des Entassements 21
2.4 Hiérarchies syntaxiques du corpus 21

3 Analyse Prosodique du corpus **25**
3.1 Méthodologie 25
3.2 Mesures de durées (ralentissements) 25
3.3 Proéminences 26
3.4 Disfluences prosodique et hésitations 29
3.5 Stylisation des Contours Intonatifs, Mesures de registres locaux, Transcription Tonale et Calcul des valeurs moyennes de F0 29
3.5.1 Introduction 29
3.5.2 Registre et Contour 30
3.5.3 Transcription 30
3.5.4 Calcule des valeurs moyennes de f_0 des syllabes 31
3.6 Hiérarchie prosodique du corpus 32

II Structures de données hiérarchiques pour l'analyse du Français spontané **33**

4 Introduction **37**

5 Formalisme adopté **39**
5.1 Une approche objet . . . 39
5.2 Les différentes classes considérées . . . 40
5.3 Word . . . 45
5.3.1 Attributs . . . 46
5.3.2 Méthodes . . . 46
5.4 Node . . . 46
5.4.1 Attributs . . . 47
5.4.2 Méthodes . . . 47
Méthodes mineures . . . 47
Méthodes majeures . . . 48
5.5 Token . . . 49
5.5.1 Attributs . . . 49
5.5.2 Méthodes . . . 50
5.6 Lexeme . . . 50
5.6.1 Attributs . . . 50
5.6.2 Méthodes . . . 51
5.7 Dependency . . . 51
5.7.1 Attributs . . . 51
5.7.2 Méthodes . . . 51
5.8 TextGrid, Tier et Interval . . . 52
5.9 Récapitulatif du formalisme informatique . . . 52
5.10 Graphe de la hiérarchie objet intono-syntaxique . . . 53

6 Exploitations du formalisme **55**
6.1 Dépliage de la transcription balisée . . . 55
6.2 Repliage : enrichissement des annotations micro-syntaxiques du parseur automatique . . . 56
6.3 Projections, transformation de l'arbre . . . 58
6.4 Requêtes . . . 60
6.4.1 Principes généraux . . . 60
6.4.2 Conversion en formats structurés XML pour la base de données RhapSQL . . . 60

7 Implémentation **63**
7.1 Algorithme et script de construction du multi-arbre à partir du balisage . 63
7.2 Dépliage de l'arbre . . . 66
7.3 Repliage . . . 68
7.4 Projections . . . 69
7.5 Correspondance avec le TextGrid . . . 71

8 Conclusion **73**

IIIÉtude Intono-syntaxique du phénomène d'entassement **75**

9 L'entassement **79**
9.1 Étude syntaxique du phénomène d'entassement . . . 79
9.2 L'entassement : une globalisation de différents niveaux linguistiques . . . 80
9.2.1 La répétition et la disfluence . . . 80
9.2.2 La co-composition . . . 80
9.2.3 Les binômes irréversibles . . . 81
9.3 La Typologie des entassements selon Pietrandrea et Kahane . . . 81
9.3.1 Les entassements dit de re . . . 82
Les coordinations compositionnelles . . . 82

Coordinations non relationnelles 83
9.3.2 Les entassement dits de dicto ou les listes dialogiques 84
Les disfluences 84
Les reformulations 85
9.3.3 La négociation 85
La demande de confirmation 85
La confirmation 86
La réfutation 86
La correction 86
9.4 Topologie des entassements 87
9.4.1 Quelques statistiques sur le corpus 87
Statistiques sur la topologie des entassements (continus et discontinus confondus) 87
Statistiques en fonction du nombre de couches des entassements . 87
9.4.2 Les entassements continus 88
Les entassements continus uniques 88
Les entassements continus parents 88
Les entassements continus enfants 88
Les entassements continus enfants et parents 88
9.4.3 Les entassements discontinus 89
Les entassements discontinus uniques 89
Les entassements discontinus parents 89
Les entassements discontinus enfants 89
Les entassements discontinus enfants et parents 89
9.5 Conclusion 90

10 L'entassement marqué prosodiquement ? 91
10.1 Introduction 91
10.2 Études sur différents types de coordinations du français 91
10.3 Étude du phénomène de liste en allemand standard 92
10.4 Étude sur la coordination et la coordination enchâssée en allemand et en hindi 93
10.5 À la recherche d'unité d'entassement prosodiquement marquées 94

11 Méthodologie 97
11.1 Introduction 97
11.2 Recueil des données intono-syntaxiques 98
11.3 Étude des transitions 99
11.3.1 Définition des données 99
11.3.2 Comparaison des données 102
11.4 Étude du patron prosodique des couches 102
11.4.1 Définition des données 102
11.4.2 Analyse des résultats 103

12 Résultats 105
12.1 Étude des transitions entre couches 105
12.1.1 Étude descriptive 105
12.1.2 Étude quantitative 109
12.2 Étude des patrons prosodiques intra-couche 111
12.2.1 Étude des distributions marginales du ralentissement 111
12.2.2 Étude appariée 114
Étude appariée de la f_0 114
Étude appariée du ralentissement 115

13 Conclusion 121

IV Conclusion Générale **123**

A Tests statistiques considérés **127**

A.1 Table du t-test de Student 127
A.2 Test de normalité avec le Test du χ^2 d'adéquation 128
A.3 Table du χ^2 129
A.4 Test de Kolmogorov-Smirnov à deux échantillons 130
A.5 Table de Kolmogorov-Smirnov 131
A.6 L'ANOVA à un facteur 132

B Code source complet des scripts Python **133**

B.1 Classe Word 133
B.2 Classe Token 136
B.3 Classe Lexeme 137
B.4 Classe Dependancy 139
B.5 Classe Node 140
B.6 Construction de l'arbre 154
B.7 Récolte des données statistiques 157
B.8 Script principal et fonctions annexes 160
B.9 Construction des données Matlab depuis Python 164
B.10 Script de manipulation des TextGrids 165

C Code source des scripts Matlab **171**

C.1 Script principal des analyses 171
C.2 Recherche des données dans la matrice 175
C.3 TextGrid Loader 177
C.4 TextGrid Writer 178
C.5 Write TextGrid's transcription to file 179

Index **181**

Bibliographie **183**

Bibliographie **183**

Table des figures

2.4.1 Hiérarchie des arbres de dépendance micro-syntaxiques 22
2.4.2 Hiérarchie micro-syntaxique des entassements 22
2.4.3 Hiérarchie macro-syntaxique des Unités illocutoires et rectionnelles 22

3.2.1 Exemple de ralentissement pour la syllabe /ma/ (D0003 Corpus Rhapsodie . 26
3.3.1 Proéminence par allongement syllabique (1) [79] 27
3.3.2 Proéminence par allongement syllabique (2) [79] 28
3.3.3 Annotation catégorielle des proéminences 28
3.3.4 Exemple d'annotation en proéminences (D2003 Corpus Rhapsodie) 29
3.5.1 Exemple de stylisation et de transcription des contours de F0. (D2001 Corpus Rhapsodie) 30
3.5.2 Tableau récapitulatif de l'encodage des niveaux de hauteur des contours et des registres 30
3.5.3 Tableau récapitulatif de l'encodage des positions saillantes 31
3.5.4 Exemple de transcription des registres et des contours de F0 sur la syllabe et le groupe rythmique (D2001 Corpus Rhapsodie) 31
3.5.5 Tier des valeurs de f_0 32
3.6.1 Hiérarchie prosodique 32

4.0.1 Exemple de texte annoté (D2001 Corpus Rhapsodie) 37

5.2.1 Transformation du texte balisé en arborescence 40
5.2.2 Diagramme UML de classes haut niveau 44
5.3.1 Classe Word 45
5.4.1 Classe Node 47
5.5.1 Classe Token 49
5.6.1 Classe Lexeme 50
5.7.1 Classe Dependency 51
5.8.1 Schéma UML des classes TextGrid, Tier et Interval 52
5.10.1Graphe de la hiérarchie intono-syntaxique 53

6.1.1 Résultat du dépliage d'un entassement 55
6.2.1 Arbre de dépendance visualisé sous Arborator 57
6.3.1 Arbre comportant l'information macro-syntaxique et d'entassement 58
6.3.2 Arbre macro-syntaxique 59
6.3.3 Arbre d'entassement 59
6.4.1 Arbre macro-syntaxique 61
6.4.2 Arbre d'entassement XML 61

7.2.1 Production du dépliage d'après l'algorithme de dépliage donné en 7.3 67

9.3.1 Typologie des entassements selon Pietrandrea et Kahane [74][74] 87
9.4.1 Table de la distribution topologique des entassements 87

9.4.2 Table des valeurs syllabiques en fonction du nombre de couches dans un entassement 87
9.4.3 Topologie des entassements 90

11.2.1 Tables des données prosodiques étudiées 98
11.2.2 Tables des données topologiques étudiées 98
11.2.3 Processus d'extraction des information syntaxiques et prosodiques encodées dans la structure de données 99
11.3.1 Transitions inter-couches 100
11.3.2 Test toutes-syllabes 100
11.3.3 Transitions intra-couches 101

12.1.1 Histogramme bidimensionnel des transitions de f_0 toutes-syllabes. 106
12.1.2 Histogramme bidimensionnel des transitions de f_0 intra-couche. 106
12.1.3 Histogramme bidimensionnel des transitions de f_0 inter-couches. 106
12.1.4 Histogramme bidimensionnel des transitions de ralentissement toutes-syllabes 107
12.1.5 Histogramme bidimensionnel des transitions de ralentissement intra-couche . 107
12.1.6 Histogramme bidimensionnel des transitions de ralentissement inter-couches . 107
12.1.7 Histogramme bidimensionnel des transitions de proéminence toutes-syllabes . 108
12.1.8 Histogramme bidimensionnel des transitions de proéminence intra-couches . . 108
12.1.9 Histogramme bidimensionnel des transitions de proéminence inter-couches . . 109
12.1.10 Résultats des tests d'adéquation à la distribution normale 110
12.1.11 Résultats des tests de Kolmogorov-Smirnov à deux échantillons 110
12.2.1 Diagramme en boîte des distributions marginales de la f_0 en début, milieu et fin de couche 112
12.2.2 Diagramme en boîte des distributions marginales des ralentissements en début, milieu et fin de couche. 112
12.2.3 Histogramme bidimensionnel de l'évolution de la f_0 par rapport au ralentissement début, milieu et fin de couche. 113
12.2.4 Diagramme en boîte des distributions des f_0 en début (1) et milieu (2) de couche 114
12.2.5 Diagramme en boîte des distributions des f_0 en fin (1) et milieu (2) de couche 115
12.2.6 Diagramme en boîte des distributions des ralentissements en début (1) et milieu (2) de couche 116
12.2.7 Diagramme en boîte des distributions des ralentissements en fin (1) et milieu (2) de couche 117
12.2.8 Exemple d'allongement syllabique et d'augmentation de la f_0 en fin de couche 117
12.2.9 Exemple d'augmentation de la f_0 alors que le ralentissement n'augmente pas en fin de couche 118
12.2.10 Contours de f_0 montants en fin de couche de l'entassement "{ deux petites phrases | deux vraies options }" 119
12.2.11 Valeur moyenne de f_0 montante en fin de couche de l'entassement "{ qui travaillent avec moi | ^ou qui ont travaillé avec moi }" 120

A.1.1 Table t-test Student 127
A.3.1 Table du χ^2 129
A.5.1 Table de Kolmogorov-Smirnov 131

Liste des algorithmes

7.1 Algorithme de construction du multi-arbre à partir du balisage (1) 64
7.2 Algorithme de construction du multi-arbre à partir du balisage (2) 65
7.3 Algorithme de dépliage 67
7.4 Algorithme de repliage 68
7.5 Algorithme de regroupement 70

Première partie

Rhapsodie :

"un treebank de français oral spontané"

Chapitre 1

Contexte et objectifs

Le projet ANR Rhapsodie [76] a pour but d'étudier l'intono-syntaxe du français en créant un corpus de français spontané annoté en prosodie et syntaxe et de proposer un système d'annotation riche qui puisse servir de référence pour l'analyse de données orales[9][10][54].

Corpus

Prosodie

1.1 Un défi pour l'analyse syntaxique

Syntaxe

Les syntacticiens de Rhapsodie ont développé un système d'annotation syntaxique en se basant sur les travaux de l'école d'Aix [17] et la syntaxe de dépendance introduite par Tesnière [89].

Intono-syntaxe

L'enjeu du balisage syntaxique élaboré par l'équipe des syntacticiens de Rhapsodie était de pouvoir annoter conjointement les niveaux macro et micro-syntaxiques. La difficulté d'un tel objectif réside dans le fait que ces deux niveaux habituellement étudiés séparément nécessite d'appréhender des phénomènes qui sont la plupart du temps non concomitants.

Macro-syntaxe, Micro-Syntaxe

La problématique qui sous-tend le système d'annotation syntaxique de Rhapsodie est que le français parlé transcrit ne présente pas suffisamment de similitudes par rapport à la syntaxe de l'écrit pour pouvoir être traité directement par des parseurs syntaxiques tels que FRMG [26]. Les transcriptions de l'oral sur lesquelles ont travaillé les syntacticiens ne sont ni ponctuées, ni segmentées et comportent un grand nombre de phénomènes propres à l'oral tel que les disfluences etc. Ce sont des particularités inhérentes à la transcription qui posent un réel problème aux parseurs classiques.

Par conséquent, il a été nécessaire d'introduire une étape intermédiaire de préparation du corpus, le balisage[10], qui permet en particulier de transformer le texte transcrit en une structure arborescente et notamment de le rendre analysable plus facilement par un analyseur automatique (après parsing de ce balisage).

1.2 Une analyse prosodique complexe

Parallèlement une analyse prosodique a été menée. Sur l'empan syllabique les proéminences selon une étude perceptive du corpus ont été codées sur PRAAT[19] par une équipe de locuteurs naïfs, un contrôle a ensuite été effectué par des experts. Toujours sur la syllabe, les valeurs de F0 et de durée a été calculée. Enfin sur l'intégralité des empans prosodiques et syntaxiques une analyse stylisée de la F0 et des registres locaux ont été calculés.

Proéminences Prosodiques

valeurs de F0

Durées syllabiques

Dans un premier temps, je présenterai les notions syntaxiques nécessaires à la description du phénomène d'entassement en partie 2. Ensuite, je présenterai les travaux antérieurs réalisés en syntaxe, en prosodie et en intono-syntaxe se rapprochant de notre étude. Je décrirai ensuite la méthodologie adoptée pour mener à bien cette étude et enfin je présenterai les résultat obtenus.

Chapitre 2

Analyse syntaxique de corpus : Phénomènes micro et macro syntaxiques

2.1 Unités rectionnelles

Approche "bottom-up"

Unité rectionnelle

L'approche adoptée ici est une une approche de bas en haut[54, 10]. Une Unité Rectionnelle (UR) est une unité construite autour d'une tête, qui n'est à priori syntaxiquement dépendante d'aucun élément de rang supérieur dans le texte. La rection est caractérisée par les contraintes imposées sur une position donnée en termes de parties du discours, de marques morphologiques et de possibilités de restructuration (commutation avec un pronom, effacement, passivation, clivage, etc.). Il est important de souligner le fait que les UR ne sont pas définies dans l'absolu. C'est toujours relativement à un texte donné que l'on peut affirmer raisonnablement que certaines constructions ne dépendent d'aucune catégorie du contexte, de plus les UR sont par défaut des composantes illocutoires (CI). Les UR, unités micro-syntaxiques sont souvent considérées comme les unités significatives maximales et sont définies à la fois par leur connexité rectionnelle interne[11] et par leur autonomie externe : *"La micro-syntaxe vise à décrire des constructions syntaxiques conçues comme des ensembles rectionnels complets" [10].*

2.2 Unités Illocutoires

Unité illocutoire

Parallèlement à l'UR, il y a l'Unité Illocutoire (UI) dont la délimitation est liée à la reconnaissance de la force illocutoire[23] qui peut affecter un segment dans un texte. UR et UI sont des unités relativement autonomes qui ont leurs propres règles de formation et leurs propres combinatoires. L'UI fait partie de la macro-syntaxe et *"on appelle unité illocutoire une portion de discours comportant un unique acte illocutoire, soit une assertion, soit une interrogation, soit une injonction"*. [10]

Les syntacticiens de Rhapsodie ont considéré que ces deux modules de l'analyse syntaxique sont complémentaires mais que la sortie de l'un ne constitue pas l'entrée de l'autre. Ainsi les UI sont constituées d'UR variées, allant de l'interjection à des constructions plus complexes à plusieurs enchâssements. Les UI peuvent combiner plusieurs UR, mais leurs frontières ne coïncident pas forcément entre elles [1].

Le principe d'annotation consiste à segmenter dès que l'on ne peut plus effectuer de rattachement micro-syntaxique à l'intérieur du texte.

1. Il arrive qu'une UR se prolonge au delà d'une UI par exemple.

Chaque UI se décompose en un certain nombre d'unités, prosodiquement marquées (c'est du moins l'hypothèse qui est faite : [23][16]), que l'on appelle composantes illocutoires (CI). On appelle noyau (kernel) la CI qui porte la force illocutoire. L'une des principales caractéristiques du noyau est de pouvoir être nié ou interrogé. Les CI qui précèdent et suivent le noyau sont appelées respectivement des pré-noyaux (prekernel) et des post-noyaux(postkernel)[10].

2.3 Phénomènes annotés manuellement et balises associées

2.3.1 Balisage des UI

Les UI sont délimitées par le symbole // qui est une marque de fin d'UI :

je m'appelle Angelina //
j'ai dix-huit ans // (M1003 Corpus Rhapsodie)

Par défaut, le symbole //, qui marque la fin d'une UI, marque aussi la fin d'une UR. Cependant, UR et UI ne se correspondent pas toujours. Il existe des UR qui dépassent la frontière de l'UI. Le symbole //+ (le + indique de manière générale une relation de rection) indique que l'UR se poursuit après la fin de l'UI.

c'est un Chinois //+ très riche // (D2010 Corpus Rhapsodie)

La notion de parallélisme permet de limiter la notion d'entassement à l'entassement d'unités à l'intérieur de la même UR. Cependant, un certain nombre de parallélismes syntaxiques et lexicaux sont reconnus qui assurent une cohésion entre UI. Ils sont annotés par le signe //=.

"oh" tout est relatif //= tout est relatif // (D0009 Corpus Rhapsodie)

2.3.2 Balisage des enchâssements et parenthèses

Une UI peut se trouver à l'intérieur d'une autre UI. On distingue deux cas : les enchâssements et les insertions. Les enchâssement comportent le discours rapporté et les greffes alors que les insertions sont les parenthèses.

Le discours rapporté dans cet exemple, "*casse-toi pauvre con*" forme une UI. Par contre, "il a dit" n'est ni une UI complète, ni une UR complète. On considère donc que "*casse-toi pauvre con*" dans "*il a dit casse-toi pauvre con*" est régi par le verbe dire (il est son objet direct).

$L2 [**ça < ce sont les anglais //**] // (D0003 Corpus Rhapsodie)

La greffe est un cas “un peu limite” d’UI. Il s’agit du procédé qui consiste à remplir une position syntaxique à l’aide d’une autre catégorie que celle attendue [27]. On est donc face à une rupture de sous-catégorisation. En général, cette rupture consiste en une sorte de commentaire périphrastique venant combler ou commenter un choix lexical. Comme dans le discours rapporté, une UI vient occuper une position régie à l’intérieur d’une UR. Ce type d’enchâssement est donc lui aussi annoté par des crochets [ ] :

vous avez dit que [**disons ma carrière pour simplifier // **] témoigne de ma bonne conduite // (D2001 Corpus Rhapsodie)

L’enchâssement ne contient pas toujours une UI, en effet il peut aussi contenir des sous-composantes d’une composante illocutoire (CI). Par exemple, ici il s’agit d’un enchâssement d’une proposition avec un pré-noyau mais qui n’est pas une UI :

ce qui fait que [**au moment de la guerre < nous étions toujours en Bretagne**] // (D0003 Corpus Rhapsodie)

On parle d’insertion d’UI chaque fois qu’une UI vient interrompre momentanément une autre UI. On utilise les parenthèses simples : **( )** pour délimiter l’UI insérée. Ce choix de symbole peut-être discuté dans la mesure où c’est le même symbole qui est utilisé pour annoter les in-noyaux et que le seul marqueur qui permet de différencier ces deux type de phénomènes sont les // qui indiquent le fin d’une UI.

"euh" et sinon < les spécialités { les m~ | un { peu moins **(je sais pas si c’est ça qui vous intéresse //)** | petit peu moins } } prises < "bah" { { c’est les | c’est les } spécialités à risques //+ | { la gynéco obstétrique (par exemple) | la cancérologie } } // (D0006 Corpus Rhapsodie)

2.3.3 Balisage des marqueurs d’UR et de Composante illocutoire

Ci-dessous les balises utilisées pour l’annotation des composantes illocutoires et unités rectionnelles.

Pré-noyau

Le pré-noyau est une unité de type macro-syntaxique qui se trouve obligatoirement à gauche du noyau. Il peut être régi par un élément appartenant au noyau (dans ce cas on dit qu’il est intégré) ou bien non régi. Le pré-noyaux est annoté grâce aux symboles < et <+ pour les pré-noyaux intégrés (soit une composante illocutoire) :

bien évidemment < c’est vrai pour la peinture religieuse en Occident // (M2002 Corpus Rhapsodie)

au début <+ il n' y avait pratiquement pas d' informatique // (D0005 Corpus Rhapsodie)

On note que que le symbole + dénote une relation de rection avec le noyau de l'UI, ce qui fait que <+ n'est pas une frontière d'UR contrairement à <.

Post-noyau

Le post-noyau est une unité de type macro-syntaxique qui se trouve nécessairement à droite du noyau. Il inclut normalement des éléments non régis et des éléments régis. Il se caractérise par le fait qu'il ne porte pas de modalités, qu'il peut être antéposé et que la portée des modalités présentes dans le noyau ne peut pas l'atteindre. Le symboles > et >+ signalent les post-noyaux :

^et "euh" Charlot s'est accusé > **plutôt que de laisser la jeune fille s'accuser** // (M0024 Corpus Rhapsodie)

^mais vous étiez auprès des femmes >+ **là-bas** // (D2004 Corpus Rhapsodie)

In-noyau

L'in-noyau est une unité de type macro-syntaxique qui se trouve nécessairement à l'intérieur du noyau. Il se caractérise par le fait qu'il ne porte pas de modalités, qu'il peut généralement être antéposé ou postposé et que la portée des modalités présentes dans le noyau ne peut pas l'atteindre. Les in-noyaux sont constitués de catégories diverses. Les in-noyaux se différencient des insertions par le fait qu'ils ne peuvent constituer un noyau à eux seuls. On les annote par les symboles suivants, **( )** et **(+)** :

une rallonge à venir **(également)** dans le secteur automobile // (M2006 Corpus Rhapsodie)

le cri de Job **(+ que nous avons entendu dans la première lecture)** retentit à nos oreilles // (M2003 Corpus Rhapsodie)

Introducteurs

Une UI peut commencer par un ou plusieurs introducteurs : ce sont des conjonctions ou des conjonctions de subordination comme alors, et, mais, car, puis... Ces éléments ont les propriétés suivantes : ils ont la fonction de préciser la nature de la relation entre l'UI qu'ils introduisent et d'autres UI dans le discours (notamment l'UI qui précède) ils n'ont pas de relation de dépendance syntaxique avec d'autres éléments de l'UI. Ils sont

prosodiquement intégrés dans leur UI. On les annote par le symbole ^.

$L2 ^et ^puis en se mariant < ils sont partis vivre en province // (D0003 Corpus Rhapsodie)

^et tu arrives à la fontaine place Notre Dame // (M0001 Corpus Rhapsodie)

Sont annotés avec le même symbole les marqueurs d'entassement comme et, ou, mais, etc :

{ les uns | **^et** les autres } (M2003 Corpus Rhapsodie)

2.3.4 Balisage des Entassements

Les entassement font partie de la micro-syntaxe, l'entassement, aussi appelé pile (voir [**?**][50][48]), est un dispositif de connexion syntaxique qui relie tous les éléments qui occupent la même position syntaxique à l'intérieur de l'UR (cf chapitre 9). On utilise les symboles **{** et **}** pour marquer le début et la fin de la liste et **|** pour signaler le ou les points de jonction dans le prolongement des listes paradigmatiques et de l'analyse en grille proposées dans [14]. Les entassements ont des fonctions sémantiques et pragmatiques variées et peuvent aussi contenir des éléments macro-syntaxiques : ils peuvent servir à signaler des disfluences, à établir des relations entre référents, à créer des nouveaux référents, à reformuler, à exemplifier, à préciser , à intensifier. Les conjoints ne sont pas nécessairement des constituants micro-syntaxiques mais peuvent être des disfluences par exemple :

^et **{ la | la }** Loire est en bas // (D0003 Corpus Rhapsodie)

“euh" **{ deux petites phrases | deux vraies options }** qui dessinent **{ votre route //+ | une route qui témoigne { d'une certaine | d'une bonne | d'une très bonne } conduite }** // (D2002 Corpus Rhapsodie)

2.4 Hiérarchies syntaxiques du corpus

Nous avons vus que l'ensemble du corpus a été annoté selon deux niveaux majeurs de la syntaxe : le niveau macro-syntaxique et le niveau micro-syntaxique. Ces deux niveaux d'annotations créés sur l'ensemble des échantillons du corpus deux hiérarchies distinctes dont voici les arborescences : Arborescence Syntaxique

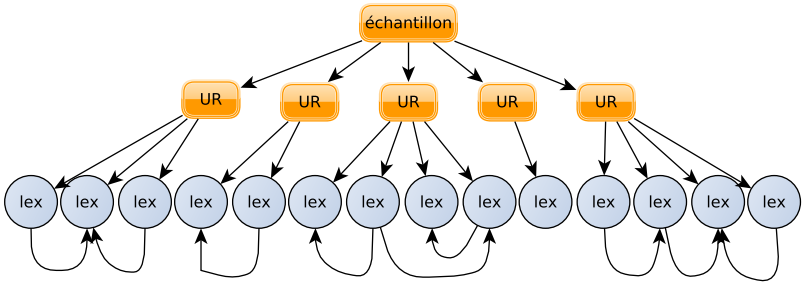


FIGURE 2.4.1 – Hiérarchie des arbres de dépendance micro-syntaxiques

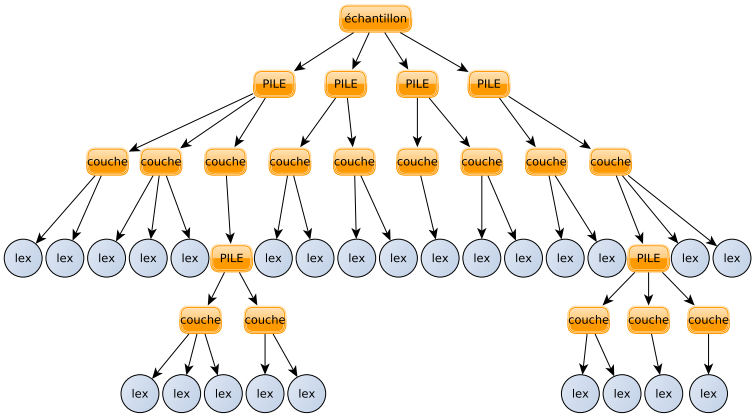


FIGURE 2.4.2 – Hiérarchie micro-syntaxique des entassements

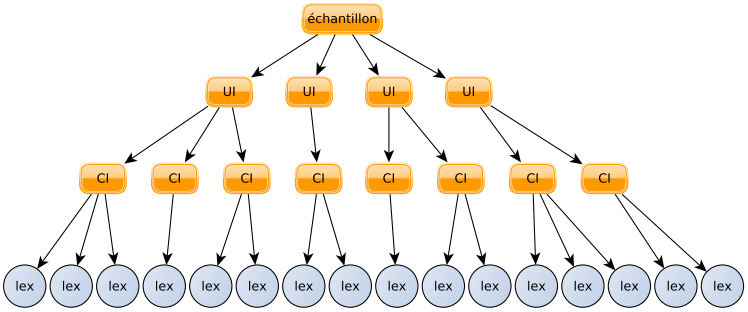


FIGURE 2.4.3 – Hiérarchie macro-syntaxique des Unités illocutoires et rectionnelles

L'un des objectifs du traitement du balisage est donc l'obtention de données structurées. On cherche alors à générer à partir du corpus annoté l'ensemble des arbres topologiques et des arbres d'entassement possibles. Pour ce faire, l'équipe de recherche a opté pour une structure XML, ce format sert de format d'import-export pour le corpus annoté et la base SQL du projet. À terme, tous les résultats des différentes phases d'annotations syntaxiques du corpus sont donc appelés à être chargés dans une base de données SQL. Le tables relationnelles de la base sont enrichies à partir de ces fichiers XML. Le format XML des différentes phase d'annotation sert également d'input au logiciel Vakyartha-Arborator

[37] qui permet à l'équipe des syntacticiens de procéder à une phase de vérification et correction manuelle après les phases de projection, dépliage et repliage des données annotées.

La problématique qui se pose est donc de convertir les structures de données obtenues vers des fichiers structurés XML. La représentation du balisage sous la forme d'arbre permet d'effectuer cette tâche de manière triviale à partir du multi-arbre ainsi créé. Pour l'opération de projection de la micro ou de la macro-syntaxe, des algorithmes récursifs très simples permettent de convertir un arbre en format structuré de type XML.

Balise	Phénomène syntaxique	Type d'unité			Exemple
		UI	UR	Pile	
//	unité illocutoire	X			$L1 vous étiez sténodactylo //
//+	unité illocutoire associée	X			^et ^puis je suis toujours étonnée //+ maintenant < plus //
//=	unité illocutoire entassée	X		X	$L1 vous savez ce que c'est que d'être pauvre //= $L2 je sais ce que c'est que d'être pauvre //
<	pré-noyau		X		**Françoise Giroud <** vous occupez un poste d'observation que des gens haut placés vous envient //
<+	pré-noyau régi		X		**aujourd'hui <+** "euh" je ne crois pas que ce serait possible //
>	post-noyau		X		$L2 j'aime bien comprendre comment ça marche **> les autres** //
>+	post-noyau régi		X		je reviens jamais en métro **>+ sauf quand il y a des manifestations** //=
( )	in-noyau				c'est une expérience **(ça)** que je n'ai jamais oubliée //
(+)	in-noyau régi		X		il faut s'appeler Rachida Dati **(+ écrit aussi Libération Champagne)** pour oser affirmer qu'il ne s'agit pas d'un vote sanction //
[ ]	enchâssement		X		^et je précise d'ailleurs que [**quand Marcel Achard a écrit ce texte <+ il y avait déjà presque dix-sept ans**] //
[//]	greffe	X			ce qui est horrible < c'est de se dire [**je n'en sortirai jamais //]**
(//)	incise	X			et est-ce que c'est avec ces yeux **(si vous voulez dire //)** que je sais regarder //
“”	marqueur discursif		X		**"bon"** je l'ai toujours eu **"je crois"** //
{ }	entassement			X	il n'y a que les hommes **{ qui travaillent avec moi**
|	jonction				**| ^ou qui ont travaillé avec moi }** qui pourraient vous répondre //
|}	interruption d'entassement				"euh" **{ pourquoi |}** j'ai fait du journalisme //+
{|	reprise d'entassement				**{| parce que "euh" ça se passait tout de suite après la guerre }** //
^	introducteur		X		**^et** vous me direz si vous êtes d'accord //
#	ancrage de discontinuité				[il va sans doute faire la même chose qu'avant //+ #]
## //	segement à rattacher à la discontinuité				(+ pronostique Francis Brochet) **## ^mais autrement // //**

TABLE 2.4.1 – Détail des balises utilisées pour annoter manuellement les UR, UI et entassements

Chapitre 3

Analyse Prosodique du corpus

Comme pour l'analyse syntaxique, l'analyse prosodique a été réalisée sur plusieurs niveaux.

3.1 Méthodologie

La perspective phénoménologique (perception...) adoptée dans Rhapsodie est relativement indépendante des théories en phonologie prosodique, sans pour autant bloquer ces dernières. L'annotation fournit les primitives prosodiques a priori nécessaires et suffisantes pour permettre une analyse qualitative et quantitative exhaustive des corrélats prosodiques associés aux structures et processus linguistiques analysés. Le système d'annotation et segmentation mis au point sont reproductibles simplement et de fait réalisables par un automate. Le codage des données est mutualisable, reproductible (par une double annotation de naïfs puis d'experts). Les principes d'économie et de simplicité ont été privilégiés.

Annotations :
– codeurs naïfs (+ automates),
– vérification expert(s)
– codage référence
Données acoustiques quantifiées :
– outils automatiques (Analor[3], Prosogram[65], Winpitch[60], etc)

3.2 Mesures de durées (ralentissements)

Les mesures de durées ont été effectuées par Cédric Gendrot [36]. Dans un premier temps ont été effectuées des mesures de durée de phonèmes en contexte sur 30 heures de corpus, ce qui constitue les durées de référence. Ces mesures ont été réalisées sur des di-phones plutôt que des tri-phones parce que les occurrences de tri-phones étaient trop rares dans le corpus Rhapsodie (3 heures).

À partir des valeurs de référence, on peut établir l'allongement, quand la mesure dépasse 2 par exemple, cela veut dire que la syllabe en question est deux fois plus longue que la durée de référence des phonèmes dans le même contexte segmental. Par exemple, pour "mélomane", la syllabe "ma" a un indice de $2,044$, elle est environ 2 fois plus longue que la référence AB (cf figure 3.2.1).

Pour la syllabe /ma/, on a calculé les durées de référence en additionnant les durées moyennes du phonème /m/ précédé de /o/ (mélomane) et du phonème /a/ suivit de /n/ calculées d'après le corpus de référence et en les divisant par 2 ce qui nous donne A et de

la même manière on a calculé les durées moyennes des phonème /a/ précédé de /m/ et /a/ suivit de rien (on cherche la syllabe /ma/) dans le corpus de référence et on divise par 2 ce qui nous donne B. AB nous fournit ainsi la durée de référence pour la syllabe /ma/ et le calcul de l'allongement syllabique s'obtient en divisant le durée de la syllabe /ma/ étudiée par le durée de référence AB.

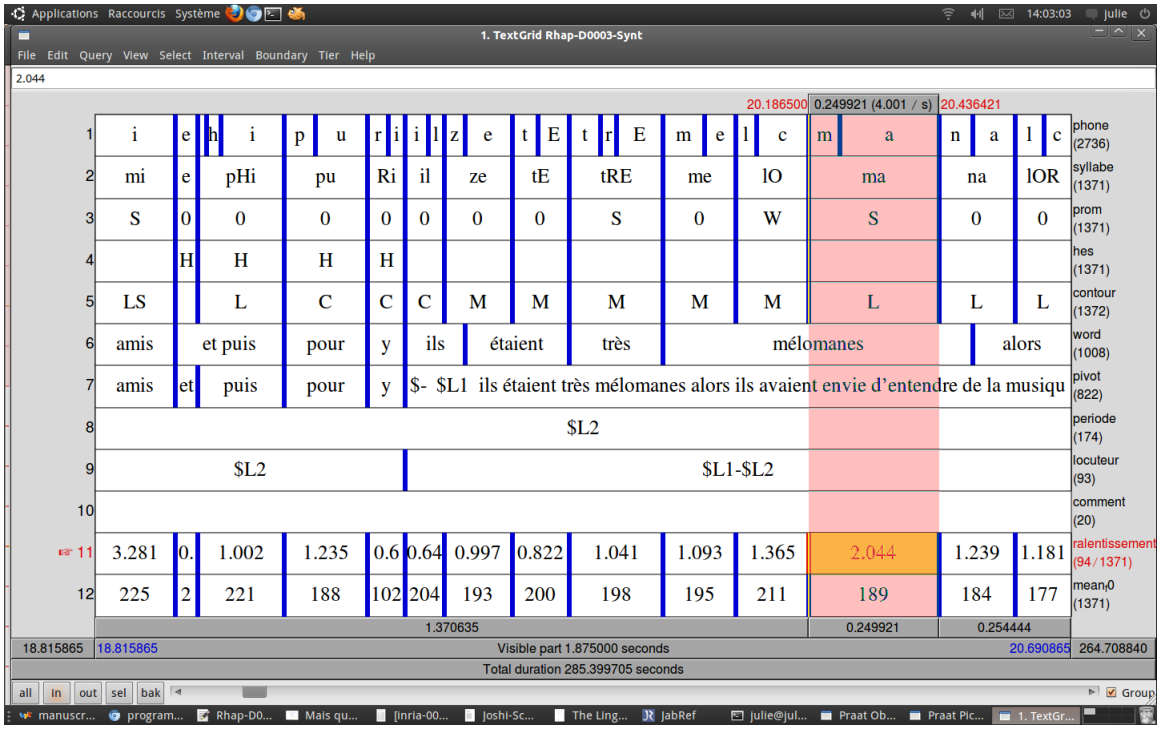


FIGURE 3.2.1 – Exemple de ralentissement pour la syllabe /ma/ (D0003 Corpus Rhapsodie

$$A = \frac{d_{moy}/\text{m}/\text{précédée par/o/}}{2}^{+d_{moy}/\text{a}/\text{suivi par/n/}}, \text{ avec } d_{moy} : \text{durée syllabe}$$

$$B = \frac{d_{moy}/a/\text{précédée par/m/}}{2}^{+d_{moy}/\text{a}/\text{suivi par/./}}$$

$$\text{durée référence} = A + B$$

$$\text{mesure} = \frac{d/\text{ma}/}{\text{durée référence}}, \text{ avec } d_{/\text{ma}/} : \text{durée de le syllabe /ma/ en question}$$

Ces résultats sont importants dans la mesure où ils sont complémentaires d'une mesure de f_0. En effet lors d'une étude prosodique il est primordial d'observer également la durée qui peut être significative de frontière. On peut signaler une frontière par un allongement et/ou un contour spécifique de f_0. Cependant les deux ne sont pas toujours combinées en particulier pour de la parole spontanée.

3.3 Proéminences

La proéminence est une étiquette attribuée à la syllabe qui ne présage a priori en rien du statut fonctionnel de l'élément saillant rencontré. Chacun pourrait potentiellement associer le marqueur à ses objets de description en fonction de sa théorie (accent intonatif, accent focal, délimiteur d'intonation phrase, syntagmes intermédiaires, etc).

Dans le cadre de Rhapsodie, les unités définies subsument la distinction entre accent intonatif (AI), accent focal (AF), et tout ce qui s'y rattache (final de groupe, focal, expressif, etc.). La notion est proche de ce que l'on pourrait appeler le pied rythmique et ne comporte pas d'ancrage sur la notion de mot graphique (donc évite les problèmes des clitiques et des mots dont le statut hésite entre mots lexicaux et mots grammaticaux).

Selon Sauvage et Lacheret [79] le principe de proéminence est définit comme le détachement par contraste du flux verbal d'une syllabe par rapport à ce qui précède. On dénote dès lors un détachement perceptif et une mise en abîme de cette syllabe qui n'a alors n'a pas le même poids que son entourage. Plusieurs raisons (phonétiques et linguistiques) sont à l'origine d'un tel détachement et par conséquent, la position de la proéminence peut s'avérer très variable. Dans le cadre du projet les proéminences ont été annotées dès qu'un contraste sur une syllabe donnée a été perçu.

Il est important de souligner que la notion de proéminence n'est pas seulement relative à la hauteur. D'autres paramètres entrent en ligne de compte (cf figure 3.3.1 et figure 3.3.2). On annote notamment comme tel : un ton perçu dans l'infra-bas suivi d'une pause, un allongement syllabique et aussi les variations de qualité vocale (voix craquée ou "creaky-voice").

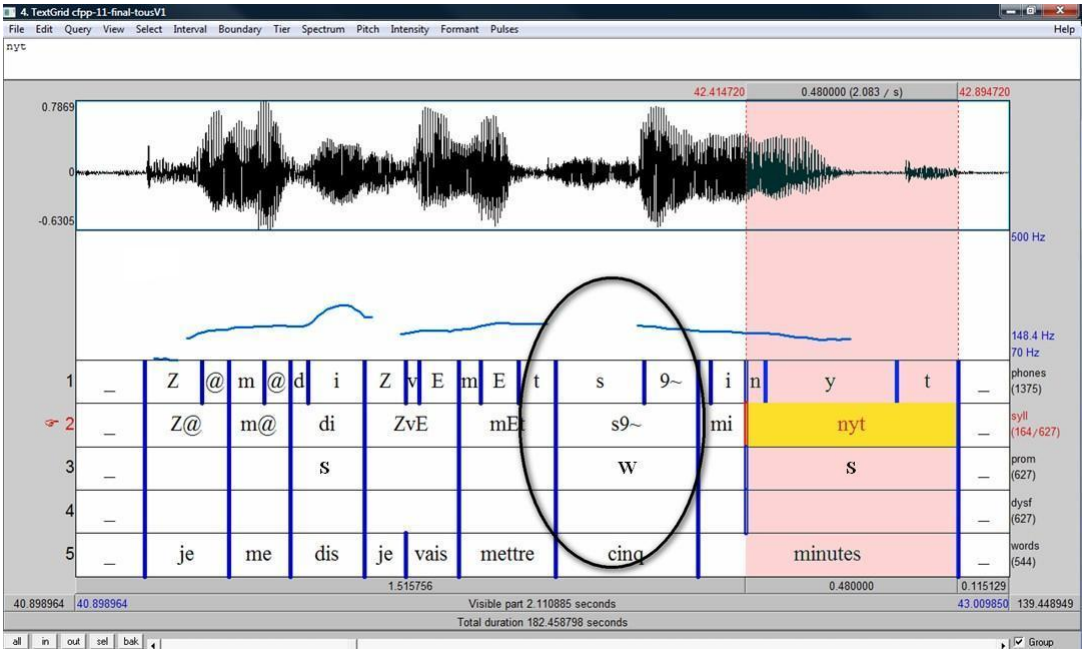


Figure 3.3.1 – Proéminence par allongement syllabique (1) [79]

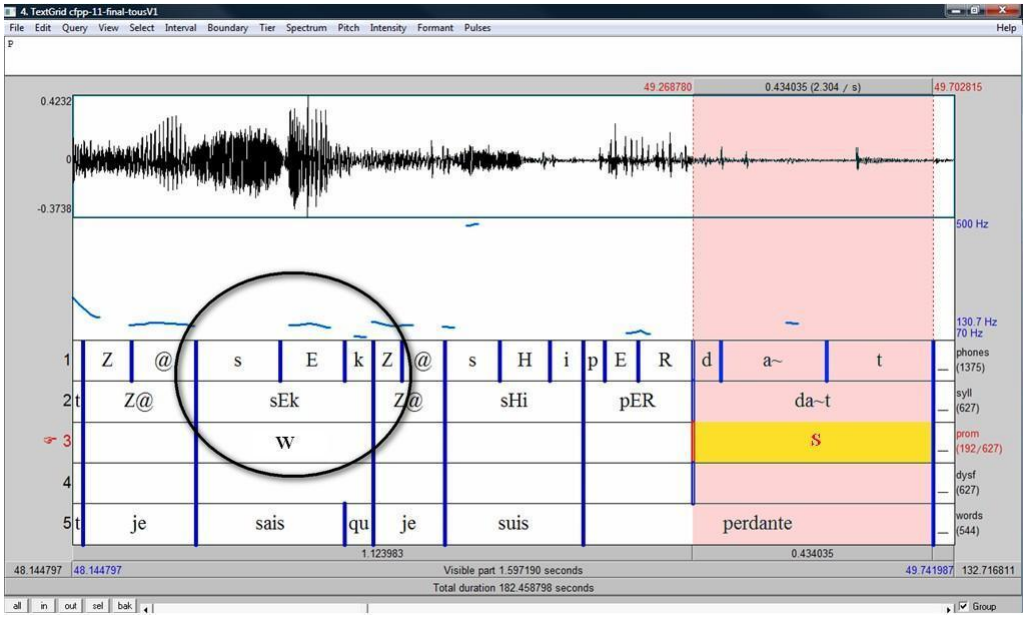


FIGURE 3.3.2 – Proéminence par allongement syllabique (2) [79]

1

La tâche d'annotation des proéminences se base sur des appréciations perceptives ainsi elle est continue[4]. En revanche son annotation est catégorielle. On ne peut donc qualifier une syllabe de strictement proéminente ou strictement non-proéminente, il existe ainsi différents degrés de proéminence allant du non proéminent au très proéminent.

Le continuum des proéminences annotées dans le cadre du projet est détaillé dans la table de la figure 3.3.3 :

valeur de proéminence	**marqueur**
non proéminente (nul)	0
proéminente (weak)	W
très proéminente (strong)	S

FIGURE 3.3.3 – Annotation catégorielle des proéminences

1. Les figures 3.3.1 et 3.3.2 ont été réalisées par J. SAUVAGE-VINCENT[79].

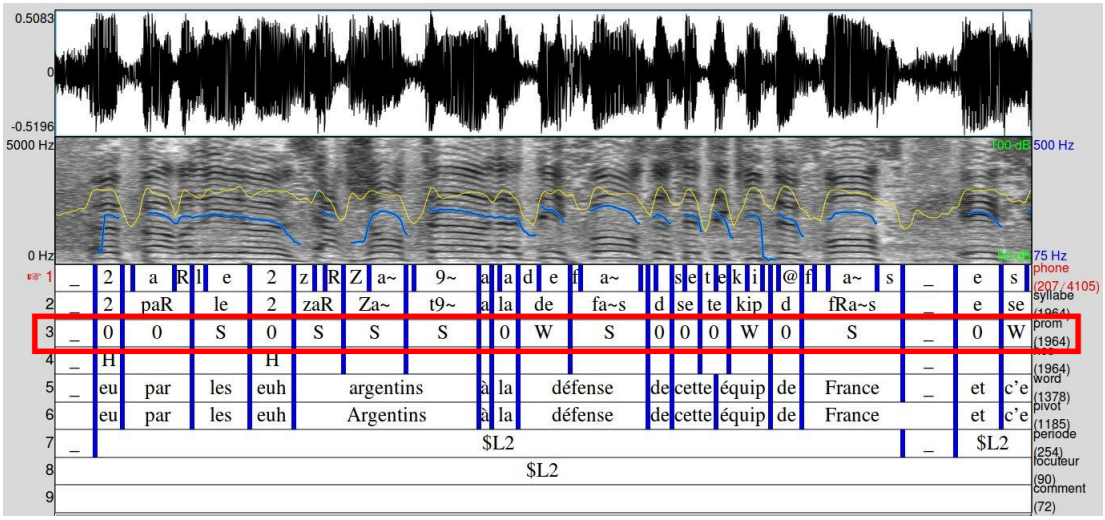


FIGURE 3.3.4 – Exemple d'annotation en proéminences (D2003 Corpus Rhapsodie)

3.4 Disfluences prosodique et hésitations

On définira la disfluence prosodique ou hésitation comme un élément qui brise le déroulement syntagmatique dans la chaîne parlée, une sorte de trébuchement vocal. Il peut prendre différentes natures et correspond souvent à un allongement syllabique excessif associé au travail de formulation en cours. Il peut s'agir aussi d'une répétition de morphème et de bien d'autres choses encore. Toute syllabe perçue comme disfluente sera repérée par une balise spécifique, notée « H ».

3.5 Stylisation des Contours Intonatifs, Mesures de registres locaux, Transcription Tonale et Calcul des valeurs moyennes de F0

3.5.1 Introduction

L'outil de stylisation des contours intonatifs et de transcription tonale développés dans le cadre du projet Rhapsodie par NICOLAS OBIN [69][8][70] décrit les contours prosodiques en prenant le locuteur principal comme référence, et en excluant les segments de tour de parole des autres locuteurs s'il y en a.

L'outil mis au point se décompose en 3 parties :

- La stylisation des contours de f_0 à proprement parlé qui opère par décomposition sur une base de fonctions lentement variable en temps. Cela dans le but de filtrer les variations micro-prosodiques mais également les éventuelles erreurs d'estimation de la f_0
- La détermination des caractéristiques acoustiques de f_0 sur un segment linguistique arbitraire (ex : syllabe, mot, pile)
- La transcription des contours de f_0 sur un alphabet tonal

3.5.2 Registre et Contour

Le registre illustre le registre local du locuteur par rapport à sa tessiture. Les contours sont déterminés sur les points de f_0 situés respectivement au niveau initial, aux niveau et position des saillances intermédiaires (si elles existes) et enfin au niveau final. Le niveau est également normalisé en fonction du registre moyen de chaque locuteur.

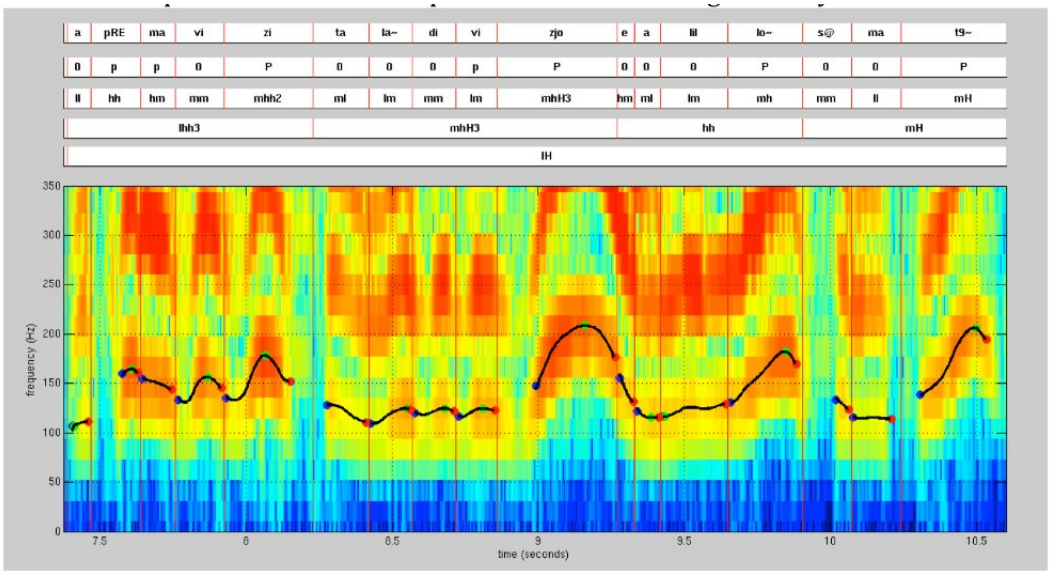


FIGURE 3.5.1 – Exemple de stylisation et de transcription des contours de F0. (D2001 Corpus Rhapsodie)

La courbe noire illustre les variations de F0 stylisé surimposé sur un spectrogramme. Les points de couleurs indiquent les valeurs de F0 de début (noir) /fin (rouge) ainsi que les saillances de F0 (vert) sur les syllabes. En haut : segmentation syllabique, annotation des proéminences prosodiques, et transcription tonale.

3.5.3 Transcription

Lors de la transcription des contours, les niveaux et les positions prennent les valeurs suivantes :

Niveau	
L	niveau bas extrême
l	niveau bas m niveau médium
h	niveau haut
H	niveau haut extrême

FIGURE 3.5.2 – Tableau récapitulatif de l'encodage des niveaux de hauteur des contours et des registres

Les valeurs de niveaux sont déterminées à partir d'une discrétisation du niveau de hauteur est 5 intervalles égaux en échelle logarithmique, et sont utilisés pour déterminer les contours et les registres.

La position des saillances permet de décrire la structure dynamique des contours, en spécifiant la position des saillances de F0 à l'intérieur d'une unité donnée (e.g., syllabe, mot, etc...)

Position	
1	Premier tiers de l'unité considérée
2	Deuxième tiers de l'unité considérée
3	Dernier tiers de l'unité considérée

FIGURE 3.5.3 – Tableau récapitulatif de l'encodage des positions saillantes

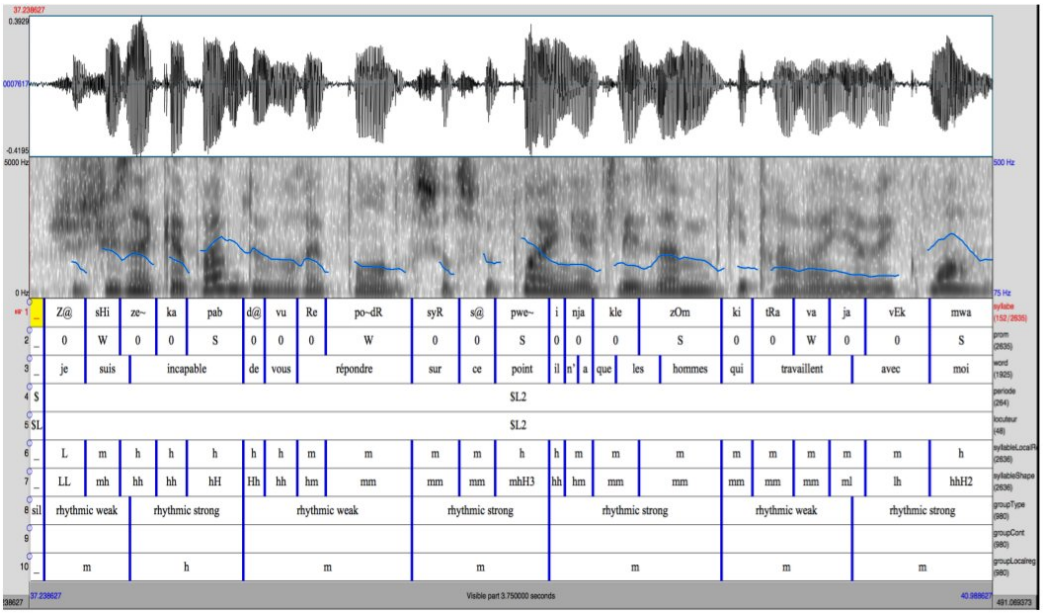


FIGURE 3.5.4 – Exemple de transcription des registres et des contours de F0 sur la syllabe et le groupe rythmique (D2001 Corpus Rhapsodie)

Un contour "mm" indique un plateau dans le registre médium du locuteur. Un contour "mH2h" indique un contour initié dans le niveau bas ("l"), terminé dans le registre haut ("h"), et avec une saillance dans le registre haut-extrême ("H") réalisé dans le cœur de l'unité ("2")

2

3.5.4 Calcule des valeurs moyennes de f_0 des syllabes

Un calcul de la valeur moyenne de f_0 a été effectué par CÉDRIC GENDROT pour chaque syllabe du corpus. Le calcul est simple et consiste à faire la moyenne des valeurs de f_0 de chaque syllabe.

Les valeurs ainsi trouvées sont ensuite stockées dans le TextGrid dans une tier "meanf_0" :

2. Les figures 3.5.1 et 3.5.4 ont été réalisées par N. OBIN.

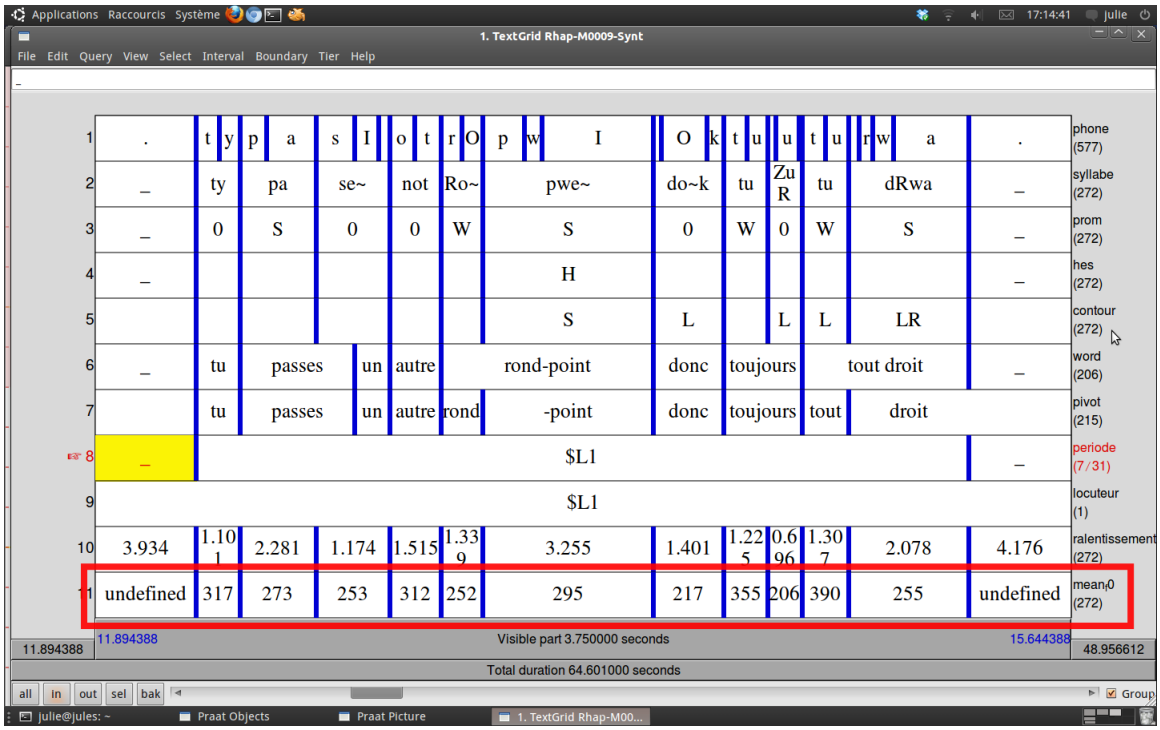


FIGURE 3.5.5 – Tier des valeurs de f_0

3.6 Hiérarchie prosodique du corpus

Hiérarchie prosodique

Chaque échantillon du corpus a été segmentés selon la hiérarchie suivante :

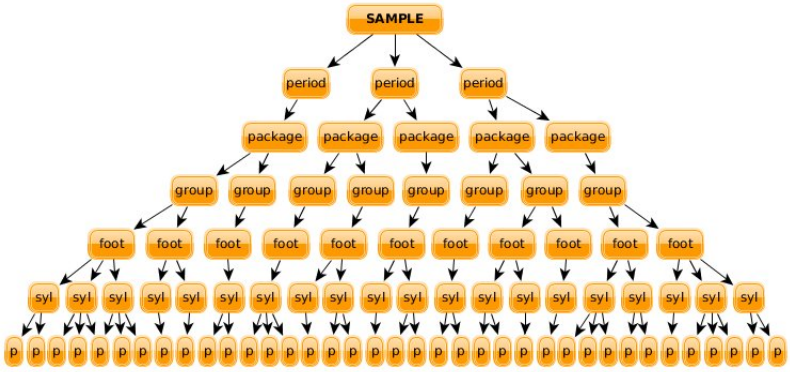


FIGURE 3.6.1 – Hiérarchie prosodique

Deuxième partie

Structures de données hiérarchiques pour l'analyse du Français spontané

Résumé

Sur l'ensemble du corpus présenté précédemment, ont été effectuées des annotations syntaxiques et prosodiques qui reposent chacune sur une structure hiérarchique différentes. De manière à pouvoir les intégrer et à effectuer des études intonosyntaxiques, l'introduction d'un formalisme spécifique est nécessaire.

Dans cette partie,je présente mes efforts en vue de produire une structure hiérarchique objet enrichie, appelée multi-arbre, de l'ensemble des données syntaxiques et prosodiques du corpus. Chaque échantillon est modélisé comme une interaction d'objets de différents types. Sa transcription tout d'abord est considérée comme une suite d'objets "*mots*". L'annotation syntaxique consiste à associer ces "*mots*" à des unités syntaxiques. Un nœud est caractérisée par son type, comme "*noyau*", "*unité illocutoire*", etc. et entretient avec les autres nœuds des relations hiérarchiques. Par exemple, un nœud de type "*noyau*" pourra avoir pour père un nœud de type "*unité illocutoire*".

L'annotation prosodique se concentre sur un modèle hiérarchique où un échantillon, objet de type *"TextGrid"*, est constitué de plusieurs objets de type *"Tier"*. Chaque tier contient contient différents *"Intervals",* qui contiennent des informations d'un type prosodique particulier, comme "*le ralentissement*", "*les proéminences*", etc.

Je propose de réunir ces hiérarchies en associant à chaque mot de la transcription ses différents objets Intervals. De cette manière, on peut déterminer la valeurs d'intervals trouvés dans des nœuds d'un type syntaxique particulier.

Pour construire l'arborescence de ces objets, il faut analyser le langage de balisage utilisé pour l'annotation syntaxique. Ce travail a donné lieu à la création d'un analyseur ad-hoc, capable de générer un multi-arbre à partir de la transcription annotée. Muni de cette structure de données, on peut procéder à une analyse syntaxique automatique, vue comme produisant des objets "*tokens*" et *"lexèmes"* ainsi que des liens de dépendances syntaxiques entre eux. Ces objets ont également été réunis aux précédents, de manière à offrir une architecture faisant interagir les TextGrids, leur Tiers et Intervals, avec les unités syntaxiques et les Token et Lexemes issus de l'analyse automatique.

Chapitre 4

Introduction

Dans ce chapitre, je m'intéresserai au problème de l'analyse d'un corpus annoté de l'oral. Le système d'annotation considéré, présenté en partie I, est celui introduit par l'équipe des syntacticiens du projet Rhapsodie. La principale problématique qui sous-tend un tel projet est que la base écrite sur laquelle on travaille est en réalité une transcription de l'oral, balisée par les annotateurs de manière à délimiter un ensemble de structures arborescentes.

transcription

Ma contribution, plutôt pragmatique que théorique sur ce point, a consisté à aller dans le sens d'un système d'annotation analysable automatiquement et ainsi à fixer certaines limitations au formalisme pour le rendre exploitable en pratique.

Ce balisage se présente sous la forme de symboles clés ajoutés au texte de la transcription, comme en figure 4.0.1.

$L1 ce qui vous a toujours intéressée < c'était chez les gens { le mécanisme de la carrière |} //+

$L2 {| le mécanisme { tout simplement | pas spécialement de la carrière } //+

$L1 | ^et la personnalité aussi } //

$L2 j'aime bien comprendre comment ça marche > les autres // ça < c'est "euh" & // "bon" je l'ai toujours eu "je crois" // par conséquent < j'ai eu envie de savoir un petit peu mieux comment ça marchait // ^et ^puis je suis toujours étonnée //+ maintenant < plus // ^et je précise d'ailleurs que [quand Marcel Achard a écrit ce texte <+ il y avait déjà presque dix-sept ans] // { ^et | ^mais } ensuite <+ j'ai toujours été étonnée de voir que les gens ne voyaient pas des choses qui me paraissaient évidentes >+ dans le comportement des autres "je veux dire" // j'ai toujours l'impression { que { c'est | c'est } énorme | que { c'est là en noir sur blanc | { ^et | ^et } les autres ne voient pas } } // ça m'étonne //

$L1 est-ce que vous avez eu { un desc~ | un destin } exceptionnel // ^ou est-ce simplement la réussite qui devait venir // ^ou s'est-il passé tout à fait autre chose dans votre vie // $L2 "oh" "écoutez" j'ai eu un destin // "euh" exceptionnel me paraît un grand mot // disons que j'ai eu un destin un peu différent "euh" parce que les conditions objectives de ma vie ont été en effet différentes "euh" //

FIGURE 4.0.1 – Exemple de texte annoté (D2001 Corpus Rhapsodie)

Cette annotation introduit plusieurs structures, en particulier macro et micro-syntaxiques. qui sont parfois, comme on le verra, étroitement imbriquées. Plusieurs exploitations de ces données étaient senties dans le cadre du projet Rhapsodie.

objectifs : dépliage et repliage

Tout d'abord, l'annotation devait permettre l'extraction des arbres macro-syntaxiques et des arbres d'entassements, tels que décrits en partie I. Ensuite, il devait être possible de produire grâce à l'annotation une version alternative de la transcription (voir section 2.4) qui en rende possible l'analyse syntaxique automatique. Enfin, il fallait pouvoir répercuter les résultats de cette analyse automatique d'un texte transformée sur le texte original, en utilisant dans ce but la transcription annotée.

Représentation informatique

En l'état, la transcription annotée ne permet pas de réaliser de tels objectifs : il est tout au plus exploitable par un humain.

L'objectif de mon travail a été de construire une structure informatique arborescente à partir du texte annoté, grâce à laquelle tous les traitements voulus pourraient être effectués. Ma tâche a donc consisté à tout d'abord définir une structure de données capable de représenter des phénomènes d'une telle complexité, mais aussi de créer l'outil capable de convertir un texte balisé dans cette structure. La grammaire de balisage développée comportant un grand nombre constituants discontinus, il a été nécessaire de mettre au point un parseur *ad-hoc* permettant l'analyse de l'intégralité des symboles du balisage. En effet le balisage Rhapsodie encode simultanément plusieurs structures d'arbres, ainsi certains symboles servent à marquer une des frontières de constituants pour différents types de structure. On aura par exemple, $<$ (marqueur de pré-noyau) qui marque à la fois une frontière d'UR et une frontière de composante illocutoire (CI). Le balisage mis au point encode également des discontinuités dans la transcription et dans certains constituants (les entassements par exemple).

Cette section est structurée comme suit. Tout d'abord, je présente au chapitre 5 le formalisme adopté pour la représentation informatique des phénomènes micro et macro syntaxiques. Cette présentation prend la forme d'une analyse orientée objet du problème et conduit à l'établissement de plusieurs *classes* d'objets interagissant entre elles au sein d'une même structure hiérarchique de données appelée *multi-arbre*. Pour sa fabrication le multi-arbre utilise l'intégralité des balises du niveau manuel d'annotation micro et macro-syntaxique. On obtient ainsi une hiérarchie de plusieurs niveaux syntaxiques encodée dans un même arbre. Ensuite, je montre au chapitre 6 comment cette structure peut être exploitée pour réaliser les différents traitements complexes demandés. Enfin, je présente les algorithmes utilisés pour implémenter ces traitements au chapitre 7.

Chapitre 5

Formalisme adopté

5.1 Une approche objet

motivations

Dans le formalisme adopté présenté en partie I, les niveaux micro et macro syntaxiques considérés font intervenir des concepts qui interagissent les uns avec les autres. Ainsi, les entassements contiendront plusieurs couches, constituées de différentes unités rectionnelles telles que les noyaux, pré-noyaux, etc. A ces entités s'attachent de plus certaines propriétés telles que le locuteur qui les a énoncées ou leur position dans la transcription. De plus, l'analyse syntaxique automatique des données doit produire des attributs tels que catégorie, fonction, genre, nombre etc, ainsi que des dépendances fonctionnelles entre les différents constituants d'une UI.

Dans cette étude, j'ai considéré d'une manière générale que le texte annoté pouvait être fidèlement représenté comme des *objets* qui ont des liens les uns avec les autres. Ainsi, un entassement ayant deux enfants sera représenté comme un *nœud* de type entassement, auquel sont attachés deux *nœuds* de type couche.

objet

En informatique, un objet est une structure de données qui incorpore à la fois des *attributs* et des *méthodes* portant sur un concept manipulé dans le programme.

Ainsi, un objet VOITURE DE JULIE disposera non seulement de ses propriétés COULEUR et VITESSE ACTUELLE, mais également des méthodes ACCÉLÈRE() et FREINE(). Il peut également être en interaction avec d'autres objets, tel qu'un objet MOTEUR qui lui permet d'avancer. La force du formalisme objet est de permettre une bonne séparation des différents problèmes considérés. Dans notre exemple, la voiture n'a pas à connaître le fonctionnement précis d'un moteur, elle a juste à appeler ses méthodes d'accélération ou de changement de vitesse. C'est dans le moteur que sont menées les différentes opérations internes permettant d'accomplir la tache demandée. On parle d'encapsulation. Un autre moteur, reposant sur un principe totalement différent pourra toujours être utilisé par la VOITURE DE JULIE, pour peu qu'il présente les mêmes méthodes.

classe

Il est clair que de nombreux autres objets comme la VOITURE DE JULIE sont du même modèle. Il existe un *patron* selon lequel ils ont tous été générés à l'identique. Un tel patron porte le nom de *classe*. Il permet de préciser les attributs et les méthodes de tous les objets qui en dérivent. C'est un concept central en programmation orientée objet. Une *classe* définit les attributs et les méthodes de tous les objets qui en sont des *instances* . C'est un canevas sur lequel les différents objets sont bâtis. Les attributs sont des variables (comme voiture.vitesseActuelle) et les méthodes sont des fonctions (comme voiture.accelere()) [22].

Dans toute la suite, j'adopterai le formalisme UML (Unified Modeling Language, [83]) qui permet de représenter de manière graphique une structure objet.

5.2 Les différentes classes considérées

mots

Tout d'abord, le texte annoté présente un certain nombre de mots. Un mot est défini comme un ensemble de caractères, séparés des autres par un espacement. L'ensemble des mots considérés constitue le texte annoté dans son ensemble. Un mot est soit une suite de graphèmes (soit les mots de la transcription), soit un symbole du balisage. S'il fait partie du balisage, il peut être de plusieurs types, autant qu'il y a de symboles définis dans le système utilisé (cf Table 2.4.1). Je présente la classe Word correspondant aux mots en section 5.3.

Word

unités syntaxiques

Au delà des seuls mots du texte annoté, le formalisme définit surtout un ensemble d'unités syntaxiques. Ces unités sont les UI, les entassements, les couches des entassements, les noyaux, les pré-noyaux, etc. Ces unités syntaxiques ont toutes en commun d'être liées à d'autres unités par un rapport de filiation. Par exemple, un entassement aura autant d'enfants que de couches et chacune de ces couches pourra avoir comme enfants d'autres unités telles qu'un pré-noyau, un noyau, ou même d'autres entassements. Ce mécanisme est décrit plus en détail au chapitre 6. C'est ainsi que j'ai défini une classe Node pour représenter n'importe laquelle de ces unités syntaxiques (ou constituants). Le formalisme peut en effet être compris comme établissant une structure arborée dont les unités syntaxiques sont les nœuds et dont les graphèmes sont les feuilles. Un objet de cette classe Node correspondra ainsi à une unité syntaxique particulière présente dans l'annotation. Dans la figure 5.2.1, je montre comment un texte annoté se traduit en pratique comme une arborescence de nœuds et de mots.

Node

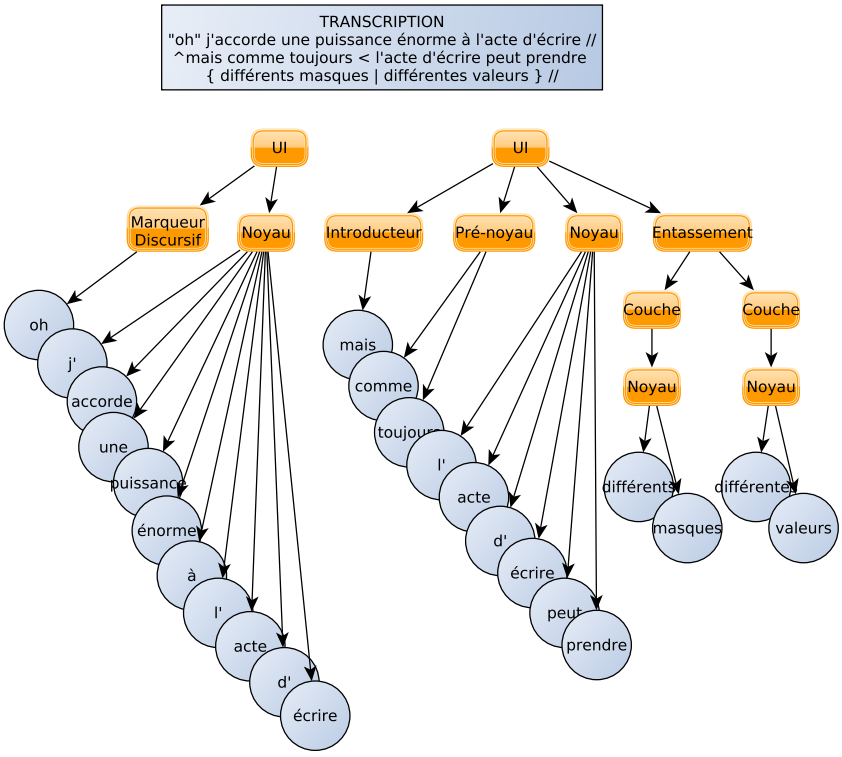


FIGURE 5.2.1 – Transformation du texte balisé en arborescence

La seule considération des mots et des unités syntaxiques pourrait suffire à représenter fidèlement le texte annoté. Cependant, ce texte annoté n'est pas seul : il est la transcription

d'un enregistrement audio. Chaque mot prononcé correspond ainsi à une portion précise du signal audio considéré. Des outils scientifiques tels que PRAAT[19] et EASYALIGN[40] ont été utilisés dans le cadre du projet Rhapsodie pour découper le signal audio initial en syllabes et effectuer sur ce signal une analyse prosodique complexe. C'est ainsi que j'ai considéré un objet TextGrid, qui permet de représenter l'ensemble des analyses prosodiques faites sur un extrait donné. Comme dans PRAAT , un objet TextGrid se décompose en plusieurs niveaux d'analyse, appelés Tier. Chaque Tier correspond à la division de l'ensemble du signal en intervalles. Par exemple, les différentes analyses effectuées dans le cadre du projet Rhapsodie et présentées en détail en partie III sont un découpage en syllabe, une analyse de f_0 par syllabe, une annotation des proéminences, etc. Chaque intervalle d'une Tier est caractérisé par une position dans le signal et un attribut particulier. Il sera représenté dans mon formalisme comme un objet de la classe Interval. Ainsi, un objet Interval sera relié à un objet Tier qui sera lui même relié à un objet TextGrid.

TextGrid

Tier

Interval

Chaque objet de la classe Word qui n'est pas un élément de balisage correspondra donc à un segment du signal. Formellement, il sera relié à un ensemble d'intervalles d'une Tier de référence comme la Tier des syllabes. L'opération qui consiste à établir la correspondance entre les mots et les intervalles est présentée plus en détail dans chapitre 11.

Si on considère à présent les classes Node, Word, Interval, Tier et TextGrid, on peut fidèlement représenter toutes les annotations syntaxiques et prosodiques faites dans le cadre du projet Rhapsodie. Cependant, on n'est pas encore capable de représenter correctement les résultats d'une analyse syntaxique automatique. En effet, un analyseur syntaxique automatique de type FRMG [26] ne considère pas le mot (un ensemble de caractères séparés des autres par des espacements) comme son unité de référence. Au contraire, le texte qu'il analyse est découpé en unités lexicales (Tokens pour l'analyseur automatique FRMG).

intégration d'une analyse syntaxique

FRMG

FRMG est une grammaire TAG/TIG à large couverture du français dérivée d'une description sous forme de méta-grammaire très compacte (< 200 arbres) grâce aux opérateurs de factorisation de DYALOG[25]. La grammaire est couplée avec le lexique LEFFF [78] et le pré-processing morpho-syntaxique SXPIPE. Historiquement les grammaires d'arbres adjoints est un formalisme d'analyse grammaticale introduit par A.K. JOSHI et ses collègues en 1975[47]. Ce formalisme a été utilisé à des fins diverses et particulièrement en linguistique formelle et en informatique pour le traitement de la syntaxe des langues naturelles. Il a d'abord permis de représenter de manière directe des dépendances à longue distance[52] et il permet également de représenter les dépendances croisées, phénomène qui ne peut se traiter avec une grammaire de réécriture hors-contexte, comme S. SHIEBER [82] l'a démontré. Enfin il permet de représenter aisément des grammaires reconnues comme fortement lexicalisées.

Token

TAG/TIG

DYALOG est un environnement pour la construction de l'unification des tableaux basés sur des analyseurs et des programmes. Des traces de calculs sont rassemblés dans le but de partager des sous-calculs communs et de détecter (la plupart des types) des boucles. En gardant des traces des calculs, il est également possible, après calcul, d'extraire, comme une forêt partagée, l'ensemble des épreuves réussies ou des arbres syntaxiques.

Ces caractéristiques sont particulièrement utiles pour gérer les grammaires très récursives et ambiguës pour le langage naturel[25].

Après la tokenisation du parseur FRMG, on constate que le lien entre les mots est les tokens n'est pas bijectif. Si la définition du mot ne repose que sur la présence d'espacements entre eux, celle du Token est spécifique à l'analyseur syntaxique. Ainsi, un mot peut correspondre à plusieurs Tokens, comme dans le cas du mot "lui-même" qui correspond aux tokens "lui" et "même", mais plusieurs mots peuvent aussi être regroupés en un seul token, comme dans le cas des deux mots "vingt deux". La classe Word présentera ainsi

des liens avec la classe Token. Plus précisément, un objet instance [1] de la classe Word pourra être lié à un ou plusieurs Token et chaque Token pourra être lié à un ou plusieurs mots.

Lexeme

Cependant, l'objet de plus forte granularité considéré par les syntacticiens du projet n'est pas le Token, mais le Lexème. Il correspond à l'élément atomique considéré par l'analyseur. Ainsi, lors de l'analyse syntaxique, chaque objet Token peut faire l'objet d'une fission en plusieurs objets lexèmes. Par exemple, le token "au" peut donner lieu à la création des deux lexèmes "à" et "le". Il est remarquable que les lexèmes ne correspondent pas nécessairement aux mots tels qu'ils sont présents dans l'annotation. Par contre, un objet Lexeme ne peut avoir qu'un et un seul objet Token référent. C'est aux objets Lexeme que sont attribuées les différents traits syntaxiques produits par l'analyse automatique, tels que la catégorie, le genre, le nombre, etc.

Dependency

Enfin, l'analyse syntaxique automatique conduit à l'établissement de relations de dépendance entre plusieurs lexèmes et à leur étiquetage en fonction de leur partie du discours. Les dépendances syntaxiques retenues dans le cadre du projet Rhapsodie sont :

Étiquette	**Type de Dépendance**
dep	dépendance classique
sub	sujet
pred	prédication
obj	dépendance objet
obl	dépendance oblique
ad	adjonction
root	racine
para	dépendance paradigmatique
junc	jonction
dep_inherited	dépendance classique héritée
sub_inherited	sujet hérité
pred_inherited	prédication héritée
obj_inherited	dépendance objet héritée
obl_inherited	dépendance oblique héritée
ad_inherited	adjonction héritée
root_inherited	racine héritée
para_coord	lien paradigmatique de coordination
para_hyper	lien paradigmatique hypéronymique
para_intens	lien paradigmatique d'intensification
para_disfl	lien paradigmatique disfluent
para_reform	lien paradigmatique de reformulation
para_dform	lien paradigmatique
para_negot	lien paradigmatique de négociation

TABLE 5.2.1 – Tableau récapitulatif des dépendances syntaxiques retenues

Une relation de dépendance se définit comme établissant un lien entre un Lexeme *gouverneur* et un lexème *dépendant*. Cette dépendance est *typée*.

Les parties du discours retenues dans le cadre du projet sont :

1. On appelle classe la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule. En réalité on dit qu'un objet est une instanciation d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence). *"Classe et instance d'objet" issu de CommentCaMarche (www.commentcamarche.net)*

Étiquette	Partie du discours
V	verbe
N	nom
Adj	adjectif
Cl	clitique
Qu	mots qu- (relatifs et interrogatifs)
Pro	pronom
D	déterminant
Adv	adverbe
Pre	préposition
J	joncteur
I	interjections (ah, oh, bon, ben, euh, hein ...)
CS	conjonction de subordination
X	indéterminés (partie inaudible, amorces, positions non instanciées annotées &).

TABLE 5.2.2 – Tableau récapitulatif des parties du discours retenues pour le projet.

Features

L'étiquetage des lexèmes créé une liste d'attributs relatifs à ce lexème. Ces attributs sont appelés des *features*. De nombreuses autres features existent comme le genre, le nombre, etc.

L'ensemble de la structure objet considérée est représentée en figure 5.2.2. On le voit, cette structure présente une certaine complexité, mais permet à la fois de représenter correctement les systèmes d'annotations syntaxiques et prosodiques considérés et les résultats d'une analyse syntaxique automatique. A partir de cette structure de données, de nombreux traitement peuvent être effectués. Je vais maintenant présenter plus en détail les différentes classes Word, Node, Token, Lexeme et Dependency ainsi que leurs attributs et méthodes.

Méthode d'instance

En programmation orientée objet, la méthode est une fonction faisant partie de l'interface d'un objet. Les méthodes peuvent aussi être appelées des *méthodes d'instance* et n'agissent que sur un seul objet à la fois. D'autres méthodes, appelées *méthodes de classe* ou *méthodes statiques*, permettent d'agir sur tous les objets d'une même classe. Dans de nombreux langages, l'encapsulation permet de gérer les droits d'accès à une méthode ou à une donnée membre. Les méthodes publiques sont les méthodes accessibles de l'extérieur de l'objet. En principe, l'utilisation des méthodes publiques (et donc de son interface) d'un objet est le seul moyen pour accéder (indirectement) à l'état de l'objet. [2]

Les attributs quant à eux sont des entités qui définissent les propriétés d'objets, d'éléments, ou de fichiers. Les attributs sont habituellement composés d'un identificateur (ou nom ou clé) et d'une valeur.

2. Wikipédia *(http ://en.wikipedia.org/wiki/Class_(computer_programming)).*

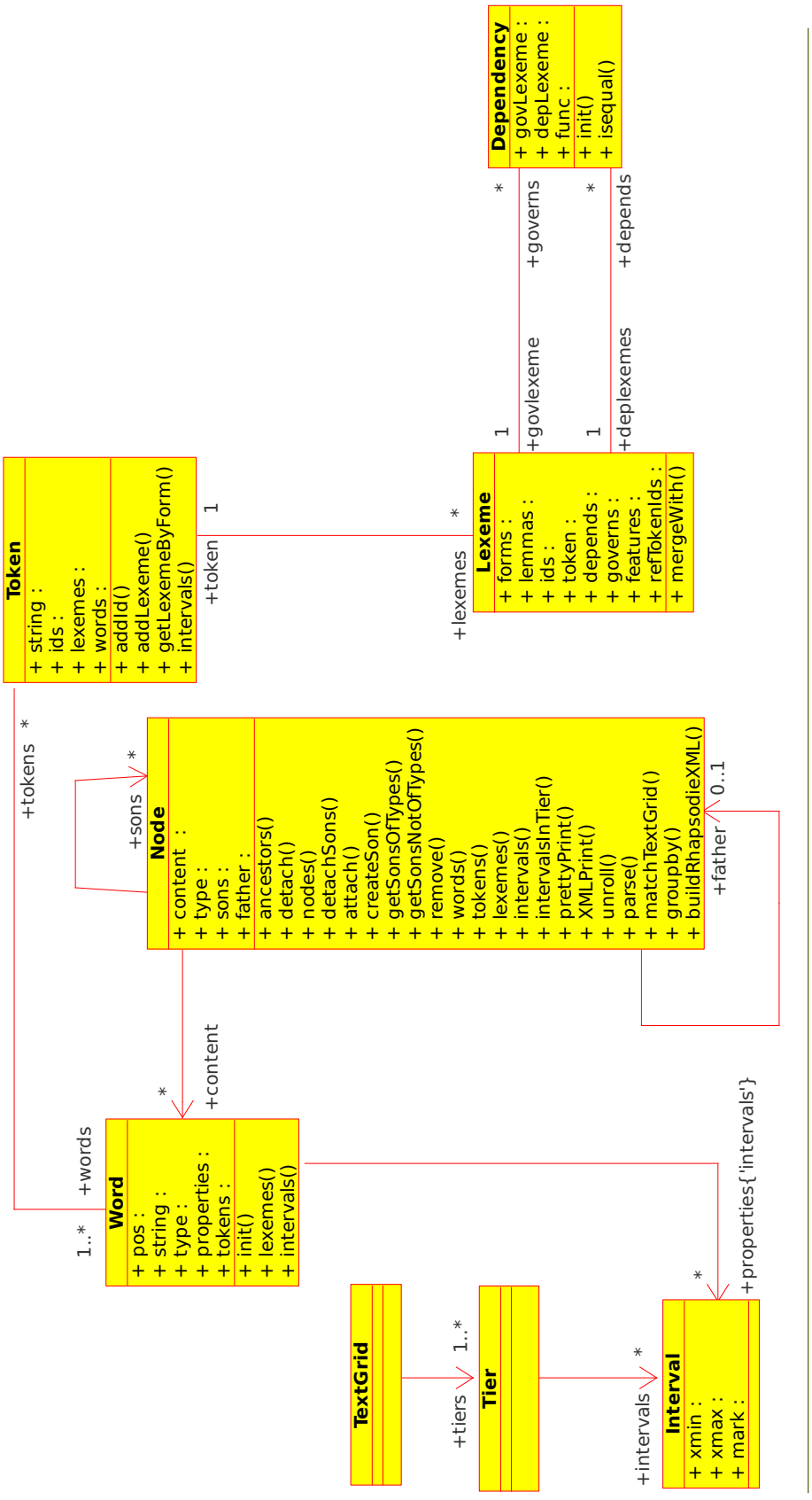


FIGURE 5.2.2 – Diagramme UML de classes haut niveau

Diagramme de Classes

Ci-dessus le diagramme de classes UML du formalisme implémenté. Une classe est représentée par un rectangle séparée en trois parties :

- la première partie contient le nom de la classe
- la seconde contient les attributs de la classe
- la dernière contient les méthodes de la classe

On rappelle qu'une classe déclare des propriétés communes à un ensemble d'objets. La classe déclare des attributs représentant l'état des objets et des méthodes représentant leur comportement.

Les flèches reliant les classes entre-elles signalent les relations entre ces classes ou leur connexion sémantique (relation logique) ; on parle alors d'association. Une association peut être nommée. L'invocation d'une méthode est une association. Elle peut être binaire, dans ce cas elle est représentée par un simple trait, ou n-aires, les classes sont reliées à un losange par des traits simples. Ces relations peuvent être nommées.

Plusieurs critères caractérisent ces relations :

- la multiplicité : comparable aux cardinalités du système Merise [3], sert à compter le nombre minimum et maximum d'instances de chaque classe dans la relation liant 2 ou plusieurs classes. Ainsi la flèche reliant la classe TextGrid à la classe Tier nous indique qu'un objet TextGrid doit contenir au minimum un objet (+tiers 1*).
- la navigabilité : indique si on pourra accéder d'une classe à l'autre. Si la relation est entre les classes A et B et que seulement B est navigable, alors on pourra accéder à B à partir de A mais pas vice versa. Par défaut, la navigabilité est dans les 2 sens. De cette façon on peut remarquer que la relation entre les classes Lexeme et Token est biunivoque c'est-à-dire qu'il faut 1 token (+token 1) (unité lexicale) pour donner un lexème mais qu'en revanche aucun ou plusieurs lexèmes peuvent être reliés à un token (+ lexeme *).

5.3 Word

La classe Word modélise les mots de la transcription annotée.

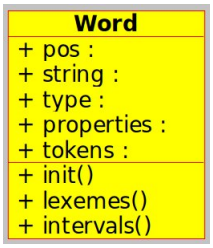


FIGURE 5.3.1 – Classe Word

3. La méthode Merise est une méthode d'analyse, de conception et de réalisation de systèmes d'informations informatisés.

5.3.1 Attributs

– pos : position ordinale du mot dans le texte annoté.

– string : le libellé du mot dans l'annotation

– type : le type d'un mot est particulier s'il s'agit d'un symbole de l'annotation syntaxique tel que listé dans la table 2.4.1. Il est "standard" si c'est une unité lexicale de la transcription.
Par exemple, le type d'un symbole du balisage reste un symbole du balisage (le symbole >+ sera stocké comme >+). En revanche le type de symbole "*chat*" sera "standard". Le type des unités lexicales de la transcription sera donc "standard" et de type "balise" pour chacun des symboles de l'annotation (>+ sera de type >+, (sera de type ( ).

– properties : liste des différentes propriétés d'un mot.

  – properties{'intervals'} : si une correspondance a été établie entre une annotation et un objet TextGrid, cette propriété donne la liste des intervalles qui lui correspond dans la Tier de référence du TextGrid.

  – properties{'Speaker'} : donne le numéro du locuteur qui a prononcé ce mot

– tokens : liste des objets Token auxquels est lié ce mot

Le texte de l'annotation n'est ainsi plus considéré comme une chaîne de caractères, mais comme une liste d'objets Word "standard" ou "balise". De cette manière, si on veut obtenir le texte de l'annotation, il suffit de concaténer les champs *string* des mots le constituant. Si on veut obtenir la position originale d'un mot particulier de l'annotation, il suffit de considérer la valeur de son champ *pos*.

5.3.2 Méthodes

– init(self,string) : constructeur de la classe Word. Prend en entrée le libellé du mot et en déduit son type.

– liste=lexemes(self) : si une analyse syntaxique automatique a été effectuée, renvoie la liste des lexèmes rattachés au mot self. Cette liste est obtenue en concaténant les objets lexemes reliés aux tokens auxquels est lié self.

– liste=intervals(self) : si une correspondance avec un objet TextGrid a été effectuée, renvoie la liste des intervalles de self, identique à properties{'intervals'}

5.4 Node

Cette classe encapsule toutes les propriétés et méthodes relatifs aux unités syntaxiques.

FIGURE 5.4.1 – Classe Node

5.4.1 Attributs

– content : liste des mots que ce nœud contient directement

– sons : liste des nœuds que ce nœud engendre dans la hiérarchie.

– father : nœud père de ce nœud dans la hiérarchie. None pour le nœud racine.

– type : chaîne de caractères qui désigne le type d'unité syntaxique représentée par ce nœud. Les différents types possibles sont : root, iu, pile, layer, embedded, graft, prekernel, intro, kernel, inkernel, parenthesis, integratedprekernel, integratedpostkernel, postkernel, discursivemarker etc.

5.4.2 Méthodes

La classe Node est centrale dans l'exploitation de la structure arborée considérée. A ce titre, elle offre à l'utilisateur de nombreuses fonctions permettant de parcourir l'arborescence, de la modifier et d'effectuer des requêtes à son sujet. Au cours de mon travail, j'ai donc été amenée à définir sur les nœuds de nombreuses méthodes. Ces méthodes peuvent être partagées en deux groupe. Le premier inclut les méthodes mineures de la classe Node, qui correspondent à des traitement de faible complexité. Le deuxième, qui regroupe les méthodes majeures, correspond à des traitements complexes que je présenterai plus en détail au chapitre 6. Je me contente pour l'heure de les présenter de manière succincte.

Méthodes mineures

– liste=ancestors(self) : donne la liste des nœuds ancêtre de self dans la hiérarchie [4]

4. Pour un nœud donné, par exemple un nœud *prekernel* on pourra ainsi obtenir l'ensemble de ses nœuds parents.

- liste=nodes(self) : donne la liste des nœuds descendants de self dans la hiérarchie
- detach(self) : détache self de son actuel père
- liste=detachSons(self) : détache tous les enfants de self et en envoie la liste
- attach(self,father) : attache self comme nouvel enfant du nœud father. Détache au besoin self de son père précédent
- nouveaunœud=createSon(self,type) : crée un enfant de type spécifié à self
- liste=getSonsOfTypes(self,types) : renvoie la liste des descendants de self d'un type dans types.
- liste=getSonsNotOfTypes(self,types) : renvoie la liste des descendants de self qui ne sont pas d'un type dans types.
- remove(self,word, recursive) : si recursive est faux, supprime le mot word de self.content. Sinon, supprime le mot word du contenu de tous les nœuds de self.nodes()
- liste = words(self,recursive) : si recursive est faux, renvoie self.content. Sinon, renvoie l'agrégation du contenu de tous les nœuds de self.nodes()
- liste = tokens(self, recursive) : renvoie les intervalles reliés à des mots de self.words(recursive)
- liste = lexemes(self, recursive) : renvoie les lexemes reliés à des tokens de self.tokens(recursive)
- liste=intervals(self,recursive) : si une correspondance n'a pas été établie avec un TextGrid, renvoie une liste vide. Sinon, renvoie tous les intervalles correspondants aux mots de self.words(recursive). Ces intervalles sont issus d'une Tier de référence, comme on le verra dans la description de la méthode matchTextGrid.
- liste = intervalsInTier(self,tier,strict,recursive) : si une correspondance n'a pas été établie avec un TextGrid, renvoie une liste vide. Sinon, utilise les intervalles de référence self.intervals(recursive) pour en déduire les intervalles d'une autre Tier tier reliés à ce nœud. Le paramètre strict indique si on souhaite que les intervalles trouvés soient strictement compris dans la portion du signal correspondant aux intervalles de référence, ou si au contraire, on souhaite choisir tous les intervalles qui ont une intersection non nulle avec les intervalles de référence.
  Par exemple, ayant établi une correspondance avec les intervalles d'une tier "syllabe", on peut vouloir chercher les intervalles d'une autre tier "f_0" qui leur correspondent.
- prettyPrint(self), XMLPrint(self) : un nœud peut être converti en texte ou XML. L'ensemble de la hiérarchie sous self est exportée sous forme de texte. Ces fonctions sont récursives, dans le sens où un nœud appelle les fonctions de ses enfants, de manière à ce que toute la hiérarchie sous lui soit traitée.

Méthodes majeures

- liste = unroll(self) : cette méthode opère le dépliage [5] tel que décrit en section 6.1. Elle renvoie une liste des différentes phrases (entendues comme des listes d'objets Word)

5. le dépliage est l'opération qui consiste à fournir des segments *réarrangés* (on réorganise la transcription pour la rendre plus linéaire) permettant ensuite une analyse syntaxique automatique par FRMG.

qui correspondent au dépliage de l'arbre sous ce nœud. L'usage typique est d'appeler unroll sur le nœud racine.

– parse(self) : cette méthode opère le repliage des analyses syntaxiques automatiques telles que décrit en section 6.1. Il procède d'abord à un dépliage de l'arbre sous le nœud self, puis à une analyse syntaxique automatique de chacune des phrases correspondantes. Le résultat de cette analyse se traduit par la création d'un ensemble d'objets Token, Lexeme et Dependency. Son principal enjeu est alors de lier ces objets aux différents objets Word sous self dans l'arbre.

– matchTextGrid(self, fileName, tierName) : cette méthode opère la correspondance de l'arbre avec un fichier TextGrid. Un objet TextGrid sera tout d'abord créé depuis le fichier TextGrid. Ensuite, dans cet objet TextGrid, on cherchera l'objet Tier dont le nom est tierName. C'est alors que chacun des mots de self.words(recursive=True), sera mis en correspondance avec les intervalles de la tier *de référence* choisie.

– groupby(self, liste what, liste by) : cette méthode opère une transformation de la structure hiérarchique des nœuds. Elle est appelée *projection* dans la section 6.3.

– buildRhapsodieXML : cette méthode a été développée dans le but de générer le format de données voulu par les membres du projet Rhapsodie. Elle exporte l'ensemble des lexemes et tokens liés au nœud racine, ainsi que toutes les dépendances entre lexèmes après repliage. Par ailleurs, elle exporte également les projections de l'arbre selon les entassements d'une part et selon les unités rectionnelles d'autre part.

5.5 Token

La classe Token regroupe tous les attributs et méthodes relatifs aux objets de type "token" générés par le parseur automatique FRMG [26] (qui sont en réalité les unités lexicales).

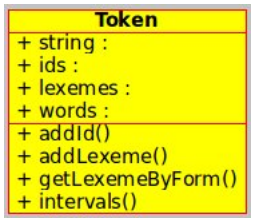


FIGURE 5.5.1 – Classe Token

5.5.1 Attributs

– string : libellé du token

– ids : les différents identifiants correspondant à ce token donnés par le parseur automatique avant repliage

– lexemes : liste de tous les objets Lexeme liés à ce token

– words : ensemble des mots liés à ce token

5.5.2 Méthodes

– addId(self, id) : ajoute un nouvel identifiant au token. Cet identifiant correspond au token dans une des analyses produites après dépliage

– addLexeme(self,lexeme) : ajoute le lexème à la liste des lexemes liés à self

– liste = getLexemeByForm(self, form) : renvoie une liste de tous les lexemes liés à self dont la forme est form

– liste=intervals(self) : renvoie la liste de tous les intervalles liés aux mots de self.words

5.6 Lexeme

La classe Lexeme encapsule les attributs et propriétés relatives aux objets lexèmes renvoyés par le parseur syntaxique automatique FRMG.

FIGURE 5.6.1 – Classe Lexeme

5.6.1 Attributs

– forms : liste de chaînes de caractères donnant les différentes formes sous lesquelles ce lexème est apparu lors des différentes analyses des UR dépliées

– lemmas : liste de chaînes de caractères donnant les différents lemmes sous lesquels ce lexème est apparu lors des différentes analyses des UR dépliées.

– ids : différents identifiants donnés par le parseur automatique pour identifier ce Lexeme lors des différentes analyses des UR dépliées.

– token : objet Token lié à ce lexème

– depends : liste d'objets Dependency pour lesquels ce lexème est le depLexeme

– governs : liste d'objets Dependency pour lesquels ce lexeme est le govLexeme

– features : liste d'attributs relatifs à ce lexeme ainsi que leurs valeurs. De tels attributs sont donnés par analyse des résultats du parseur automatique par l'outil mis au point par KIM GERDES ([37]). Chacun de ces attributs donne la liste de ses valeurs pour ce lexeme dans toutes les UR dépliées. Les attributs considérés incluent :

  – features{'cat'} : catégorie (voir la figure 3.5.2)
  – features{'countable'} : le lexeme est un dénombrable

  - features{'def'}
  - features{'dem'}
  - features{'det'}
  - features{'gender'}
  - features{'nb'}
  - features{'number'}
  - features{'wh'}
  - etc.

- refTokenIds : liste des identifiants du token régissant ce lexeme, dans l'ensemble des UR dépliées où il apparaît

5.6.2 Méthodes

L'objet Lexeme n'a que peu de méthodes. Il est surtout un conteneur d'attributs. Cependant, un certain nombre d'opérations sont effectuées à son niveau :

- mergeWith(self, lexeme) : lors du repliage, permet de regrouper deux lexemes dont on a identifié qu'ils étaient identiques, ce qui arrive souvent du fait de la présence d'entassements.

5.7 Dependency

Regroupe les attributs et propriétés relatifs aux relations de dépendance entre lexemes.

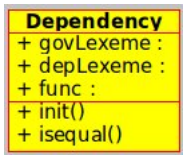


FIGURE 5.7.1 – Classe Dependency

5.7.1 Attributs

- govLexeme : objet instance de la classe Lexeme qui joue le rôle de gouverneur dans cette relation de dépendance

- depLexeme : objet instance de la classe Lexeme qui joue qui indique la fonction du dépendant par rapport à son gouverneur.

- func : chaîne de caractères indiquant l'intitulé de la fonction de la relation de dépendance (voir la figure 3.5.3)

5.7.2 Méthodes

Les seules méthodes de la classe Dependency méritant d'être mentionnées sont son constructeur et celle qui compare deux relations de dépendance de manière à dire si elles sont égales ou non.

– init(self,func,govLexem, depLexem) : crée une relation de dépendance

– isequal(self, dep) : renvoie True si self et dep ont les mêmes fonctions, et les même govLexeme et depLexeme

5.8 TextGrid, Tier et Interval

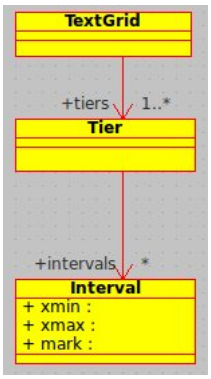


FIGURE 5.8.1 – Schéma UML des classes TextGrid, Tier et Interval

Ces classes représentent l'architecture simple d'un fichier TextGrid. Un objet TextGrid contient plusieurs Tiers, qui contiennent elles mêmes plusieurs objets Interval. Chaque objet Interval a comme attributs ses positions $x_{\min}$ et $x_{\max}$ dans le fichier, ainsi qu'une chaîne de caractères *"mark"*.

5.9 Récapitulatif du formalisme informatique

Classe	Notion linguistique
Word	mots de la transcription
Node	unités/constituants syntaxiques
Token	unités lexicales (Tokens du parseur FRMG)
Lexeme	lexèmes
Dependency	natures des relations de dépendances
TextGrid	fichier TextGrid
Tier	tiers du fichier TextGrid
Interval	intervalles des tiers

TABLE 5.9.1 – Tableau récapitulatif des termes du formalisme objet et des termes linguistiques qui leur sont associés

5.10 Graphe de la hiérarchie objet intono-syntaxique

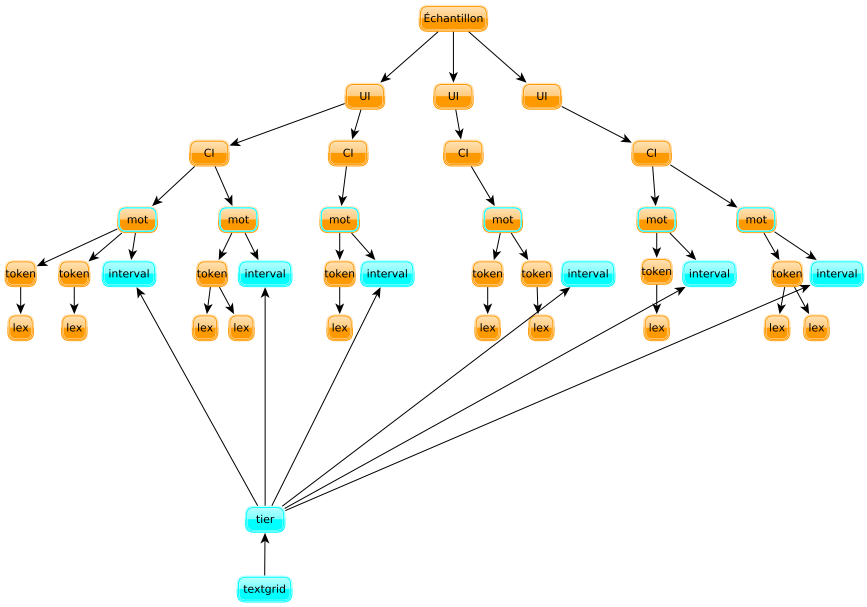


FIGURE 5.10.1 – Graphe de la hiérarchie intono-syntaxique

Chapitre 6

Exploitations du formalisme

6.1 Dépliage de la transcription balisée

Pour récupérer les informations micro-syntaxiques c'est-à-dire les fonctions de chaque constituants des UI et des CI et leurs liens de dépendance nous utilisons le parseur FRMG mis au point par Éric de La Clergerie [26]. Afin que le parseur puisse analyser correctement le texte une étape intermédiaire appelée "*dépliage*" est nécessaire. Cette étape est indispensable en raison des propriétés du français parlé transcrit qui ne présente que peu de similitudes par rapport à la syntaxe de l'écrit, or ce type de parseur est optimisé pour analyser de l'écrit. Les transcriptions de l'oral ne sont ni ponctuées, ni segmentées et comportent un grand nombre de phénomènes propres à l'oral tels que les entassements (voir chapitre 9). Dépliage

On entend par dépliage du texte un réarrangement des entassements et des phénomènes perturbateurs permettant ensuite une analyse syntaxique automatique par un programme informatique. En effet les structures d'entassement, d'enchâssement et de parenthèsage, ou encore les marqueurs discursifs, particulièrement courants à l'oral, ne sont pas analysables en l'état par les analyseurs syntaxiques qui sont calibrés pour l'écrit. Les parseurs ne savent pas traiter les disfluences et font encore beaucoup d'erreurs sur les coordinations (difficulté avec le rattachement du deuxième conjoint). Le dépliage va donc explorer chaque chemin de l'entassement et donne une UR bien formée sans entassement[38]. Pour mener à bien cette tâche il est très important d'avoir préalablement encodé les structure micro et macro-syntaxique dans le même arbre. En effet aussi bien les unes que les autres font l'objet d'un dépliage.

Nous avons vu qu'il est nécessaire de fournir des segments syntaxiques débarrassés de tout phénomènes de l'oral à des parseurs automatique. L'opération de dépliage consiste donc à recréer à partir de la transcription des segments analysables par le parseur, les entassement sont donc "désentassés", les disfluences extraites etc. Voici un exemple de dépliage d'un entassement :

```
mark-up text:
Je dirais que vous avez donné { quelque chose de plus |} à la femme //+ {| des armes de persuasion } //

unrolled text:
Je dirais que vous avez donné quelque chose de plus à la femme
Je dirais que vous avez donné à la femme des armes de persuasion
```

Figure 6.1.1 – Résultat du dépliage d'un entassement

Ainsi déplié le texte retrouve une linéarité qui facilitera la tâche du parseur FRMG. Ici linéarité

on constate de plus que chaque alternative d'entassement (chaque couche qui compose la l'entassement) est explorée, chaque couche de chaque pile va venir occuper le rôle syntaxique qu'elle peut occuper. On obtient donc autant d'alternatives qu'il y a d'entassement et de couches dans un entassement. On notera également que les éléments de type introducteurs (ici, alors), marqueurs discursifs etc [...] sont "séparés" des autres UR car eux aussi peuvent perturber l'analyse syntaxique automatique. En aucun cas ils ne seront ignorés, ils seront réintégrés aux autres UR après l'analyse en ligne du parseur FRMG grâce à un algorithme de "repliage".

repliage

De plus, il est important de voir que l'information d'entassement n'est pas suffisante à elle seule pour fournir le dépliage de l'arbre, mais que le multi-arbre complet est bien nécessaire à cette tâche. En effet l'information d'entassement est d'ordre micro-syntaxique et n'est pas suffisante pour la tâche de dépliage car on l'a vu plus haut, les marqueurs discursifs aussi peuvent poser problème et que l'on a donc besoin parallèlement de l'information macro-syntaxique.

6.2 Repliage : enrichissement des annotations micro-syntaxiques du parseur automatique

Après dépliage et parsing du parseur FRMG, le parseur renvoi pour chaque segment qui lui est soumis un fichier XML contenant les lexèmes constituant les mots du segment, leurs dépendances entre eux et leurs fonctions syntaxiques. Ces fichiers sont ensuite reformater par un script mis au point par Kim Gerdes et que j'ai intégré au parseur général.

Nous avons vu que la phase de dépliage visait à simplifier le parsing de l'analyseur automatique. Pour chaque dépliage possible d'UR, une analyse automatique produit un ensemble de traits syntaxiques et un système de dépendance. Le repliage consiste à répercuter tous ces traits et liens de dépendance sur la transcription originale. Cette phase complexe est rendue possible par l'identification des éléments communs aux différents dépliages et par le fait que les données sont représentés comme des objets, pouvant avoir plusieurs attributs et liens entre eux. Pour chaque lexème, on obtient ainsi autant de rôles syntaxiques et de liens de dépendance que le nombre d'UR dépliées dans lesquelles ce lexème apparaît. L'avantage de cette phase est qu'elle permet de désambiguïser l'analyse syntaxique de certains lexèmes, par exemple en choisissant pour trait syntaxique (genre, nombre etc...) de chaque mot celui qui apparaît le plus de fois ou, en cas d'égalité, de choisir le dernier (critère de proximité).

Ainsi, dans :

le livreur s'éloigne pour voler { un | une } baguette (M0024 Corpus Rhapsodie)

le lexème baguette reçoit le trait féminin, malgré les segments dépliés contradictoires envoyés au parseur automatique :

le livreur s'éloigne pour voler un baguette
le livreur s'éloigne pour voler une baguette

Une propriété intéressante de l'approche par dépliage/repliage est qu'elle pourrait permettre — avec seulement des modifications mineures — de considérer plusieurs annotations

différentes du même texte, supprimant ainsi les phases de corrections manuelles contraignantes pour les annotateurs. En effet, chaque annotation différente produirait son propre lot de dépliages à analyser. Le repliage permettrait alors de rendre compte des différences entre les annotations et rendrait possible une plus grande robustesse en cas d'ambiguïtés dans les choix des annotateurs par un choix statistique ou bien en conservant l'ensemble des variantes d'annotation dans le structure.

Pour finir, la phase de repliage permet de visualiser l'ensemble des traits et dépendances ainsi construits directement sur la transcription originale.

On obtient ainsi pour chaque mot du corpus transcrit, un ou des lexèmes associés, les liens qu'il entretient avec les autres lexèmes et leur fonction syntaxique. Ces information viennent enrichir le multi-arbre créé à partir du balisage qui stockait déjà l'information macro-syntaxique et d'entassement. On obtient dès lors un super arbre contenant toute l'information syntaxique souhaitée pour l'analyse de ce corpus (pour le moment). multi-arbre

Une fois cette étape réalisée on obtient un arbre de dépendances dont chacune des relations entre les mots est typée en fonction des fonctions syntaxiques régissant leur relation. Grâce au logiciel Arborator mis au point par Kim Gerdes [37] on dispose d'un outil efficace permettant la visualisation des ces dépendances et leur correction le cas échéant. Arborator

Voici l'exemple d'un arbre et des dépendances syntaxiques que l'on peut observer avec le logiciel Arborator après les étape détaillées ci-dessus :

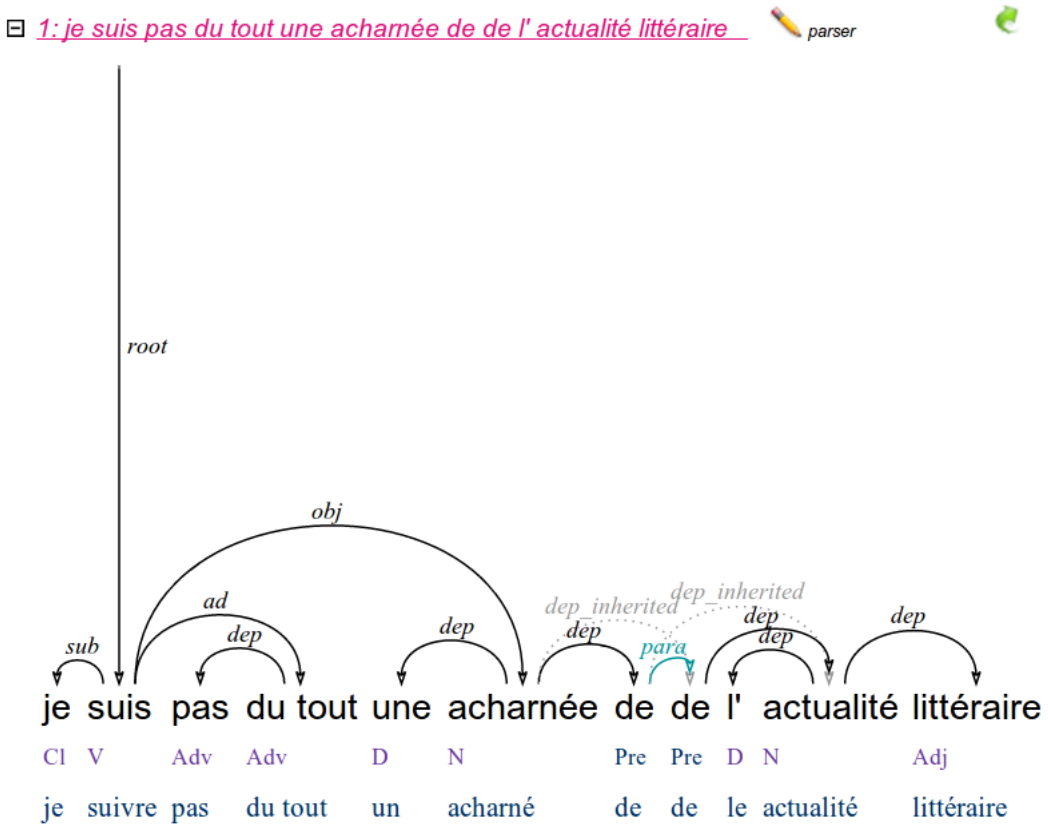


Figure 6.2.1 – Arbre de dépendance visualisé sous Arborator

6.3 Projections, transformation de l'arbre

Considérons l'exemple suivant :

il guérit { en donnant la vie | en donnant sa propre vie } // (M2003 Corpus Rhapsodie)

Dans un premier temps chaque élément est analysé successivement et classé selon ses caractéristiques propres. L'opération consiste donc à stocker chaque mot et chaque symbole constituant l'exemple. On obtient alors l'arbre suivant :

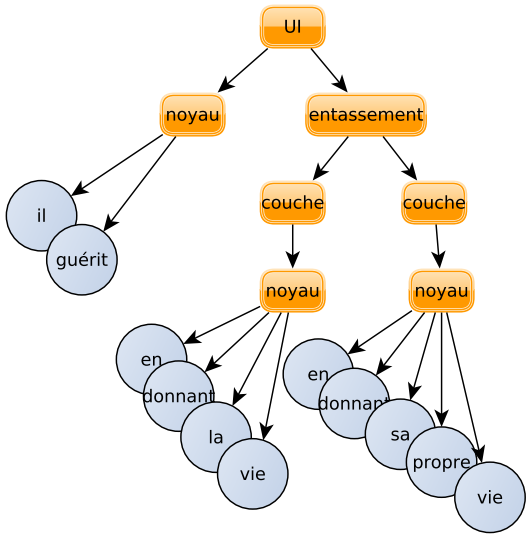


FIGURE 6.3.1 – Arbre comportant l'information macro-syntaxique et d'entassement

On constate qu'en l'état de multi-arbre, l'information macro-syntaxique et d'entassement sont regroupée. L'information globale est intéressante en soi mais inélégante à visualiser dans un arbre (un graphe serait plus approprié) et on pourrait souhaiter n'en étudier qu'une partie.

projection

Pour faire un parallèle géométrique, on peut difficilement conceptualiser un hypercube. L'opération de projection ou d'extraction consiste ainsi à diminuer le nombre de dimensions présentes dans le multi-arbre, de manière à se focaliser sur un point de vue particulier, cela revient à extraire un arbre particulier, par la sélection des nœuds désirés. Si l'on effectue une projection sur cet arbre afin d'obtenir de façon séparée les deux types d'information encodées on obtient alors les arbres d'entassement en figure 6.3.3 et macro-syntaxique en figure 6.3.2 suivants :

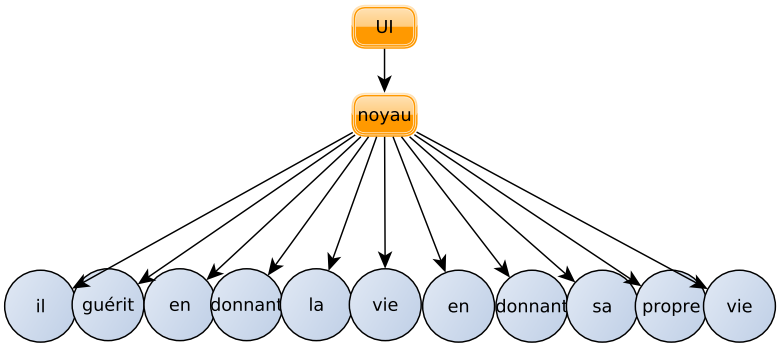


FIGURE 6.3.2 – Arbre macro-syntaxique

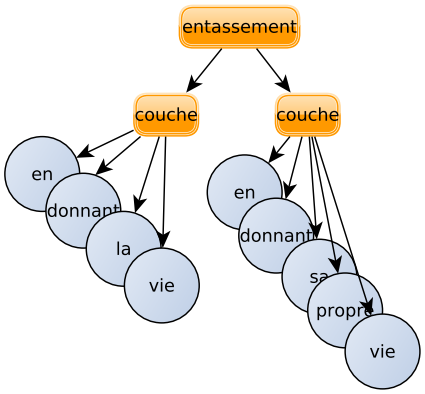


FIGURE 6.3.3 – Arbre d'entassement

Ces arbres permettent de savoir à quel type d'UR les mots du segment appartiennent, s'ils font ou non partie de l'entassement et selon quelle stratification au sein de l'entassement ils s'organisent.

D'une manière générale, on voit que cette opération de projection revient à modifier la structure hiérarchique des objets Node. Plus particulièrement, certains nœuds sont supprimés. Dans le cas de la création d'un arbre d'entassement par exemple (figure 6.3.3), tous les nœuds de type kernel, inkernel, intro, embedded, etc. sont supprimés. Leur contenu est transféré dans les nœuds de type layer ou pile les plus proches. Dans le cas de la création d'un arbre macro syntaxique, on voit en figure 6.3.2 que les nœuds de type kernel répartis dans les différents entassements sont regroupés lors d'une suppression de ces entassements.

Toutes ces opérations peuvent donc être comprises comme des cas particuliers d'un regroupement de certains nœuds de l'arbre et de l'exclusion de certains autres. Ce regroupement prend en paramètre les types des nœuds à regrouper et les types des nœuds en fonction desquels regrouper. Son prototype est :

– groupby(self, liste what, liste by)

L'objectif de ce regroupement est de modifier l'arbre self de telle manière à ce que tous les nœuds restants soit d'un type contenu dans la liste *what* ou bien dans la liste *by*. Les

nœuds dont le type n'est ni dans *what* ni dans *by* sont éliminés et leur contenu déplacé vers le nœud d'un type à garder qui en est l'ancêtre le plus proche. Les nœuds d'un type contenu dans la liste *what* sont par ailleurs regroupés entre eux de telle manière à ce que chaque nœud d'un type dans *by* ne contienne au maximum qu'un seul nœud de chaque type dans *what*.

Par exemple, la projection du multi-arbre pour la création de l'arbre macro-syntaxique présentée figure 6.3.2 élimine les nœuds d'entassements et de couches et s'effectue par :

6.4 Requêtes

6.4.1 Principes généraux

A partir de l'arbre on peut vouloir exporter les données de syntaxe ou de prosodie correspondant à certains critères. Cette recherche d'information est rendue possible par le biais de toutes les méthodes de gestion de la structure qui ont été présentées au chapitre 5. Dans la structure syntaxique des nœuds, on peut rechercher tous ceux qui sont d'un certain type. En leur sein, on peut chercher tous les lexèmes qui présentent une valeur donnée pour un trait choisi, et en exporter alors un autre trait tel que leur fréquence fondamentale. De plus on peut vouloir exporter certaines informations relatives à la position de ces lexèmes dans l'arborescence syntaxique.

Toutes ces opérations peuvent être effectuées efficacement en exploitant la structure présentée au chapitre 5. Prenons comme premier exemple la recherche de tous les entassements d'un échantillon du corpus. Tout d'abord, la structure de donnée est construite :

```
texte_annote=' ici la transcription annotée'
root=buildTree(texte_annote)
```

Ensuite, on cherche dans cet arbre tous les nœuds de type "*pile*" :

```
piles = [n for n in root.nodes() if n.type=='pile']
```

L'information recherché est donc trouvée en deux lignes. Un exemple plus complexe de recherche d'informations dans la structure de données sera donné en partie III.

- On peut très facilement extraire tous les symboles pré-noyaux d'un arbre
- On peut très facilement déterminer quels symboles sont à la fois dans des entassements et énoncés par un locuteur donné
- On peut facilement ne conserver d'un arbre que les entassements etc .

6.4.2 Conversion en formats structurés XML pour la base de données RhapSQL

Donnée Structurées

structure XML

L'un des objectifs du traitement du balisage est l'obtention de donnée structurées. Dans le cadre du projet, on cherche à générer à partir du corpus annoté l'ensemble des arbres topologiques et des arbres d'entassement possibles. Pour ce faire, l'équipe de recherche a opté pour une structure XML . Ce format sert de format d'import-export pour le corpus annoté et la base SQL du projet.

La problématique qui se pose est donc de convertir les structures de données obtenues vers des fichiers structurés XML. La représentation du balisage sous la forme d'arbre permet d'effectuer cette tâche de manière triviale. Par des algorithmes récursifs, on peut convertir un arbre en format structuré de type XML.

Pour l'exemple a. on aura la représentation de la figure 6.4.1 pour la représentation topologique et la figure 6.4.2 : pour la représentation de l'entassement :

^qui (donc) reste { toute seule | fort étonnée } // (M0002 Corpus Rhapsodie)

```
<constree const_type="topology" idref="a">
    <const type="iu">
        <const type="intro">
            <const const_type="lexeme" idref="qui"/>
        </const>
        <const type="inkernel">
            <const const_type="lexeme" idref="donc"/>
        </const>
        <const type="inkernel">
            <const const_type="lexeme" idref="reste"/>
            <const const_type="lexeme" idref="toute"/>
            <const const_type="lexeme" idref="seule"/>
            <const const_type="lexeme" idref="fort"/>
            <const const_type="lexeme" idref="étonnée"/>
        </const>
    </const>
</constree>
```

FIGURE 6.4.1 – Arbre macro-syntaxique

1

```
<constree const_type="pile" idref="a">
    <const type="pile">
        <const type="layer">
            <const const_type="lexeme" idref="toute"/>
            <const const_type="lexeme" idref="seule'"/>
        </const>
        <const type="layer">
            <const const_type="lexeme" idref="fort"/>
            <const const_type="lexeme" idref="étonnée"/>
        </const>
    </const>
</constree>
```

FIGURE 6.4.2 – Arbre d'entassement XML

Le multi-arbre n'est pas généré en format XML dans le cadre du projet, il n'est utilisé que comme structure relais permettant l'ensemble des traitement nécessaires à la réalisation des tâches de dépliage et de projection.

1. Dans ces exemple les numéros d'identifiants des lexèmes ont été remplacés par les lexèmes en question à des fins pédagogiques.

Chapitre 7

Implémentation

7.1 Algorithme et script de construction du multi-arbre à partir du balisage

Arbre

Hiérarchie

On appellera arbre, une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud. Le nœud initial étant appelé racine (root). Dans l'arbre, chaque élément possède un certains nombre d'éléments enfants (children) au niveau inférieur. Du point de vue de ces éléments enfants, l'élément dont ils sont issus au niveau supérieur est appelé père. Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a des nœuds enfants. Un nœud n'ayant aucun enfant est appelé feuille. Le nombre de niveaux total, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé hauteur de l'arbre. Le niveau d'un nœud est appelé profondeur.

Dans ce cas, j'ai choisi d'implémenter un parseur qui construit un arbre multiple ou englobant qui représente à la fois l'information micro et macro-syntaxique encodées par le balisage manuel. De cette façon chaque nœud correspond à un type d'unité et est typé en tant qu'unité illocutoire, entassement, enchâssement, marqueur discursif et de manière générale tout typage donné par le balisage. Ainsi, cet arbre intègre toutes les informations contenues dans le balisage. L'implémentation d'un tel arbre présente l'avantage de pouvoir être parcouru selon un point de vue macro et micro-syntaxique ou les deux en même temps, permettant différents traitements impliquant ou non toute l'information.

Initialization
- root = node(type='root')
- A=root.createSon(type='ui')
- currentNode←A
- currentPos = 0
- words = text.split()

while currentPos < len(words)
- switch words[currentPos].type
  - case "standard" : **add word to current node**
    - currentNode.content.extend(words[currentPos])
  - case [,(,(+: **create a new son [] or () and mark it if integrated**
    - A=currentNode.createSon(type = '[]' or '()' respectively)
    - currentNode←A
    - if (+ then A.mark('integrated')
  - case {: **create a new pile and its first layer**
    - A=currentNode.createSon(type = '{}')
    - currentNode←A.createSon(type = 'layer')
  - case < or <+: **if currentNode is a *kernel*, it should be a prekernel, and focus goes to father. If currentNode is not a *kernel*, then create a new prekernel, copy content of currentNode except *kernel* stuff, and move it to this new prekernel**
    - if 'kernel' in currentNode.type:
      - set currentNode.type to 'prekernel', 'integratedprekernel' respectively
      - currentNode←currentNode.father
    - else:
      - A←currentNode.content
      - reset all content of currentNode except nodes of type 'intro', 'kernel', 'prekernel', 'postkernel', 'integratedprekernel', 'integratedpostkernel'
      - B=currentNode.createSon(type='prekernel' or 'integratedprekernel' respectively)
      - copy all content of A into B except nodes of type 'intro'
  - case > or >+: **go to container parent, and create a new postkernel as currentNode**
    - while currentNode.type not in ['[]', 'layer', '()', 'ui']:
      - currentNode←currentNode.father
    - currentNode←currentNode.createSon(type = 'post kernel' or 'integratedpost kernel' respectively)
  - case ": **find next " and insert all content in between to a new discursivemarker**
    - currentPos ← currentPos+1
    - beginPos ← currentPos
    - while words[currentPos].type != ":
      - currentPos← currentPos+1
    - newNode=currentNode.createSon('discursivemarker')
    - newNode.content.append(words[beginPos:currentPos])
  - case ^: **add next word to a new intro node**
    - A=currentNode.createSon(type='intro')
    - add next word to A
    - currentPos←currentPos + 1
  - case //: **go to first container parent, and if its sons are ui, create a new one, else, insert all its content into a new ui son, and then create a new one**
    - while currentNode.type not in ['[]', 'layer', '()', 'ui']:
      - currentNode←currentNode.father
    - if currentNode.type == 'ui'
      - currentNode←currentNode.father.createSon(type='ui')
    - else:
      - insert node A of type currentNode.type between currentNode and currentNode.father
      - set currentNode.type to 'ui'
      - currentNode←A.createSon(type='ui')

Algorithme 7.1 Algorithme de construction du multi-arbre à partir du balisage (1)

- case] or) or }: **go to corresponding parent node. For ')', mark it as integrated if node is integrated. For ']' and ')' and if node contains node of type different than 'ui', then change its type. Finally, if ']', node must be included in a *kernel*. If it is not the case, create that kernel and move currentNode in it.**
  - until currentNode.type == '[]', '()' or '{}' respectively
    - currentNode ←currentNode.father
  - if ')' AND currentNode.ismarked('integrated')
    - set currentNode.type to 'integratedinkernel'
    - currentNode.unmark('integrated')
  - elseif ']' or ')': AND if there are sons of currentNode that are not of type 'ui' or if content of currentNode is not empty
    - set currentNode.type to 'inkernel' for ')' or to 'embedded' for ']'
  - if ']' and currentNode.father.type not in ['*kernel*']:
    - A←currentNode.father
    - currentNode.detach()
    - kernels← A.getSonsOfType['kernel']
    - if kernels is empty, father←A.createSon('kernel') and move A.content into father.content
    - else father←kernels[0]
    - currentNode.attach(father)
  - currentNode ←currentNode.father
- case |: **find parent pile, and create a new layer**
  - until currentNode.type == '{}'
    - currentNode ←currentNode.father
  - currentNode←currentNode.createSon(type='layer')

- case |}: **find parent pile, mark it as interrupted and go to its parent node**
  - until currentNode.type == '{}'
    - currentNode ←currentNode.father
  - currentNode.mark('interrupted')
  - currentNode ←currentNode.father

- case {|: **find the last interrupted pile, unmark it, and create a new layer**
  - until currentNode.type == 'root'
    - currentNode ←currentNode.father
  - A= last (currentNode.getSonsOfType('{}')) where ismarked('interrupted'))
  - A.unmark('interrupted')
  - currentNode=A.createSon(type='layer')

currentPos←currentPos+1

Termination

- delete all nodes such that neither them nor their progeny have any content
- for all nodes A in the tree:
  - if (A.type is not in '*kernel*', discursivemarker, intro) and A.content is not empty and A has no son of type kernel:
    - B=A.createSon(type='kernel')
    - move content of A into content of B

Algorithme 7.2 Algorithme de construction du multi-arbre à partir du balisage (2)

7.2 Dépliage de l'arbre

Le dépliage de l'arbre est la tâche qui consiste à générer une liste d'UR (séquences d'objets words) compatibles avec la structure arborée. Plus précisément, on considère qu'une phrase du dépliage ne contient plus d'entassements, ni de marqueurs discursifs etc. Ainsi, en cas de présence d'entassements dans l'arbre, chaque UR dépliée contiendra une des différentes possibilités de choisir une seule des couches de chaque entassement. D'autres unités syntaxiques viennent jouer un rôle plus restreint dans le dépliage, comme les marqueurs discursifs qui sont extraits de leur contexte, ou bien les UI enchâssées, qui sont remplacés par un symbole "XXX" pour être traités comme une phrase à part entière. Un avantage très notable du multi-arbre est qu'elle rend très facile le dépliage de l'arbre une fois qu'elle a été construite grâce à l'algorithme présenté en section 7.1.

Multi-arbre

L'algorithme de dépliage fait intervenir deux fonctions annexes, dénommées addLists et multiplyLists.

addLists(a,b) Cette fonction prend en entrée deux listes $a = [a_1, \dots a_N]$ et $b = [b_1, \dots, b_M]$ et renvoie

$$[a_1, \dots a_N, b_1, \dots, b_M]$$

de longueur $N + M$.

multiplyLists(a,b) Cette fonction prend en entrée deux listes $a = [a_1, \dots a_N]$ et $b = [b_1, \dots, b_M]$ et renvoie

$$[[a_1, b_1], \dots, [a_1, b_M], \dots, [a_N, b_1], \dots, [a_N, b_M]]$$

de longueur $N \times M$.

Le principe de l'algorithme de dépliage est de constituer la liste des phrases de manière récursive. Lorsqu'on appelle la fonction de dépliage sur un nœud, il l'appelle lui même sur ses enfants et combine ces résultats. Si le nœud en cours est la racine, l'ensemble des nœuds de l'arbre sont ainsi sollicités et le résultat est la réponse cherchée.

Récursivité

L'essentiel de la subtilité de l'algorithme est que lorsqu'on appelle la fonction de dépliage sur un nœud non racine, elle renvoie deux listes de phrases. La première de ces listes, appelée *absolue*, contient les phrases à faire remonter inchangées jusqu'à la racine. Par exemple, quelle que soit sa profondeur, si on rencontre un nœud de marqueur discursif, son contenu ne sera pas combiné avec son contexte, mais bien remonté intact. Une deuxième liste, appelée *relative*, contient les phrases qui sont susceptibles de se combiner et d'être augmentées progressivement lors de la remontée dans l'arbre. C'est le nœud père qui effectue l'agrégation des listes relatives et absolues de ses enfants pour renvoyer son propre résultat. Si le nœud est le nœud racine, alors il renvoie une seule liste, qui est le résultat cherché.

Algorithme
entrées nœud en court : self
Initialisation absolu←[]
relatif←[self.content]
Pour chaque enfant son de self.sons [sonAbsolu, sonRelatif] = son.unroll()
– si son.type = 'couche'
relatif←addLists(relatif,sonRelatif)
– si son.type = greffe'
newWord←Word('xxx')
relatif←multiplyLists(relatif,[[newWord]])
absolu←addLists(absolu, sonRelatif)
– si son.type dans ['iu', 'parenthesis', 'inkernel', 'prekernel', 'postkernel', 'discursivemarker'] : absolu=addLists(absolu,sonRelatif)
– si son.type dans ['integratedprekernel', 'integratedpostkernel', 'integratedinkernel'] :
newWords←création de mots 'virgule' soit avant soit apres soit les deux en fonction du type de nœud
relatif = multiplyLists(relatif, [newWords])
relatif = multiplyLists(relatif, sonRelatif)
– sinon :
relatif = multiplyLists(relatif, sonRelatif)

Terminaison
– Si self.type différent de 'root'
renvoyer (absolu,relatif)
– sinon, renvoyer addLists(absolute,relative)

Algorithme 7.3 Algorithme de dépliage

Par l'utilisation de addLists au lieu de multiplyLists si l'enfant considéré est une couche d'entassement, l'algorithme effectue bien une disjonction des UR de sortie. En effet, on voit que pour la plupart des autres types de nœud, la fonction appelée est multiplyList, qui combine les phrases relatives déjà existantes, tandis que ces phrases sont simplement juxtaposées sans être mélangées après appel à addLists. Le reste de l'algorithme correspond principalement à des finesses d'ajout de ponctuation avant ou après le contenu des nœuds pour guider l'analyseur automatique.

par exemple < sur la sexualité <+ il y a { des choses ex~ | des descriptions { extrêmement crues | extrêmement belles } } // (D2002 Corpus Rhapsodie)

par exemple
sur la sexualité , il y a des choses ex~
sur la sexualité , il y a des descriptions extrêmement crues
sur la sexualité , il y a des descriptions extrêmement belles

FIGURE 7.2.1 – Production du dépliage d'après l'algorithme de dépliage donné en 7.3

7.3 Repliage

L'opération de repliage est, avec celle de création de l'arbre, la plus complexe que j'ai du considérer. Pour des raisons de clarté de l'exposé, je ne rentrerai pas ici dans tous ses détails mais me contenterai d'en souligner les étapes majeures. Le lecteur intéressé est renvoyé au code source du programme en annexe B, à la fonction "parse".

Algorithme

Dépliage de l'arbre La première étape consiste à obtenir toutes les UR possibles de l'arbre.

Pour chaque phrase dépliée

- Obtenir de la liste d'objets Word la chaîne de caractères correspondant à la phrase
- Envoyer cette phrase au parseur FRMG
- Appliquer le filtrage des résultats (Script de KIM GERDES), et ouvrir le fichier XML des résultats
- Extraire du XML la liste des tokens
- Effectuer une correspondance entre les mots et les tokens
- Pour chaque token, une fois cette correspondance effectuée, identifier s'il existe déja dans la structure.
  Si c'est le cas, rajouter ses informations dans ce dépliage à l'objet Token trouvé
  Sinon, créer un objet Token et le lier au mot
- Extraire du XML la liste des Lexemes
- Pour chaque Lexeme :
  - Créer un objet Lexeme
  - Trouver son Token référent (grâce à son identifiant, ajouté en propriété des objets Token)
  - Identifier si cet objet Lexeme existe déjà (possible en cas d'entassements)
  - Si c'est le cas, effectuer un "merge" des deux objets. Sinon, ajouter le lexeme aux Lexemes du Token trouvé
- Extraire toutes les dépendances du fichier XML
- Pour chaque dépendance, construire un objet Dependancy correspondant, après avoir trouvé les objets Lexeme concernés

Terminaison

- Pour tous les mots qui ne sont liés à aucun Token ni Lexeme, leur créer un Token et un Lexeme correspondant
- Création ou rajout d'attributs "invisible", "para" et "junct" pour les dépendances entre les lexèmes des différentes couches des entassements et l'extérieur. cf XXX
- Création de liens entre les objets Token et les objets Words, ainsi qu'entre Lexeme et Token

Algorithme 7.4 Algorithme de repliage

7.4 Projections

Projection

L'opération de "projection" permet de modifier la structure de l'arbre de manière à la présenter selon une "vue" particulière, telle que l'arbre des entassements ou l'arbre topologique. Comme on l'a vu en section 6.3, cette opération peut se comprendre comme un regroupement des nœuds entre eux selon certains critères. Assez complexe et faisant encore intervenir de la programmation récursive, l'algorithme de regroupement prend en entrée le nœud considéré (souvent le nœud racine de l'arbre), une liste de types de nœuds à regrouper et une liste de types de nœuds en fonction desquels regrouper.

Récursivité

De manière à fonctionner, cet algorithme utilise plusieurs fonctions utilitaires.

exclude(self) Cette fonction détache le nœud en court, et attribue tous son contenu ainsi que tous ses enfants à son père.

accessibleNodes(self,stopTypes,includeStopNodes) Cette fonction parcourt la descendance de self jusqu'à rencontrer des nœuds dont le type est dans la liste "stopTypes", qui sont alors appelés nœuds "d'arrêt", parce que l'énumération ne va pas plus loin. Tous les nœuds rencontrés en chemin sont renvoyés. Un paramètre includeStopNodes indique si la fonction doit également retourner les nœuds d'arrêt.

filterHierarchiesOf(self, types, by) Cette fonction commence par récupérer les nœuds d'un type contenu dans la liste "types" accessibles jusqu'aux nœuds d'un type contenu dans la liste "by" (par l'appel à getAccessibleNodes). Ensuite, elle regroupe tous les nœuds du même type parmi ces nœuds et fusionne leurs contenus.

mergeUntil(self, others, stopTypes) Cette fonction fusionne avec self tous les nœuds dans la liste "others" ainsi que les nœuds accessibles depuis eux jusqu'à des nœuds d'un type contenu dans la liste "stopTypes"

Algorithme de regroupement

Entrée

- self : nœud en cours
- what : liste de nœuds à conserver et à regrouper
- by : liste de nœuds à conserver mais pas à regrouper

Algorithme

- typesToKeep←union(what,by)
- Exclusion des nœuds à éliminer
  - accessible←self.accessibleNodes(by)
  - utiliser exclude() pour tous les nœuds dans accessible qui sont d'un type qui n'est pas dans typesToKeep
  - recommencer jusqu'à ce qu'aucune modification n'ait été faite
- self.filterHierarchiesOf(what,by) : regrouper les nœuds accessibles du même type
- accessibles←self.accessibleNodes(by) .
- stopBranches←nœuds de accessibles d'un type contenu dans "by"
- pour branchNode dans stopBranches
  - attacher branchNode à self
  - excludedSons=branchNode.getSonsNotOfTypes(typesToKeep)
  - tant qu'excludedSons est non vide :
    - pour chaque excludedSon dans excludedSons :
      - excludedSon.detach()
      - ajouter le contenu de excludedSon à branchNode
      - et maintenant fusionne le nœud branche avec ses enfants jusqu'à ce qu'il n'ait plus d'enfants d'un type à supprimer : sons = excludedSon.detachSons(), puis branchNode.mergeUntil(sons,by)
    - recalculer excludedSons=branchNode.getSonsNotOfTypes(typesToKeep)
  - pour chaque enfant son de branchNode, appeler son.groupby(what,by)
- filtrage final des hiérarchies : self.filterHierarchiesOf(what,by)

Algorithme 7.5 Algorithme de regroupement

L'algorithme de regroupement procède ainsi à un regroupement des éléments de types à garder, et à l'exclusion des nœuds d'un type à éliminer, d'une manière récursive. De cette manière, l'arbre des entassements est obtenu très simplement avec le programme suivant :

```
self.groupby([],['layer','pile','iu'])
self.groupby(['layer'],['pile'])
```

qui permet de tout d'abord éliminer tous les nœuds qui ne sont ni des piles, ni des couches, ni des ui, puis ensuite de regrouper les couches par entassement, et ainsi de supprimer les éventuelles UI qui pourraient se trouver enchâssées dans l'arbre.

7.5 Correspondance avec le TextGrid

Correspondance

L'algorithme de mise en correspondance du texte annoté et du texte transcrit tel qu'il apparaît dans le TextGrid a été un des traitements clé du projet, dans la mesure où il a permis de mettre en commun dans la même structure de données les données syntaxiques et prosodiques du corpus. Cette correspondance consiste à mettre en relation les objets Word de la structure de données avec les objets Interval d'une Tier donnée.

En théorie, cette mise en correspondance ne devait pas poser problème, puisque le contenu de la transcription dans le TextGrid devrait être le même que celui du texte annoté, mis à part les symboles du balisage. Dans la pratique, il en va tout autrement. La première difficulté vient des modifications apportées aux textes par les annotateurs. La deuxième vient du fait que le libellé d'un objet Word (unité lexicale de la transcription) peut se retrouver réparti sur plusieurs objets Interval.

L'algorithme élimine tous les symboles de balisage du texte annoté, puis cherche pour chacun des objets Word tous les objets correspondants dans la transcription. Dès qu'une partie gauche du mot en cours a été trouvée, l'Interval correspondant est ajouté aux liens du mot, puis ce qu'il en reste est analysé à partir de l'Interval suivant, jusqu'à ce qu'il n'en reste plus.

Dans la mesure où il s'agit d'un des travaux manuels qui a pris le plus de temps, j'ai également fourni un outil de diagnostic permettant d'identifier une erreur et d'en afficher son contexte, afin de rendre les corrections plus rapides. Ces corrections consistaient la plupart du temps à corriger des accents ou des erreurs de transcription, soit dans le texte annoté, soit dans la transcription.

Chapitre 8

Conclusion

Dans cette présentation on a proposé une systématisation informatique du système d'annotation du corpus Rhapsodie pour son exploitation par un parseur FRMG et son enrichissement de données prosodiques. Cette proposition allie des structures de données objet, adaptées au formalisme souhaité, et les différents algorithmes permettant de mettre en œuvre cette proposition [7].

Il a été vu que l'implémentation, de la totalité de l'information encodée dans le balisage, dans un multi-arbre (duquel on projette ensuite la structure souhaitée) s'avère nécessaire pour les traitements réalisés (dépliage notamment). L'efficacité du système d'annotation syntaxique de Rhapsodie a été démontré à condition d'opter pour une vision en multi-dimension pour son implémentation et sa pertinence quant à sa représentation arborée.

La vision en multi-arbre permet un plus grand nombre de manipulations et permet de généraliser l'ensemble des structures macro et micro-syntaxiques en une seule structure. Ce multi-arbre est facilement manipulable dès lors qu'on maîtrise les différentes dimensions qui le composent (macro, micro-syntaxe, prosodie etc.). Une perspective intéressante serait de travailler sur d'autres hiérarchies afin d'enrichir le multi-arbre d'autre niveaux linguistiques (sémantiques, pragmatiques etc.)

Il reste encore un grand nombre de points à éclaircir en particulier pour ce qui est de la représentation du multi-arbre, en effet la 2D ne semble pas suffisante pour permettre la bonne conceptualisation de ses multiples dimensions. Une autre perspective intéressante serait de convertir le multi-arbre en graphe afin d'en augmenter la souplesse et de rendre sa représentation plus élégante. En effet si en l'état les structures syntaxiques du multi-arbre restent lisibles, c'est parce que les les structures macro et micro sont fortement corrélées. L'introduction de nouvelles structures ou d'autres niveaux syntaxiques moins corrélés entre eux impliquerait de passer à une représentation en graphes.

On peut également souligner que l'utilisation d'un parseur ad-hoc et d'un formalisme objet aurait peut-être pu être éviter en recourant à des grammaires hors contextes (CFG), mais mon manque de recul par rapport aux structures syntaxiques souhaitées par les syntacticiens du projet m'ont fait prendre un tout autre chemin. On peut toutefois souligner que le formalisme objet s'est avéré fonctionnel.

Enfin peut-être faudrait-il explorer les possibilités offertes par les bases de données orientées objet pour ce type de corpus qui permettrait une plus grande optimisation de traitement des données et permettrait de se libérer de la rigidité inhérente aux formats XML.

La base de donnée de Rhapsodie repose sur des tables relationnelles remplies par un parsing des fichiers XML finaux. À mon avis il serait intéressant d'explorer les possibilités offertes par les bases de données de graphes qui est une technologie assez récente mais qui je pense peut s'avérer très prometteuse pour le stockage et l'exploitation des données linguistiques.

Le multi-arbre n'est pas généré en format XML dans le cadre du projet, il n'est utilisé que comme structure relais permettant l'ensemble des traitements nécessaires à la réalisation des tâches de dépliage et de projection. Le stockage du multi-arbre dans une base de données de graphe (OWL préférentiellement) aurait à mon avis permis un plus grande souplesse au niveau des requêtes possibles et aurait proposé un plus grande offre au niveau des structures interrogeables. En effet pour mes propres recherches j'ai encodé les informations syntaxiques et prosodiques dans le même arbre et l'ai ensuite directement exploité en requêtant grâce à des scripts Python.

Troisième partie

Étude Intono-syntaxique du phénomène d'entassement

Résumé

Dans cette partie je propose une étude intono-syntaxique mettant à profit les structures de données hiérarchiques syntaxiques et prosodiques implémentées dans le multi-arbre. Le multi-arbre est une structure de données regroupant l'ensemble des hiérarchies encodées dans les annotations syntaxiques et prosodiques du projet RHAPSODIE. Le principe d'un travail en intono-syntaxe est d'étudier certains phénomènes syntaxiques par le biais de leurs propriétés prosodiques et vice versa. On peut ainsi imaginer étudier la variation de f_0 en fonction de structures syntaxiques particulières. Le phénomène syntaxique étudié ici est l'entassement. L'entassement est une dimension orthogonale à la rection. Les couches constituant l'entassement peuvent réciproquement venir occuper la même position syntaxique : *on dit qu'un segment Y s'entasse sur un segment X si Y vient occuper la même position régie que X*. Par exemple si on considère l'unité illocutoire *"‿mais il y avait des gens comme { Gide | Claudel | Valéry | Malraux } //"*, les mots entassés *Gide*, *Claudel*, *Valéry* et *Malraux* sont tous substituables les uns aux autres car ils peuvent occuper la même place régie. Mes travaux se concentrent sur les segments constitutifs des entassements et leur caractéristiques prosodiques. L'unité minimale choisie est la syllabe et les critères prosodiques retenus pour l'étude sont les valeurs de f_0, les mesures normalisées d'allongement syllabique et l'annotation des proéminences. Je montre ainsi que l'allongement syllabique est plus important aux frontières droites des constituants de l'entassement et va souvent de pair avec une augmentation de la f_0. Enfin par des mesures différentielles des critères prosodiques retenus, je mets en évidence l'existence de transitions marquées entre les différents constituants des entassements.Cette étude est rendu possible par l'exploitation de structures objets présentées en partie II et par un ensemble de requêtes statistiques appliquées dessus.

"*Le discours foisonne de phénomènes d'entassement*"
BLANCHE-BENVENISTE 1990

Chapitre 9

L'entassement

9.1 Étude syntaxique du phénomène d'entassement

L'entassement tel qu'il a été analysé dans le projet Rhapsodie à déjà fait l'objet de nombreuses publications [50, 38, 74]. Un état de l'art précis a déjà été dressé par Paola Pietrandrea et Sylvain Kahane dans [74] et comme ils le font remarquer Claire Blanche-Benveniste est l'une des première linguiste à avoir évalué et étudié l'importance de ces constructions dans les productions du français oral spontané et avoir proposé un regroupement partant du principe que ces constructions relevaient du même procédé de listage ([16], [14]). La syntaxe de dépendance de Tesnière ([89]) a pour sa part caractérisé la coordination comme un procédé orthogonal à la dépendance pure (ce que Blanche-Benveniste qualifie de rection) et a nommé ce procédé la jonction. Le procédé est orthogonal dans la mesure les dépendances peuvent être vues comme verticales et perpendiculaires à la jonction[74]. Blanche-Benveniste procède selon le même principe en le renversant : les productions orales, c'est-à-dire le défilement syntagmatique étant en abscisse de l'énoncé et plaçant en ordonnées les listes paradigmatiques d'éléments pouvant occuper une même position syntaxique. C'est l'analyse en grille ([16], [14]).

Entassement

Les entassements([74], [73, 38, 50, 49]) font normalement partie de la micro-syntaxe, l'entassement, aussi appelé pile (voir [38][50][48]), est un dispositif de connexion syntaxique qui relie tous les éléments qui occupent la même position syntaxique à l'intérieur de l'UR. Pour l'annotation manuelle du corpus (voir les trois exemples ci-dessous), on a utilisé les symboles **{** et **}** pour marquer le début et la fin de la liste et **|** pour signaler le ou les points de jonction dans le prolongement des listes paradigmatiques et de l'analyse en grille proposées dans [14]. Les entassements ont des fonctions sémantiques et pragmatiques variées et peuvent contenir des éléments macro-syntaxiques : ils peuvent servir à signaler des disfluences, à établir des relations entre référents, à créer des nouveaux référents, à reformuler, à exemplifier, à préciser , à intensifier. Les conjoints ne sont pas nécessairement des constituants micro-syntaxiques mais peuvent être des disfluences par exemple :

Liste Paradigmatique

Analyse en Grille

^et **{ la | la }** Loire est en bas // (D0003 Corpus Rhapsodie)

$L1 "bon" **{ il | il }** y a vraiment "euh" aucun problème // (D0002 Corpus Rhapsodie)

[1]

{ ^et | ^et } donc < **{ elle | elle }** tombe // (M0023 Corpus Rhapsodie)

Dans le cadre du projet, l'entassement est considéré comme étant une dimension orthogonale à la rection : *"on dit qu'un segment Y s'entasse sur un segment X si Y vient occuper la même position régie que X. Nous notons { X / Y } le fait que X et Y sont entassés. Dans l'entassement { X / Y }, X et Y sont chacun une couche de l'entassement"*[74].

Un entassement peut avoir plusieurs couches ou davantage et s'organiser selon plusieurs profils topologiques (voir sec 9.4).

9.2 L'entassement : une globalisation de différents niveaux linguistiques

Notre étude se concentre donc particulièrement sur ce phénomène d'entassement. Cette dénomination a été introduite pour faire référence à un objet linguistique abstrait qui englobe un certains nombre d'autres phénomènes. Ces phénomènes se répartissent selon des niveaux différents de la structure syntaxique et ne sont pas étudiés ensemble habituellement. Un entassement peut-être globalement vu comme une multi-réalisation d'une même position syntaxique, plus précisément, chaque couche de l'entassement peut remplir la même position syntaxique que ces pairs.

Je vais mettre en évidence le fait que cet objet englobe un certains nombre d'autres phénomènes identifiables au niveau lexical, dont voici quelques exemples :

9.2.1 La répétition et la disfluence

Répétition

La répétition ([45], [85], [86] et [87]), qui implique la répétition d'un mot ou d'une partie de mot. Cet effet peut avoir une grande variété de portée pragmatique. La réduplication est considérée dans le cadre du projet seulement si l'on considère la rection.

La disfluence est un cas particulier de réduplication, il s'agit également d'une répétition d'un segment (avec éventuellement des ajustements morphologiques), qui peut soit être immédiate soit se produire à distance après une interruption. Cette répétition résulte d'une disfluence de la part d'un locuteur :

Disfluence

et vous appartenez à ce qu'on appelle les flying doctors de la Mref > qui est **{ une | une }** association { des médecins | ^et **{ de | de | de }** recherche } qui est basée { ici | **{ à | à }** Nairobi | au Kenya } // (D2004 Corpus Rhapsodie)

9.2.2 La co-composition

Co-composition

Toujours dans le mesure où l'on peut parler de rection syntaxique, la co-composition ([91]), qui est essentiellement un composé dont le sens est le résultat de la coordination

1. Les exemples sont extraits du corpus annoté dans le cadre du projet Rhapsodie, un corpus de 3 heures, d'environ 36 000 mots, de français oral spontané, et pour lequel les entassements ont été systématiquement annotés (Benzitoun et al. 2009, 2010).

Les numéros à la suite des exemples, comme D0005 ou M2006 (D pour dialogue, M pour monologue), indiquent l'échantillon dont l'exemple est extrait ; ces échantillons durent de 2 à 5 minutes.

des significations de ses composants, c'est à dire un mot-unité constitué par deux parties ou plus qui expriment la coordination physique :

une vitrine de **pâtissier-boulanger** (M0002 Corpus Rhapsodie)

^et ^puis j'avais toujours présent à l'esprit Lewis Carroll **mathématicien-écrivain** // (D2005 Corpus Rhapsodie)

9.2.3 Les binômes irréversibles

Les binômes irréversibles ([59], [56] et [61]) dans le cas où un lien de rection est constaté : Binôme Irréversible

"noir et blanc"

"bigger and better"

"back and forth"

mais on n'inclut pas dans les entassements les binômes irréversibles dont l'un des constituant serait non régit :

*On se donne **corps et âme** à cette cause.

9.3 La Typologie des entassements selon Pietrandrea et Kahane

Ce modèle caractérise les coordinations identifiables au niveau syntaxique ([1], [44] et [63]) : Coordination

"Le Président apprécie et approuve votre proposition." ("In defense of lexical Coordination", Anne Abeillé)

"My uncle or your in-laws or the neighbors will come to visit us." ("Coordination" Martin Haspelmath)

Pietrandrea et Kahane ainsi que Bonvino et al. [74, 20]ont établi une distinction des différents types de coordinations dans laquelle ils distinguent deux grandes classes :

- "*les coordinations ou **entassements de re** qui construisent une expression dont la dénotation diffère de la dénotation des conjoints ; autrement dit, la coordination est une construction (et donc un signe linguistique) qui possède une contribution sémantique réelle*"[74]
- "***les entassements de dicto** qui établissent des relations entre formulations ; soit les conjoints n'ont pas réellement de dénotation (disfluence), soit les conjoints dénotent la même chose de différentes façons ; l'entassement de dicto n'a donc pas de contribution sémantique et il n'est pas un signe linguistique, mais une configuration structurale.*"[74]

9.3.1 Les entassements dit de re

Les coordinations compositionnelles

coordination compositionnelle

Les coordinations simples permettent de relier au moyen de joncteurs (et, ou, ni, mais, etc.) des constituants de catégories variées. D'un point de vue sémantique, la coordination construit de manière prototypique une disjonction ou une conjonction de propositions. Des interprétations non propositionnelles sont toutefois possibles dans le cas de la conjonction. D'un point de vue discursif, on observe que les coordinations simples sont compatibles avec tous les statuts informationnels reconnus : focus, fond, contraste. [20][62][74]

Coordinations additives

Coordinations additives Les coordinations dites additives est un entassement qui réunit plusieurs conjoint en un dénotation commune. Ce type de coordination est généralement caractérisée par un joncteur conjonctif entre ses constituants comme et, puis, ainsi que, ni, mais etc :

je travaille à la préfecture de Paris //+ qui **{ n'est pas connue | ^mais néanmoins existe }** "euh" // (D0001 Corpus Rhapsodie)

vous allez vous diriger vers le fond des jardins de ville **{ en restant "euh" &** | **{ en | en } y allant | ^mais en restant { s~ | sur } la gauche }** // (D0008 Corpus Rhapsodie)

{ j~ | je } connaissais des gens dans le septième qui mettaient leurs enfants **{ d'abord dans le public | ^puis dans le privé }** (D2002 Corpus Rhapsodie)

il s'expose directement comme **{ concepteur | ^et président }** d'une banque internationale pour la reconstruction d' { un monde qui vacille | **{ la Russie | ^et le bloc de l'est }** } // (D2005 Corpus Rhapsodie)

Coordinations alternatives

Coordinations alternatives Les coordinations alternatives dénotent souvent d'un choix potentiel entre les constituants de l'entassement. Elles marquent que les éléments sont potentiellement substituables les uns aux autres [63]. Les joncteurs disjonctifs les marquant sont : ou, soit, ou bien etc.

votre marché < c'est { Aligre | ^plus ^que Nation } // (D0001 Corpus Rhapsodie)

Coordinations non relationnelles

Coordination non relationnelle

ou non compositionnelles

Toujours selon PIETRANDREA et KAHANE, les coordinations non relationnelles sont celles qui ne servent pas à signaler la relation entre les conjoints, mais à créer un référent[20, 74].

Ce référent entretient alors une relation sémantique avec les conjoints. Ce sont les coordinations dite non-compositionnelles. On distingue plusieurs stratégies linguistiques pour pour créer un référent en relation sémantique avec les conjoints.

^mais je trouverai pas de livre sur **{ Ségolène Royal | ^ou Nicolas Sarkozy }** // (D2002 Corpus Rhapsodie)

Dans cet exemple tiré de [74], *"Ségolène Royal ou Nicolas Sarkozy"* ne renvoie pas à une alternative entre ces deux seules personnes politiques, mais à n'importe quelle personne politique (c'est en quelque sorte une généralisation) comme le montre la reprise de l'interlocuteur par les livres politiques.

La juxtaposition qui n'est pas toujours reconnue comme une coordination à part entière. Elles sont analysée comme telles car elles ont des propriétés communes avec la coordination : elles permettent de superposer un nombre non limité de termes de catégories variées et observent des contraintes de parallélisme. On a les sous-types suivants :

Juxtaposition

Coordination hyperonymique L'entassement coordonne des co-hyponymes et sert à :

Co-hyponyme

créer un référent en relation sémantique avec les conjoints

les gens < au début <+ quand ils voyaient **{ un ordinateur | une souris "euh" | tout ça }** <+ { ils sav~ | ils savaient } pas ce que c' était // (D0005 Corpus Rhapsodie)

l'approximation d'un synonyme

Synonyme

mais "euh" "euh" { { { se | se | se } gourer | ^et ^puis chauffer comme ça } | ^c'est-à-dire **{ dragouiller la mère | ^ou draguer | ^ou faire une déclaration d'amour à la mère }** } < non // (D2007 Corpus Rhapsodie)

la création d'un hyperonyme

Hyperonyme

il avait **{ un côté aristocrate | un côté dandy | un côté esthète }** qu'il gardera jusqu'à la fin de sa vie // (D2012 Corpus Rhapsodie)

Coordination intensive L'entassement coordonne également des éléments lexicalement identiques et sert à marquer une intensificationde leurs propriétés (fréquence, durée, intensité, continuité aspectuelle etc.)

on pouvait pas s'empêcher à la fin de { Mort | ^et transfiguration } de faire **{ résonner | résonner | ^et résonner | ^et encore }** ces accords qui nous enchantaient // (D2012 Corpus Rhapsodie)

9.3.2 Les entassement dits de dicto ou les listes dialogiques

Intensification

On répertorie aussi sous l'étiquette d'entassement, un certain nombre de phénomènes qui ont la plupart du temps été étudiés dans le cadre des études du discours, comme les listes dialogiques ([46] et [81]) :

Liste Dialogique

"***$L1*** c'était une enfance **{ heureuse |}** //+ ***$L2*** oui //+ **{| très heureuse }** //" (D2005 Corpus Rhapsodie)

Répétition

Les répétitions ([88]) et les reformulations de la langue parlée ([15] et [58]) :

Reformulation

"$L3 **{ très sale //+ | $L1 { très | très } sale }** //" (D2011 Corpus Rhapsodie)

""euh" **{ deux petites phrases | deux vraies options }** qui dessinent **{ votre route //+ | une route qui témoigne { d'une certaine | d'une bonne | d'une très bonne } conduite }** //" (D2001 Corpus Rhapsodie)

Les disfluences

Comme le rappellent Pietrandrea et Kahane[74], la disfluence a été déjà très bien décrite et analysée comme un type d'entassement par Blanche-Benveniste [15, 17, 14]. La disfluence est un concept flou qui regroupe les trébuchements (volontaires ou pas) du locuteur sur une position syntaxique afin d'ajuster sa formulation. Cela se traduit par un entassement de mots ou d'amorces de mots :

{ c'est | c'est | c'est } surtout l'hôpital public qui m'attire // (D0006 Corpus Rhapsodie)

^parce ^que j'ai **{ démé~ | déménagé }** neuf fois dans ma vie // (D1003 Corpus Rhapsodie)

{ je | je } longe la Seine jusqu' à la passerelle de Solférino // (D0001 Corpus Rhapsodie)

Les reformulations

reformulation

Les entassements où le deuxième conjoint n'introduit pas un nouveau référent, mais une nouvelle dénomination du conjoint précédent. La reformulation introduit donc une relation entre références.

Il existe un grand nombre de sous-types de reformulations :

Relation entre Référents

La reformulation lexicale ou dénotative Dans la reformulation lexicale, le locuteur entasse plusieurs dénotations du même référent en modifiant son lexique au fur et à mesure.

Reformulation lexicale

c'est une conférence de philosophie // mais c'est aussi **{ une conférence { de | d' } histoire de l'art | une conférence d'esthétique }** // (M2002 Corpus Rhapsodie)

et puis après <+ "ben" j' ai travaillé sur **{ les micro-processeurs | l' informatique }** // (D0005 Corpus Rhapsodie)

^et avant d' arriver au troisième <+ vous prenez à droite là où il y a **{ les intersions | les intersections }** // (D0017 Corpus Rhapsodie)

Reformulation prédicative

La reformulation prédicative Certaines reformulations remplissent en même temps la fonction de reformuler un référent et de prédiquer quelque chose à propos de ce référent. Il s'agit d'entassements dont le deuxième conjoint est parenthétique .

Parenthétique

^et j'ai trouvé **{ cet endroit | (Olkaloo) }** où ils avaient besoin d'un médecin //(D2004 Corpus Rhapsodie)

9.3.3 La négociation

Négociation

Toujours d'après Pietrandrea et Kahane [74], l'entassement constitue pour les locuteurs un cadre idéal à la négociation ([46, 81]). Ainsi on observe souvent des entassements que l'on pourrait qualifier de "co-construits" dans lesquels les locuteurs peuvent négocier entre eux la construction du discours ou encore les formulations introduites dans le discours.

Il y a quatre types de négociations qui ont été relevées dans le corpus Rhapsodie. Ces quatre opérations de négociation opérant sur les entassements sont d'après [74] : la demande de confirmation, la confirmation, la réfutation, la correction.

La demande de confirmation

Demande de confirmation

Ici on note que le locuteur L2 demande confirmation, attend une appréciation du spécialiste (dans le cas présent un tapissier-rempailleur) au sujet de la couleur des coussins.

$L2 mon tissu là de coussins est quand même "euh" -$ **{ foncé | rouge |}** //+
$L1 **{| "ouais" genre madras un peu }** // (D0009 Corpus Rhapsodie)

Confirmation

La confirmation

Dans l'exemple ci-dessous, le locuteur L2 vient confirmer une affirmation du locuteur L3.

$L2 moi < { j'étais donc "euh" | j'étais } à l'école rue de Picpus //+
$L3 **{ qui existe toujours |}** //+
$L2 **{| qui existe toujours }** > absolument // (D0001 Corpus Rhapsodie)

Réfutation

La réfutation

Dans cet exemple-ci, le locuteur L2 réfute sa première formulation "des français", le marqueur discursif "enfin" sert à initier la réfutation.

$L2 c'est la crise générale **{ { des |** ($L1 "oui" //) **des } Français //+ |}**
$L1 oui //+
$L2 "oui" **{| "enfin" des Français |** ($L1 "mh" "mh" //) **pas simplement des Français "hein" | { { des | de } l'humanité | ^et de la lecture } }** // (D0004 Corpus Rhapsodie)

Correction

La correction

On note dans cet exemple une intervention du locuteur L3 pour corriger le nom du joueur de football qu'énonce le locuteur L2.

c'est peut-être au départ $- > alors "hein" //
$L2 l'arbitre -$ revient à la première faute avec un avertissement (effectivement) pour $- **{ Diarra |}** //+
$L3 **{| Gallas }** // -$ (D2003 Corpus Rhapsodie)

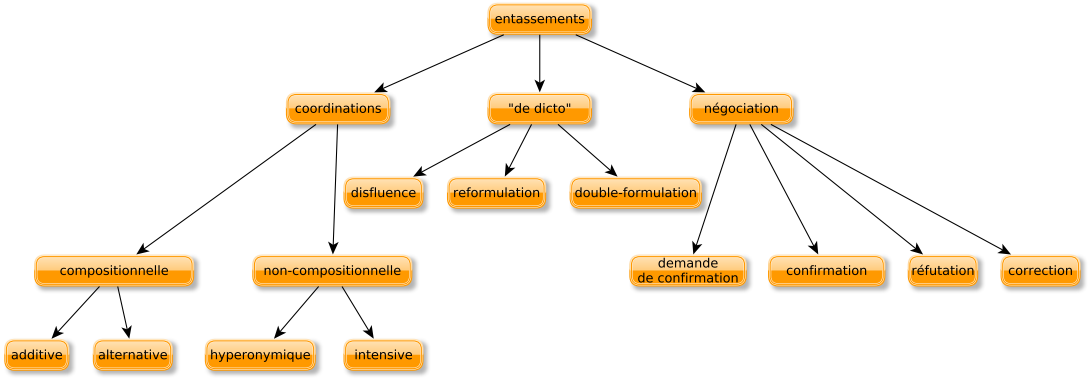


FIGURE 9.3.1 – Typologie des entassements selon PIETRANDREA et KAHANE [74][74]

9.4 Topologie des entassements

Les entassement peuvent être distingués selon leur niveau et portée linguistique (disfluences, reformulation, coordination, etc.) mais aussi selon leur topologie. On entend par topologie la façon dont les entassements s'organisent dans le texte. On distingue deux grands types d'entassements dans lesquels se répartissent plusieurs sous-groupes que l'on différencie en fonction de leurs héritages.

Topologie des Entassements

9.4.1 Quelques statistiques sur le corpus

Le corpus Rhapsodie compte environ 33000 mots, on recense un total de 3292 unités illocutoires (UI) dont 1071 comporte des entassements. Les UI qui comportent des entassement comptent en moyenne 30 syllabes d'environ 0.209 seconde chacune.

Statistiques sur la topologie des entassements (continus et discontinus confondus)

Type d'entassement	Nb d'occurrences	Nb moyen de syllabes	Durée moyenne des syllabes (s)
unique	1383	8.2	0.223
parent	993	8.2	0.219
enfant	389	8.0	0.235

FIGURE 9.4.1 – Table de la distribution topologique des entassements

Statistiques en fonction du nombre de couches des entassements

Nb de couches	Nb moyen de syllabes	Durée moyenne des syllabes (s)
2	3.9	0.154
3	4.3	0.222
4	4.8	0.211
5	4.8	0.279

FIGURE 9.4.2 – Table des valeurs syllabiques en fonction du nombre de couches dans un entassement

5

9.4.2 Les entassements continus

Les entassements continus uniques

il n'y a que les hommes **{ qui travaillent avec moi | ^ou qui ont travaillé avec moi }** qui pourraient vous répondre // (D2001 Corpus Rhapsodie)

Les entassements continus parents

"euh" { deux petites phrases | deux vraies options } qui dessinent **{ votre route //+ | une route qui témoigne** { d'une certaine | d'une bonne | d'une très bonne } **conduite }** // (D2001 - Corpus Rhapsodie)

L'entassement en gras contient un entassement enfant : { d'une certaine | d'une bonne | d'une très bonne } , c'est donc un entassement parent.

Les entassements continus enfants

"euh" { deux petites phrases | deux vraies options } qui dessinent { votre route //+ | une route qui témoigne **{ d'une certaine | d'une bonne | d'une très bonne }** conduite } // (D2001 Corpus Rhapsodie)

Ici l'entassement est enfant de la deuxième couche du deuxième entassement principal.

Les entassements continus enfants et parents

{ pour dire quoi |} //+ {| pour principalement (pour résumer assez vite là) s'opposer à { cette idée de l'art | cette idée qu'on a déjà vue "hein" ensemble "euh" de l~ | cette idée { qui s'est largement développée au dix-neuvième siècle romantique | ^et ^puis qui a continué par la suite } | cette idée que l'art est du côté **{ de la souffrance | du malheur** | { de | des } **tourments intérieurs }** de l'artiste qui doivent exploser comme ça dans la société qui de toutes façons est contre "euh" } } (D1002 Corpus Rhapsodie)

Ici l'entassement en gras, est continu, parent de l'entassement { de | des } et enfant de l'entassement principal.

9.4.3 Les entassements discontinus

Les entassements discontinus uniques

je dirais que vous avez donné **{ quelque chose de plus |}** à la femme //+ **{| des armes de persuasion }** // (D2001 Corpus Rhapsodie)

Les entassements discontinus parents

^et ^puis après <+ donc < "euh" j'étais **{ une année à la classe de philo au lycée Paul Valéry |}** //+
($L1 mh mh //)
$L2 "euh" **{| ^et en prépa évidemment "euh"** | { à | à } **la sor~ | à** { Censier | ^puis la Sorbonne } **"quoi" }** // (D0001 Corpus Rhapsodie)

L'entassement ci-dessus est discontinu et également parent de deux entassements : { à | à } et { Censier | ^puis la Sorbonne }

Les entassements discontinus enfants

$L1 donc < des jeunes $- $L1 couples qui s'installent < { dès qu'il y a un enfant |
($L2 XXX pas tellement d'enfants // $L1 voilà // $L2 voilà //) -$
$L1 dès qu'il y a **{ un |}** enfant **{| ^ou deux | au moins deux }** | <+ le d~ <+ quand le deuxième } arrive <+ ils partent // (D0004 Corpus Rhapsodie)

L'entassement en gras est inclus dans l'entassement parent qui commence à *{ dès qu'il y a un enfant* et termine à *quand le deuxième }* et est par ailleurs discontinu au sein de son parent.

Les entassements discontinus enfants et parents

Type d'entassement non trouvé dans le corpus, mais bien que la configuration soit plus complexe que les autres, rien ne l'empêche conceptuellement...

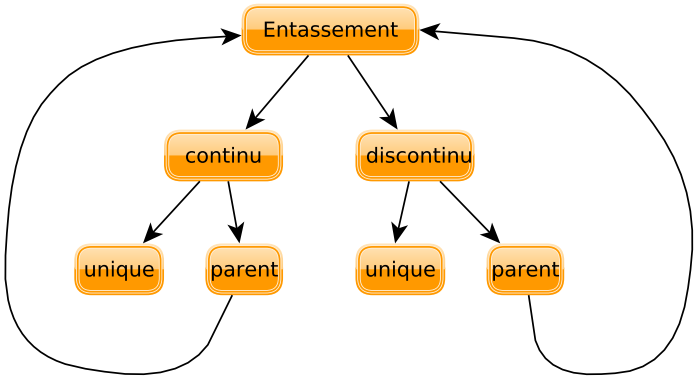


FIGURE 9.4.3 – Topologie des entassements

9.5 Conclusion

Grammaires de Construction

Ces phénomènes ont la plupart du temps été étudiés séparément par les différentes traditions linguistiques parce qu'ils appartiennent à différents niveaux d'analyse : lexical, syntaxique, dialogique etc. On peut rappeler de plus que que les entassements ont des fonctions très différentes et que par là certains sont des signes et d'autres pas. Les grammaires de construction qui proposent comme unité fondamentale du langage la construction, ou l'association entre la forme et le sens prend des constructions, plutôt que les morphèmes, mots ou phrases, comme unités de base de l'analyse linguistique permet une vision plus globale des phénomènes. Les constructions sont définies comme des associations stylisées d'une forme et d'une fonction. Cette définition présente l'avantage d'englober presque toutes les unités de sens du langage, allant des mots à des unités plus complexes et abstraites de phrase dans différents niveaux de structures. On peut évoquer la structure des constructions argument ([39]) ou des types de phrases ([66]).

Ainsi en adoptant cette approche de la construction, les syntacticiens de Rhapsodie ont tenté d'établir une description unifiée de l'ensemble des phénomènes linguistiques décris ci-dessus.

Ces phénomènes sont alors définis comme "héritiers" d'une construction abstraite d'exploitation à travers les différents niveaux linguistiques : à savoir la construction d'entassement. Leur objectif principal étant de montrer que les constructions d'entassement sont des constructions dans le sens technique de la grammaire de construction ([34], [51], [39] et [62]).

Chapitre 10

L'entassement marqué prosodiquement ?

10.1 Introduction

Je présente ici une série de trois études, respectivement menées sur les coordinations du Français par François Mouret, sur le phénomène de liste en Allemand standard par Margret Selting qui étudient les variations de pitch c'est-à-dire de f_0 et enfin une étude de Caroline Féry qui compare la prosodie des coordinations enchâssées en Allemand et en Hindi en se basant principalement sur les variations de durée et de f_0.

Ces travaux ont attirés mon attention en particulier pour les sujets qu'ils traitent qui sont tous inclus par le phénomène d'entassement, pour leur mode opératoire et les paramètres étudiés.

10.2 Études sur différents types de coordinations du français

Mouret et al. [28] ont réalisé une étude sur trois types de constructions coordonnées du français. Pour eux, s'il semble y avoir des arguments syntaxiques et sémantiques solides pour distinguer trois types de constructions coordonnées en français, l'existence d'arguments prosodiques corroborant cette tripartition est moins évident. Leurs résultats semblent tendre vers une absence d'appariement univoque entre un type de coordination et un type spécifique de réalisation prosodique (une prosodie qui serait propre aux différents types d'entassement comme la juxtaposition, le redoublement, l'intensification etc.). En revanche, il semblerait que chaque type présente un comportement différent des autres dans les dimensions qu'ils sont parvenus à isoler.

Pour le cas des coordinations à conjonction redoublée. Il a été observé que chaque conjoint, c'est-à-dire chaque couche de l'entassement appartiendrait presque systématiquement à un groupe prosodique autonome. La proéminence mélodique étant le plus souvent réalisée à l'initiale de chacune des couches de l'entassement considéré, c'est-à-dire pour le cas étudié sur la conjonction.

Mouret et al. [28] considèrent que ces deux traits prosodiques qui ont été mis en avant peuvent être analysés comme le réflexe d'une frontière phonologique de groupe intonatif alignée sur la frontière gauche de chaque conjoint. Plusieurs arguments sont proposés, en premier l'argument concernant l'impossibilité de l'enchaînement consonantique entre la couche initiale et ce qui la précède ainsi qu'entre chacune des couches de l'entassement considéré. Enfin l'argument concernant la proéminence mélodique qui serait réalisée à l'initiale de chacune des strates et qui aurait des caractéristiques particulières qui la

distinguent des autres accents initiaux (accent secondaire, accent d'emphase ou accent d'insistance, cf. Astesano [2]). En effet la proéminence mélodique serait réalisée à l'initial absolue et sa cible haute se propagerait généralement sur l'attaque de la syllabe qui suit.

En revanche les résultats obtenus avec les entassements de juxtapositions, comme par exemple les coordinations simples, présentent une plus grande variété : les strates de ce type d'entassements peuvent être traitées comme des groupes intonatifs ayant les mêmes proéminences initiales que celles observées dans les coordinations redoublées ou bien comme de simples groupes accentuels. Ils établissent cependant une distinction entre juxtapositions et coordinations simples grâce à une nette préférence pour le schéma mélodique avec mouvement montant en finale pour les strates positionnées avant le noyau verbal. Pour faire un parallèle avec le système syntaxique utilisé pour Rhapsodie, on pourrait donc émettre l'hypothèse d'après ces résultats que les entassements constituées d'UR pré-noyaux auraient des couches caractérisées par un contour montant en finale.

Cette étude apporte des résultats fondamentaux dans la mesure où il semble avéré que chaque type de coordination se distingue prosodiquement :
- *"les conjonctions redoublées appellent un phrasé où chaque conjoint porte une frontière gauche de groupe intonatif (GI)"*
- *"les juxtapositions et les coordinations simples ne déterminent pas un phrasé unique : chaque conjoint peut correspondre à un GI, un groupe accentuel (GA) ou être inclus dans un GA avec le matériel à gauche ou à droite*"
- *"les juxtapositions se distinguent des coordinations simples dans le choix des mouvements mélodiques marquant leur constituance interne."*

La conclusion de ces travaux est particulièrement forte dans la mesure où il semble avéré, notamment pour les juxtaposition que le patron mélodique varie pour marquer les constituants (couches) de l'entassement.

10.3 Étude du phénomène de liste en allemand standard

Margret Selting a travaillé sur des données d'allemand spontané et établit un description détaillée de la prosodie des listes. Elle a en particulier montré que la prosodie des listes représente dans le cas de l'allemand un repère constitutif dans la signalisation de listes en interaction[81]. Ainsi lors de son travail sur la séquentialité et la prosodie des listes, Selting (2007) dégage une structure englobante et tri-partite, dans laquelle est intégrée la liste :
- une partie "projection" (projecting component) qui marque que plus va être dit. Ce peut être une formulation générale qui projette que vont suivre des élaborations, explications, exemplifications ou illustrations ;
- la liste proprement dite, qui détaille ce que la projection a annoncé et qui consiste en une série d'items présentant une syntaxe et une prosodie au moins partiellement analogues ;
- le "composant post-liste" (post-detailing component), faisant le plus souvent le lien avec la partie projective, et présentant généralement une intonation descendante.

Les composantes de projection et postérieure sont utilisées pour l'enrobage et la contextualisation de la liste à proprement parler.

Elle montre qu'on ne peut pas identifier une unique prosodie de liste, mais que, par-delà la variété observée, on retrouve le parallélisme des deux premiers items (au moins), créé par la répétition du patron prosodique, une même intensité et une même durée, s'ajoutant le plus souvent à une même place syntaxique. Des faits séquentiels et prosodiques permettent de distinguer les listes fermées des listes ouvertes.

Les listes fermées sont celles pour lesquelles la prosodie suggère notamment des éléments de fermeture. Ils peuvent être signalés par l'intermédiaire de la prosodie ou avant la

projection du nombre d'éléments de liste de suivre. Les listes fermées produites en une seule unité de construction du tour (UCT) sont formulées le plus souvent dans des phrases simples semblent être caractérisées par des pics de hauteurs descendants "down-stepped pics" pour chaque couche de la liste (cf études ultérieures d'exemples de listes lues à haute voix pour lesquelles ils ont trouvé un down-stepping [5, 29]).

Les listes ouvertes sont celles pour lesquelles la prosodie suggère un nombre ouvert d'éléments et pour lesquelles certains contours d'intonation particuliers sont utilisés plus souvent que d'autres.

Dans ses données, Selting note qu'un décalage de la ligne de déclinaison est le plus souvent atteint après un contour qu'elle appelle "la hausse escalier" avec soit un plateau de haut niveau, soit pas de haut niveau final ou alors une hauteur finale en légère baisse. Elle observe ce schéma pour 67% de ses données. Pour 20% des données elle relève une augmentation constante hauteur de la syllabe accentuée à la fin des couches constituants la liste. Le reste des données semblent être caractérisées par un pitch de niveau moyen ou bas.

La conclusion de cette étude, est que ce qui caractérise l'intonation du phénomène de liste pour des données d'allemand standard est aussi bien :
– la présence d'un (hat-pattern) plateau haut et plutôt continu
– qu'un plateau intonatif avec seulement une légère baisse ou montée finale du pitch sur la syllabe accentuée de fin de layer.
Même si cela est moins net que pour l'étude de Mouret [28] et que l'étude porte sur l'Allemand qui est une langue à accent mélodique, l'entassement semble déclencher une variation du patron mélodique.

10.4 Étude sur la coordination et la coordination enchâssée en allemand et en hindi

Dans cette étude Caroline Féry [31] se focalise sur l'allemand et l'hindi. L'allemand est une langue intonative à accents mélodiques dans laquelle le pitch et les accents de frontière (tons) sont utilisé afin d'exprimer l'information pragmatique ou sémantique. Il a été démontré que les système intonatifs des langues intonatives est également utilisé pour indiquer et analyser la structure syntaxique des énoncés (elle cite notamment [21, 35, 80]). Pour l'hindi, en revanche n'a ni accent tonique ou de hauteur ni tons et serait plutôt une "langue à accent de phrase" (cf [30, 72]).

La prosodie des phrase étant principalement déterminée par des patrons intonatifs plutôt invariants. Par rapport à l'allemand le système intonatif de l'hindi est apparemment beaucoup plus rigide.

Caroline Féry a choisi d'étudier l'interaction syntaxe-prosodie pour ces deux langues dans le cadre de coordinations ambiguës. Le choix des coordinations ambiguës m'a intéressée dans la mesure où mon corpus d'étude est constitué d'entassements qui englobent toutes les constructions en liste détaillées en section 9.2 et ne se focalise donc pas sur une construction particulière.

Les résultats pour l'Allemand mettent très clairement en avant la relations entre structure syntaxique et structure prosodique. On voit émerger de la structure syntaxique les principes de proximité et de similarité identifiés par les variations de la structure prosodique associée.
– Principe de proximité : au sein d'un constituant C_a, réalisation d'un élément x selon un schéma prosodique proche de l'élément x_1 suivant.
– Principe d'anti-proximité : réalisation de l'élément x d'un constituant C_a selon un schéma prosodique différent de l'élément$y$ suivant appartenant à un constituant C_b.

L'anti-proximité renforce la frontière entre deux constituants par allongement du membre de droite des groupements.
– Principe de similarité : les constituants x' inclus dans un constituant x présentent le même type de réalisation prosodique que le constituant x.

Le principe de proximité se traduit par l'observation d'un abaissement du pitch et des durées plus courtes sur le constituant de gauche des entassements. La réciproque soit l'anti-proximité exerce l'effet opposé et a tendance à renforce les frontières entres les couches de l'entassement. On observe également un ralentissement des durées sur les constituants situés à droite de l'entassement (soit sur les couches finales).

Le principe de similarité est observé lorsque des constituants dits "simples" imbriqués dans un constituant plus grand dit "complexe" présentent un allongement et un rehaussement du pitch pour se calquer sur le même type de contour prosodique que le constituant dans lequel ils sont imbriqués.

Caroline Féry observe que l'allemand utilise la prosodie avec précision en vue d'interpréter et de correspondre à la structure syntaxique. Elle note que cette caractéristique de l'allemand est corrélée avec son système intonatif général (l'allemand est une langue à accent mélodique). Les résultats de son étude montre que la prosodie est donc un support fidèle de la structure syntaxique (cf une précédente étude de l'auteur [32]).

Pour l'hindi, en revanche, les résultats obtenus pour cette étude montrent une absence de corrélation entre la structure syntaxique et la structure prosodique, une autre étude de l'auteur présente des résultats similaires ([33]). Aucun des principes présentés plus haut et observés pour l'allemand ne sont vérifiés. On peut manifestement expliquer ces résultats en tenant compte des propriétés inhérentes au système intonatif de l'hindi qui organise sa prosodie selon la répartition des "phrasal tones".

La conclusion de ces résultats est que la corrélation entre structure syntaxique et structure prosodique ne semble pas être une propriété universelle lorsque l'on considère le paramètre de durée.

10.5 À la recherche d'unité d'entassement prosodiquement marquées

Selon Simon et Grobet, le phénomène de la déclinaison tonale, associé à la réinitialisation que j'essaye de mettre en évidence pour cette étude soulève un certains nombre de problèmes et débats [41]. Dans la littérature la déclinaison tonale se définit comme un *"abaissement progressif de la fréquence fondamentale du début à la fin d'un énoncé"* [53]. Selon Ladd [55], plusieurs approches permettent d'étudier la déclinaison. L'approche statistique qui permet d'observer qu'il y a plus de contours intonatifs descendants que de contours qui ne le sont pas. C'est cette approche qui est adoptée dans le cadre de l'étude présentée ici. Enfin l'étude phonologique qui traite la déclinaison comme une modification systématique qui a lieu au cours de l'énonciation et dans un cadre de référence phonétique.

Cependant, comme le rappellent Simon et Grobet [84], seule l'approche phonologique propose un modèle explicatif de la déclinaison. [1]

Il est en général postulé que le mécanisme automatique de déclinaison tonale s'explique par des causes physiologiques inhérentes aux différents organes de la phonation. Pour ce qui est de la déclinaison de f_0 elle est liée soit à la respiration (la baisse de la pression sous-glottique), soit à l'activité laryngienne, soit à une interaction des deux composantes.

1. Ladd reconnaît qu'il n'y a pas d'argument décisif pour attribuer à la déclinaison une pertinence : par exemple, comment être sûr que la chute de F0 qui caractérise souvent la fin d'un énoncé est due à la déclinaison tonale plutôt qu'à une loi phonologique d'abaissement (final lowering) qui s'appliquerait dans certains environnements [55, 84]

Simon et Grobet rappellent que d'autres auteurs voient aussi la déclinaison comme un résultat d'optimisation en vue d'un moindre effort de la part du locuteur (voir aussi [55, 53]). Enfin comme le signalent un certains nombre d'auteurs, la ligne de déclinaison constitue aussi potentiellement un principe efficace pour segmenter le discours. C'est sur ce principe de segmentation que je base mon étude.

Ainsi je propose en recourant au phénomène de réinitialisation de la ligne de base de f_0 et l'étude des durées de vérifier cette propriété segmentante. L'étude de la distribution des proéminences viendra compléter ces travaux y apportant des paramètres perceptifs.

Chapitre 11

Méthodologie

11.1 Introduction

La plupart des études intono-syntaxiques existantes se concentrent sur les phénomènes de listes, juxtapositions, coordinations etc. Le paramètre prosodique qu'elles considèrent est souvent unique et constitué par le pitch. S'il est, certes, un élément constitutif de la prosodie, il n'en est qu'un aspect. D'autres études se concentrent sur la durée, mais rarement les deux paramètres conjoints [31]. Dans ce travail, j'étudie la prosodie des entassements, à la lumière de trois critères prosodiques différents. Je me focaliserai ainsi sur deux familles de valeurs : des valeurs réelles (la durée et la valeur moyenne de f_0) et des valeurs catégorielles (l'annotation perceptive des proéminences). L'ensemble des ressources qui ont été nécessaires à cette étude est téléchargeable librement sur le site du projet **http ://www.projet-rhapsodie.fr/**.

Recueil des données

De manière à procéder à une étude systématique des caractéristiques prosodiques des couches entassées et de leurs transitions, une première étape est le recueil de ces données. Je commencerai ainsi par expliquer en section 11.2 comment les données prosodiques de chaque syllabe contenue dans une couche ont été recueillies en utilisant le formalisme présente en Partie II. Ce dépouillement des données se fait en deux temps. Tout d'abord, on extrait l'ensemble des critères prosodiques (f_0, ralentissement, proéminences, etc.) pour chaque syllabe de chaque couche de chaque entassement de la base, puis on procède dans un deuxième temps à plusieurs regroupements de ces données. Le premier type de regroupement considéré permet d'étudier les transitions entre couches, tandis que le deuxième type de regroupement permet d'étudier le comportement des critères prosodiques au sein de chaque couche.

Étude inter-couches

La caractérisation prosodique des transitions entre unités syntaxiques a déjà fait l'objet de plusieurs travaux qui serviront de guides pour les tests à effectuer. Ainsi, plusieurs études [31, 81, 28] se focalisent sur la ligne mélodique des phénomènes de listes et mettent en évidence la présence de ruptures au niveau des frontières des éléments constitutifs de la liste. D'une manière analogue, je montrerai la présence d'un phénomène de rupture de la ligne mélodique à la frontière entre deux couches d'un entassement. De la même manière, je me concentrerai sur d'autres critères prosodiques tels que le ralentissement et les proéminences, vérifiant ainsi la présence (non nécessaire) d'une pause [92] entre les couches. La méthodologie particulière pour cette étude des transitions entre couches est présentée en section 11.3.

Étude intra-couche

La deuxième partie de l'étude concerne la recherche au sein des couches d'un éventuel patron prosodique particulier. Afin d'identifier ces patrons, j'ai retenu deux critères : le pitch [77, 64, 67] et le ralentissement [36]. La méthodologie de cette étude est détaillée en section 11.2.

11.2 Recueil des données intono-syntaxiques

Pour mener à bien cette étude, j'ai extrait de l'ensemble des échantillons de la base Rhapsodie les informations relatives aux entassements et aux paramètres prosodiques à étudier. Plus précisément, on cherche à étudier les propriétés prosodiques des syllabes appartenant à des piles, et à les mettre en perspective par rapport aux propriétés de toutes les syllabes en général. Ainsi, pour toutes les syllabes du corpus, j'ai exporté un certain nombre de caractéristiques prosodiques, regroupées dans la figure 11.2.1.

donnée prosodique
f_0 (pitch)
ralentissement
libellé de la syllabe
proéminence de la syllabe
hésitation de la syllabe
forme de la syllabe
registre local de la syllabe

FIGURE 11.2.1 – Tables des données prosodiques étudiées

En plus de ces données prosodiques relatives à l'ensemble des syllabes sans distinction, il est nécessaire d'exporter des informations spécifiques pour les syllabes qui appartiennent à des couches d'entassement. Ainsi, Pour chaque syllabe présente dans un entassement, on voudra connaître la couche où elle se trouve, sa position dans la couche, la position de cette couche dans l'entassement, etc. L'ensemble de ces données, relatives à la topologie des entassements, est regroupé en figure 11.2.2.

donnée topologique
numéro de l'entassement dans l'échantillon
numéro de la couche dans l'entassement
numéro de la syllabe dans la couche
nombre d'entassements parents
nombre d'entassements enfants
discontinuité de l'entassement
nombre de couches de l'entassement
nombre de syllabes dans la couche en cours

FIGURE 11.2.2 – Tables des données topologiques étudiées

Un premier travail a donc consisté à exporter deux groupes de données :
- Un groupe "toutes syllabes" qui contient pour chacune des syllabes du corpus ses données prosodiques (figure 11.2.1)
- Un groupe "syllabes dans entassements" qui contient pour chacune des syllabes du corpus se trouvant dans un entassement ses données prosodiques et topologiques (figure 11.2.1 et 11.2.2)

Dans ce but, pour chaque échantillon du corpus, j'ai construit un multi-arbre (notion décrite en partie III). Ensuite, j'ai enrichi le multi-arbre avec le fichier TextGrid correspondant qui contient l'intégralité des données prosodiques. C'est ainsi que j'ai pu exporter ces données pour chaque syllabe de l'échantillon.

Ensuite, en exploitant le multi-arbre, on retrouve facilement l'ensemble des entassements d'un échantillon. De cette manière, on peut effectuer une boucle sur tous les entassements et pour chacun effectuer une autre boucle sur ses couches. Enfin, pour chacune de

ces couches, j'exporte l'ensemble de ces syllabes ainsi que leurs données prosodiques et topologiques. On trouvera un exemple de ce traitement pour la donnée f_0 en figure .11.2.3.

```
textGridFile = 'chemin vers le fichier textgrid'
texte_annote=' ici la transcription annotée'

# construction de l'arbre
root=buildTree(texte_annote)

#mise en correspondance avec la Tier 'pivot'
root.matchTextGrid(TextGridFile,'pivot')

#conversion en arbre d'entassement
root.pileConversion()

#recherche de la Tier 'mean_f0' de l'objet TextGrid, qui est une
    des propriétés du nœud racine
tg = tree.marks['TextGrid']
f0Tier = [t for t in tg if t.name()==mean_f0']

#boucle sur l'ensemble des piles de l'extrait et de leurs
   couches:
piles = [n for n in nodes if n.type=='pile']
for pile in piles:
    layers = [n for n in pile.sons if n.type=='layer']
    for layer in layers:
        f0s = layer.intervalsInTier(f0Tier)
        #on a ici toutes les valeurs de f0 des syllabes
           contenues pour cette pile et ce layer
```

FIGURE 11.2.3 – Processus d'extraction des information syntaxiques et prosodiques encodées dans la structure de données

Une fois qu'on a extrait ces informations de la structure pour ces deux groupes de données, il suffit de les exporter sous la forme de deux tableaux de données, où chaque ligne correspond à une syllabe et où chaque colonne correspond à une des informations qu'on veut connaître sur cette syllabe.

Ce sont des données qu'on peut dès lors exploiter pour mener une étude intono-syntaxique des entassements. L'intégralité des scripts MATLAB et Python développés pour ces travaux sont commentés en annexe. A partir de ces données, on peut produire certains tris permettant d'étudier le comportement inter-couche ou le comportement intra-couche des critères prosodiques. Je vais à présent détailler ces traitements spécifiques.

11.3 Étude des transitions

11.3.1 Définition des données

Pour mener à bien l'étude des transitions entre couches, l'approche choisie a consisté à comparer le comportement prosodique des transitions entre syllabes au niveau d'un changement de couche par rapport à celui observé dans d'autres contextes "témoins". Dans mon travail, j'ai défini une *transition* comme un couple de valeurs $(v(s), v(s+1))$, Définition transition

où $v(s)$ désigne la valeur prosodique (f_0 par exemple) d'une syllabe s et $v(s+1)$ la valeur prosodique de la syllabe suivante. Ainsi, une transition entre couches sera telle que s est la dernière syllabe d'une couche, et $s+1$ la première syllabe de la couche suivante.

Cette définition m'a permis de regrouper l'ensemble des transitions entre couches du corpus. Pour ce faire, j'ai récupéré l'ensemble des syllabes qui terminent une couche, et je les ai chacune appariées avec la syllabe suivante, de manière à former autant de transitions. Pour chacun des critères prosodiques étudiés (tels que $v = f_0$, v =ralentissement, v =proéminence, etc.), j'ai ainsi un ensemble de couples de transition inter-couches :

$$T_{\text{inter-couches}} = \{(v(s), v(s+1)) \mid s \text{ en fin de couche}\}$$

Ces couples sont représentés par des flèches sur la figure 11.3.1.

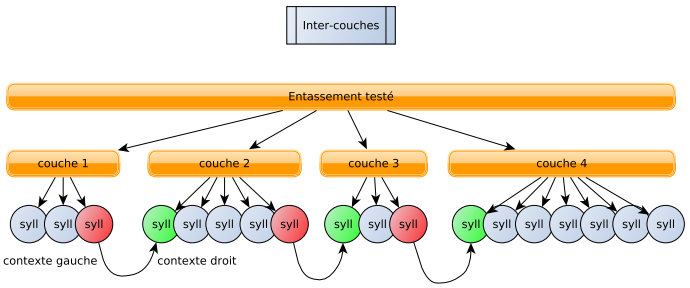


FIGURE 11.3.1 – Transitions inter-couches

Ces données constituent le groupe des transitions "inter-couches". Dans mon travail, j'ai comparé ces transitions inter-couches à deux autres types de transitions prises comme références. Le premier de ces groupes témoins est constitué de l'ensemble des transitions du corpus :

$$T_{\text{toutes-transitions}} = \{(v(s), v(s+1)) \mid \text{pour tout } s\}.$$

Ce groupe est représenté en figure 11.3.2.

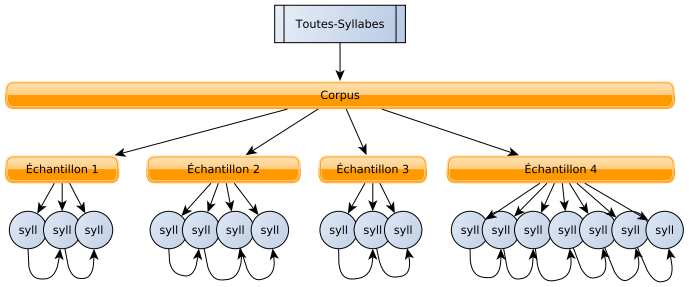


FIGURE 11.3.2 – Test toutes-syllabes

Comme on le voit le groupe $T_{\text{toutes-transitions}}$ donne bien un référent par rapport auquel comparer les transitions particulières entre couches. Cependant, ce groupe peut paraître un peu trop large et un deuxième groupe référent par rapport auquel comparer les transitions inter-couches est constitué des transitions intra-couches. Une transition intra-couche

Transitions intra-couches

est telle que s et $s+1$ sont dans la même couche :

$$T_{\text{intra-couche}} = \{(v(s), v(s+1)) \mid s \text{ et } (s+1) \text{ sont dans la même couche}\}.$$

Ce groupe est représenté en figure 11.3.3.

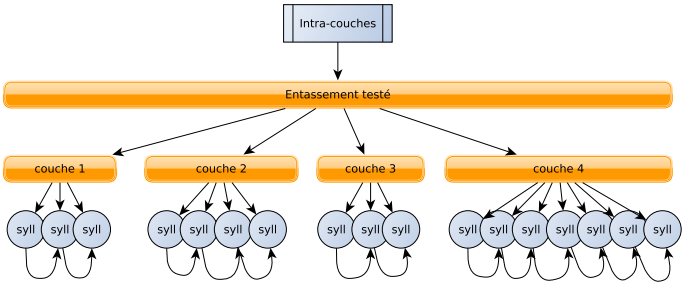


FIGURE 11.3.3 – Transitions intra-couches

L'étude des transitions inter-couches se fera donc en comparant ces trois lots de transitions. Une comparaison de $T_{\text{inter-couches}}$ à $T_{\text{intra-couches}}$ permettra d'identifier si, au sein des entassements, une transition inter-couche est différente d'une transition intra-couche. Enfin, la comparaison des transitions intra-couche et toutes-syllabes permet d'identifier si le fait d'être dans un entassement a une répercussion sur les transitions syllabiques.

On a donc importé dans MATLAB la matrice construite avec Python et détaillée en section précédente. Sur cette matrice, on peut sélectionner les différentes syllabes voulues en fonction du lot qu'on cherche à construire :

- Les dernières syllabes d'une couche seront telles que leur position en cours dans la couche est la même que le nombre de syllabes dans la couche.
- La dernière syllabe d'un entassement est telle qu'elle est la dernière syllabe de la dernière couche de l'entassement.

Tous les tests ont été effectués en ne considérant que les entassement n'ayant pas d'enfant entassement (voir section 9.4), à savoir les entassements continus uniques et les entassements continus enfants sans enfants.

Ce choix est dicté par des raisons pragmatiques, au moment de cette étude le typage des entassements n'est pas achevé et je n'ai donc pas pu l'intégrer dans la hiérarchique intono-syntaxique du multi-arbre. Le choix des entassements continus uniques et enfants sans enfant est dicté par des considération de nombre. En effet mon choix s'est porté sur cette catégorie d'entassement en raison de leur nombre suffisant pour mener cette étude. Les entassements discontinus en général et continus-parents étant beaucoup plus marginaux, il m'a paru plus prudent de les écarter de l'étude pour le moment.

Je récupère donc tous les entassements continus-uniques et continus-enfant-sans enfant du corpus. On trouvera le code source de ces scripts en annexe. Dans tous les cas, on récupère des tableaux de dimension $N \times 2$ où N est le nombre de transitions trouvées.

11.3.2 Comparaison des données

Pour chaque critère prosodique étudié, les données considérées sont :

- Un tableau T_{inter} de dimension $N_{\text{inter}} \times 2$
- Un tableau T_{intra} de dimension $N_{\text{intra}} \times 2$
- Un tableau T_{toutes} de dimension $N_{\text{toutes}} \times 2$.

Histogrammes bidimensionnels

Une première manière de représenter une donnée de transition est de l'afficher sur le plan. Ainsi, une donnée $(v(s), v(s+1))$ sera un point dans le plan (x, y). On peut alors déterminer la distribution de ces transitions dans le plan en construisant un histogramme bidimensionnel des observations. Cet histogramme donnera pour chaque transition sa proportion d'apparition dans le corpus. Plusieurs remarques importantes peuvent être faites sur ce type de représentation :

- contrairement à l'affichage de $v(s+1) - v(s)$, il permet de visualiser facilement la dynamique des transitions, tout en étant très informatif sur les valeurs absolues observées. Ainsi, on y voit assez bien si les valeurs $v(s)$ sont faibles ou fortes en plus de leur lien avec les valeurs suivantes $v(s+1)$. En cas d'absence de phénomène de rupture, les valeurs de $v(s)$ seront souvent équivalentes aux valeurs de $v(s+1)$, donc l'histogramme se concentrera sur sa diagonale.
- Dans le cas de données catégorielles, faire une différence n'a pas de sens. Ainsi, si on observe hésitation $(s) =$ _ et hésitation$(s+1) =$ H, comment définir leur différence ? Assimiler une transition à un saut de valeur réelle est trop restrictif dans le cadre de notre étude. Par contre on verra en section que les histogrammes bidimensionnel de ces données catégorielles sont bel et bien informatifs sur la présence de ruptures prosodiques entre couches.

Une deuxième manière de représenter une donnée de transition est d'étudier $\Delta v_s = v(s+1) - v(s)$. Comme on vient de le voir, une telle étude n'est valable que pour les caractéristiques réelles (ici f_0 et ralentissement). Un biais fort de cette approche est qu'il est possible que Δv_s dépende fortement de $v(s)$. Par exemple, dans le cas du f_0 c'est plutôt une différence de pitch exprimé en demi-ton qui est à privilégier. De plus, la seule étude de Δv_s ne donne pas d'information sur les valeurs absolues des variables étudiées. Cependant, une telle étude a un avantage certain qui est de permettre l'utilisation de tests statistiques permettant de quantifier la ressemblance des distributions de ces trois groupes de transitions. Ainsi, j'effectuerai pour chaque comparaison $T_{\text{inter}}/T_{\text{intra}}$ et $T_{\text{intra}}/T_{\text{toutes}}$ des tests statistiques de plusieurs sortes permettant de déterminer si les distributions de ces données sont identiques.

11.4 Étude du patron prosodique des couches

11.4.1 Définition des données

État de l'art

Grâce au formalisme adopté, il est possible d'extraire pour chaque couche chaque entassement du corpus l'ensemble des syllabes et toutes les caractéristiques qui lui sont associées. On peut donc procéder à une analyse complète de leurs profils prosodiques.

Des études existent qui se concentrent sur la ligne mélodique de certains segments syntaxiques [64, 63, 68] ou bien de leurs profils de ralentissements [36]. Pour des raisons de faisabilité dans le temps imparti, je me suis concentrée pour ma part à une étude simplifiée.

Approche simplifiée

L'approche simple que je propose est d'étudier pour chaque couche le rapport entre les valeurs initiales, finales et intermédiaires des f_0 et ralentissements. Ainsi, pour chaque couche, j'ai extrait la f_0 et ralentissement de sa première et dernière syllabe ainsi que la valeur médiane des f_0 et ralentissements intermédiaires.

Cette extraction de données a conduit à la constitution d'une matrice F_0, de dimension $N \times 3$, où N est le nombre de couches dans le corpus. La première colonne regroupe les valeurs initiales des f_0, la deuxième les valeurs intermédiaires et la dernière les valeurs finales. De la même manière, j'ai généré une matrice R regroupant les valeurs des ralentissements.

L'intérêt de cette approche est qu'elle permet de regrouper les différentes observations par couche. On dira que les données sont *appariées*. On peut alors faire l'hypothèse qu'au sein d'une même couche, le locuteur sera le même et que les caractéristiques prosodiques seront donc comparables, contrairement à une approche qui regrouperait toutes les valeurs initiales, intermédiaires et finales entre elles, sans les apparier par couche.

En effet, analyser ensemble toutes les valeurs prosodiques du corpus est impossible à cause de la grande diversité des locuteurs du corpus. Cependant, la manière dont on au départ été codés les locuteurs dans les transcriptions ne permet ni de faire une distinction en genre, ni de distinguer les locuteurs entre eux. Le locuteur L1 du fichier M0001 n'est pas le même que le locuteur L1 du fichier D2002 mais ils portent la même étiquette. Cet état de fait étant indépendant de ma volonté j'ai donc du adopter une autre approche. L'appariement par couche m'est apparu dans ce contexte la meilleure manière d'analyser les données.

11.4.2 Analyse des résultats

Munis des matrices F_0 et R donnant les valeurs prosodiques appariées par couche et par position dans la couche (initiale, finale ou intermédiaire), j'ai procédé à une étude statistique au sein des couches des valeurs initiales et intermédiaires, ainsi qu'une étude des valeurs intermédiaires et finales.

Cette étude statistique se présente sous la forme d'un t-test qui permet de déterminer si la distribution des f_0 initiales est la même que celle des f_0 intermédiaires, et de même pour la f_0 finale. Ces comparaisons permettent de déterminer si une rupture mélodique est en moyenne observée sur l'ensemble des couches du corpus, induisant l'existence de profils mélodiques. De la même manière, une étude des ralentissement permet de déterminer l'existence d'une modification significative des durées syllabes au sein des couches.

En pratique, cette étude se fait en utilisant le test de Student apparié, qui est un test paramétrique permettant de déterminer si deux échantillons appariés x et y (c'est à dire dont les éléments sont deux à deux en correspondance, comme ici nos colonnes de F_0) ont la même moyenne, sous hypothèse qu'ils sont tous les deux distribués selon une loi Gaussienne de même variance.

Chapitre 12

Résultats

12.1 Étude des transitions entre couches

12.1.1 Étude descriptive

Après avoir récolté les données de transitions de la manière décrite en section 11.3, j'en ai effectué une étude descriptive. Pour ce faire, je représente pour chaque critère prosodique étudié : f_0, ralentissement et proéminence sa distribution bidimensionnelle [1], pour les trois groupes considérés : toutes-syllabes, intra-couche et inter-couches. L'échelle de couleur indique la proportion (%) dans le corpus de la transition considérée.

Étude descriptive

1. Le contexte gauche est la première syllabe et le contexte droit la syllabe qui suit.

Pour la f_0 tout d'abord :

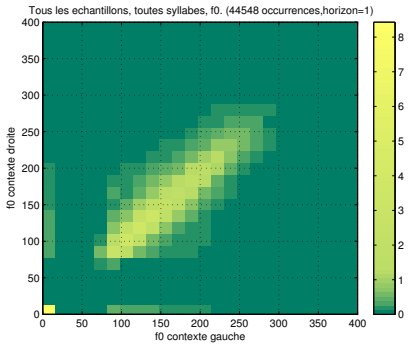


FIGURE 12.1.1 – Histogramme bidimensionnel des transitions de f_0 toutes-syllabes.

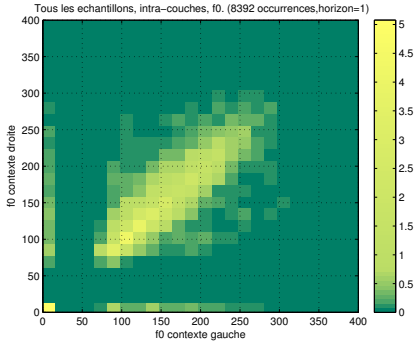


FIGURE 12.1.2 – Histogramme bidimensionnel des transitions de f_0 intra-couche.

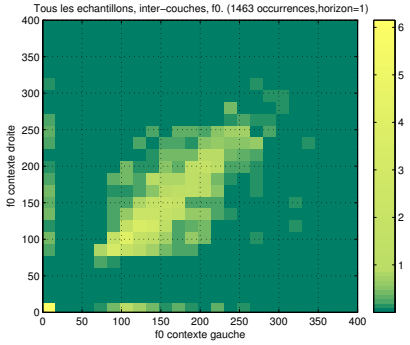


FIGURE 12.1.3 – Histogramme bidimensionnel des transitions de f_0 inter-couches.

Ces figures descriptives apportent une première constatation. On remarque que les valeurs se dispersent plus autour de la diagonale pour le groupe inter-couches que pour les autres. Cela constitue un indice de la présence d'une rupture dans la ligne de déclinaison lors de la transition d'une couche à l'autre dans les entassements. Cependant, on remarque que l'histogramme des transitions inter-couches et intra-couche est assez similaire. De plus amples observations sont encore nécessaires pour pouvoir affirmer l'existence d'un tel phénomène de rupture.

En ce qui concerne les ralentissements, les histogrammes bidimensionnels sont les suivants :

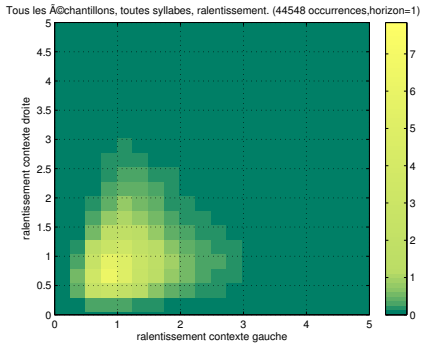


FIGURE 12.1.4 – Histogramme bidimensionnel des transitions de ralentissement toutes-syllabes

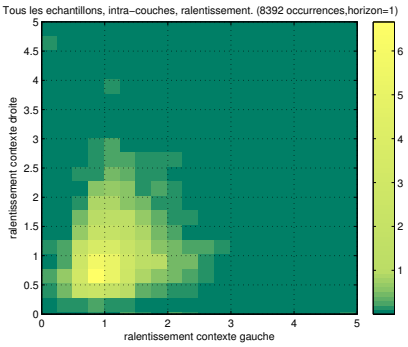


FIGURE 12.1.5 – Histogramme bidimensionnel des transitions de ralentissement intra-couche

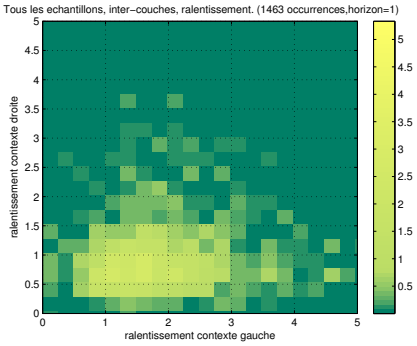


FIGURE 12.1.6 – Histogramme bidimensionnel des transitions de ralentissement inter-couches

Ces graphiques descriptifs viennent appuyer les remarques que l'on a pu faire pour les variation de f_0. En effet on remarque que les valeurs se détachent nettement plus de la diagonale pour le groupe inter-couches que pour les autres. Les syllabes de fin de couche

(contexte gauche de la paire) semblent durer plus longtemps que les premières syllabes des couches (contexte droit). L'information de ralentissement [36] apparaît ainsi comme une caractéristique forte des transitions entre couches des entassements.

Comme souligné en section 11.3, l'affichage de l'histogramme bidimensionnel est indispensable pour l'étude des variables catégorielles. Parmi elles, on trouve tout d'abord la caractéristique de proéminence d'une syllabe, qui peut être "faible", "forte", "incompréhensible", "pause" et "délimiteur de période". Dans la suite, ces valeurs sont codées de la manière suivante :

symbole	valeur
W	weak
S	strong
%	caractère non reconnu par l'aligneur, signifie segment incompréhensible
_	pause
À£	délimiteur de périodes

On a alors :

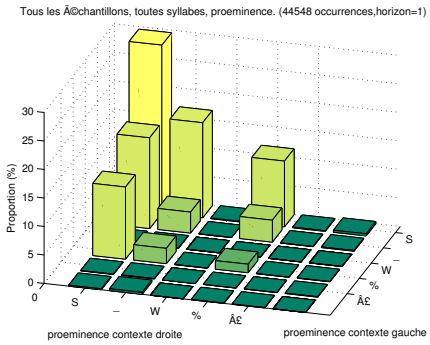


FIGURE 12.1.7 – Histogramme bidimensionnel des transitions de proéminence toutes-syllabes

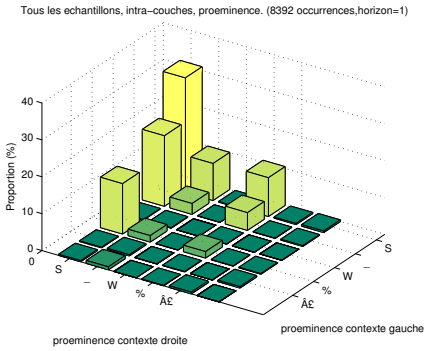


FIGURE 12.1.8 – Histogramme bidimensionnel des transitions de proéminence intra-couches

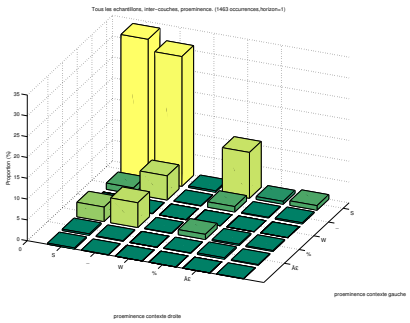


FIGURE 12.1.9 – Histogramme bidimensionnel des transitions de proéminence inter-couches

L'analyse de ces transitions permet de voir que les proéminences détectées sont souvent fortes et que d'une syllabe à proéminence "S"(strong [2]), on passe souvent à une autre syllabe à proéminence "S". Si les transitions entre pauses et fortes proéminences se font en moyenne à peu près autant dans les deux sens pour les groupes intra-couche et toutes-syllabes, on remarque que pour les transitions inter-couches, on passe très souvent d'une forte proéminence à une pause, et non pas le contraire. Cette observation permet une fois de plus de conclure à l'existence d'une rupture prosodique au moment d'un changement de couche d'un entassement. Ces relevés, pris sur des milliers d'occurrences du phénomène sur plus de trois heures de parole, tend à justifier la validité d'un relevé systématique des proéminences [4]. Bien entendu, les recherches préliminaires effectuées ici ne donnent encore que des tendances : l'étude d'une interaction entre ralentissements et proéminences offre des perspectives enthousiasmantes pour une étude prosodique plus poussée de phénomènes syntaxiques.

12.1.2 Étude quantitative

Comme souligné en section 11.3, j'ai effectué des tests statistiques permettant de déterminer si les distributions des groupes toutes-syllabes, intra-couches et inter-couches des Δf_0 et Δralentissements sont les mêmes. De part leur nature, ces trois échantillons ne sont pas appariés. Il n'ont en effet pas la même taille : le groupe toute syllabe est constitué de l'ensemble des transitions entre syllabes du corpus (plus de 40000), le groupe intra-couche est constitué de l'ensemble des transitions entre syllabes d'une même couche (environ 8000) et enfin, le groupe inter-couches est constitué de l'ensemble des transitions entre couches (environ 1500).

De manière comparer ces distributions, j'ai cherché à effectuer des tests statistiques. Afin de savoir si je pouvais dans ce but utiliser un test paramétrique comme le teste de Student, je devais d'abord vérifier si ces Δf_0 et Δralentissement peuvent être considérés comme distribués selon une loi Gaussienne. Pour ce faire, j'ai effectué sur chacun un test du χ^2 d'adéquation avec la distribution normale (voir annexe A.2). Les scores obtenus sont les suivants :

2. Les proéminences sont annotées "forte" ou "faible" relativement au contexte.

Pour le Δf_0 :

Groupe	moyenne m	variance σ	p	degré de liberté
Inter-couches	0.58	4.91	1.0000	13
Intra-couches	-0.17	4.6	1.0000	13
Toutes-syllabes	0.00	4.61	1.0000	8

(a) Test de normalité par le χ^2 d'adéquation pour les groupes inter-couches, intra-couches et toutes-syllabes effectués sur le Δf_0

Pour les ralentissements :

Groupe	moyenne m	variance σ	p	degré de liberté
Inter-couches	0.84	2.12	1.0000	5
Intra-couches	-0.13	1.22	1.0000	4
Toutes-syllabes	0.00	1.32	1.0000	4

(b) Test de normalité par le χ^2 d'adéquation pour les groupes inter-couches, intra-couches et toutes-syllabes effectués sur le Δralentissement

FIGURE 12.1.10 – Résultats des tests d'adéquation à la distribution normale

Comme on le voit, le test conduit à rejeter l'hypothèse de normalité des données. Il n'est donc pas possible d'effectuer un test paramétrique de même moyenne entre les distributions des différents groupes, qui suppose cette distribution Gaussienne. Une autre alternative est d'utiliser pour comparer les distributions des groupes un test non-paramétrique tel que celui de KOLMOGOROV-SMIRNOV à deux échantillons x et y (voir annexe A.4), dont l'hypothèse nulle est que x et y sont distribués selon la même distribution. J'ai effectué pour chaque couple de groupes le test, et j'ai obtenu les résultats suivants :

Pour le Δf_0 :

Groupe	Inter-couches	Intra-couches	Toutes-syllabes
Inter-couches	X	rejetée (p=0.000)	rejetée (p=0.000)
Intra-couches	rejetée (p=0.000)	X	rejetée (p=0.000)
Toutes-syllabes	rejetée (p=0.000)	rejetée (p=0.000)	X

(a) Hypothèse H_0 de même distribution entre groupes, pour Δf_0. Test de KOLMOGOROV-SMIRNOV

Pour les ralentissements :

Groupe	Inter-couches	Intra-couches	Toutes-syllabes
Inter-couches	X	rejetée (p=0.000)	rejetée (p=0.000)
Intra-couches	rejetée (p=0.000)	X	rejetée (p=0.000)
Toutes-syllabes	rejetée (p=0.000)	rejetée (p=0.000)	X

(b) Hypothèse H_0 de même distribution entre groupes, pour Δralentissement. Test de KOLMOGOROV-SMIRNOV

FIGURE 12.1.11 – Résultats des tests de KOLMOGOROV-SMIRNOV à deux échantillons

On peut donc dire que selon les deux critères mélodiques (f_0) et ralentissements, les distributions des transitions entre couches sont significativement différentes de celles intra-couches ou toute-syllabes.

Cependant, il faut garder à l'esprit que les données de prosodie considérées ne sont pas accompagnées d'une information sur le locuteur et l'ensemble des données du corpus ont ainsi été rassemblées sans pouvoir distinguer également l'effet du genre du locuteur sur les résultats. Ce n'est que pour l'analyse des patrons prosodiques, où les données peuvent être appariées, qu'une telle étude a pu être possible.

La mise en évidence de ruptures entre couches, visibles sur les histogrammes bidimensionnels, est donc confirmée par des tests statistiques, mais de plus amples recherches permettraient sont nécessaires pour pouvoir tirer des conclusions plus détaillées sur ce phénomène.

12.2 Étude des patrons prosodiques intra-couche

Dans cette section, je présente les résultats de l'étude sur l'existence de patrons prosodiques spécifiques au sein des couches. Comme souligné en section 11.4, la particularité de cette étude est qu'elle regroupe pour chaque couche ses valeurs initiales, intermédiaire et finale de f_0 et de ralentissement. Ainsi, on peut considérer qu'on a autant de ces triples observations qu'on a de couches dans le corpus.

Pour cette raison, le formalisme adéquat pour étudier ces données est d'effectuer des tests de variables appariées. Comme on l'a vu, la distribution marginale des données de f_0 en début ou fin de couche sur sur l'ensemble du corpus ne sont pas exploitables car elles correspondent à différents locuteurs et différents genres. Cependant, j'ai estimé qu'une étude des distributions marginales des ralentissements pouvait se justifier, puisque le ralentissement des syllabes semble moins sensible au changement de locuteur. Ainsi, j'ai tout d'abord étudié les distributions marginale des ralentissements en début, en milieu et en fin de couche et j'ai comparé ces distributions.Ce travail est présenté en section .12.2.1.

La nature des données permet cependant d'effectuer une très informative étude appariée qui peut renseigner sur les tendances observées au sein d'une même couche sur l'ensemble du corpus. Cette étude est présentée en section 12.2.2.

12.2.1 Étude des distributions marginales du ralentissement

Une première manière de représenter les distributions marginales de la f_0 et du ralentissement en début, milieu et fin de couche est d'en afficher des boites à moustache. Cette représentation est souvent choisie car elle représente de manière graphique les différents quantiles de la distribution. Si la médiane est représentée par un trait rouge, l'espace entre le premier et le troisième quartile est représenté par un rectangle, qui contient donc la moitié des données. Les valeurs extrêmes sont représentées par des points individuels. On a les résultats suivants :

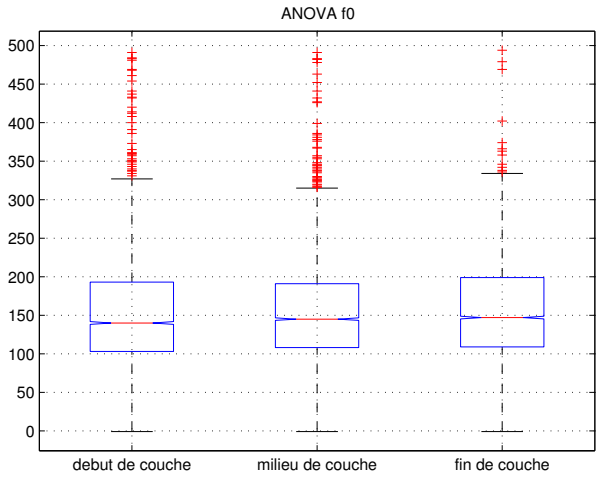


FIGURE 12.2.1 – Diagramme en boîte des distributions marginales de la f_0 en début, milieu et fin de couche

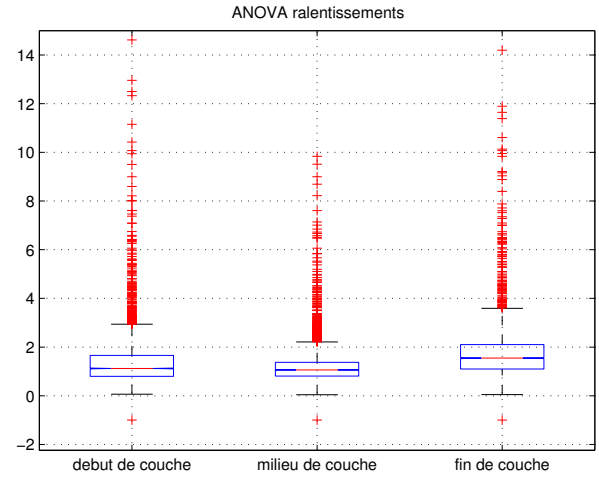


FIGURE 12.2.2 – Diagramme en boîte des distributions marginales des ralentissements en début, milieu et fin de couche.

Sur ces figures, on voit que sur l'ensemble du corpus, les syllabes en milieu de couche sont plus courtes que celles en périphérie. On voit ainsi émerger un marquage temporel des couches. En revanche pour ce qui est de le f_0 la démarcation est moins nette, on note toutefois une f_0 sensiblement plus élevée en fin de couche.

Une représentation conjointe de ces deux critères permet aussi d'évaluer si leurs valeurs sont toujours corrélées de la même façon ou pas :

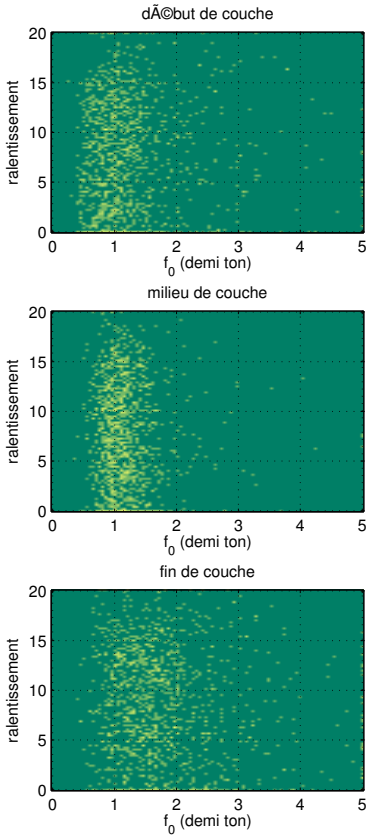


FIGURE 12.2.3 – Histogramme bidimensionnel de l'évolution de la f_0 par rapport au ralentissement début, milieu et fin de couche.

L'histogramme bidimensionnel présenté ci-dessus décrit bien l'allongement de la durée par rapport à l'augmentation de la f_0. En effet on constate qu'en fin de couche notamment les valeurs des ralentissements augmentent en même temps que les valeurs de la f_0. Cela dit, il est difficile de tirer de ces tracés plus de conclusions sur le comportement du ralentissement et de la f_0 au sein d'une couche donnée. C'est à cette question que je vais à présent me consacrer.

12.2.2 Étude appariée

Pour chaque couche, on dispose de l'information de ralentissement et de f_0 de ses syllabes. J'ai effectué une comparaison appariée des valeurs de ralentissement et de f_0 initiales et intermédiaires, ainsi qu'une comparaison appariée des valeurs finales et intermédiaires. L'objectif de ces comparaisons est de déterminer si les moyennes des propriétés prosodiques sont comparables en début, en milieu, et fin de couche. On trouvera donc une série de 4 t-tests appariés réalisés avec le logiciel R[75] et comparant pour la f_0 et pour le ralentissement les distributions initiales et finales à la distribution intermédiaire :

Étude appariée de la f_0

Valeur initiale / valeur intermédiaire

```
t.test(donnees_fo$debut_f0,donnees_fo$milieu_f0, paired=T)
t = 0.6344, df = 1098, p-value = 0.526
alternative hypothesis : true difference in means is not equal to 0
95 percent confidence interval :
-0.1591629 0.3112495
sample estimates :
mean of the differences
0.07604327
```

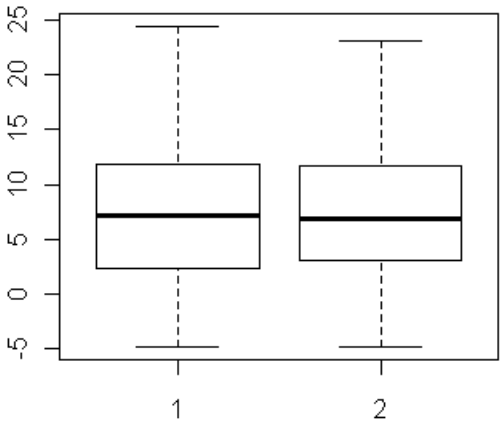


FIGURE 12.2.4 – Diagramme en boîte des distributions des f_0 en début (1) et milieu (2) de couche

Ci-dessus, en comparant pour la f_0, début et milieu, on n'a pas de différence significative entre les moyennes.

Valeur finale / valeur intermédiaire

```
t.test(donnees_fo$fin_f0,donnees_fo$milieu_f0, paired=T)
t = 6.2321, df = 1098, p-value = 6.548e-10
alternative hypothesis : true difference in means is not equal to 0
95 percent confidence interval :
0.6065766 1.1640395
sample estimates :
mean of the differences
0.8853081
```

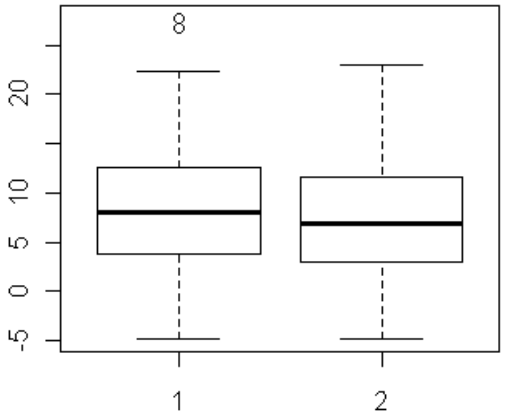


FIGURE 12.2.5 – Diagramme en boîte des distributions des f_0 en fin (1) et milieu (2) de couche

Ci-dessus, en comparant pour la f_0 fin et milieu, on a une différence significative entre les moyennes. Ce résultat tend à montrer que la f_0 au sein d'une couche d'entassement adopte un contours montant et est donc un signe fort de l'existence du phénomène d'entassement au niveau prosodique.

Étude appariée du ralentissement

Valeur initiale / valeur intermédiaire

```
t.test(donnees_slowing$fin_ralentissement,donnees_slowing$milieu_ralentissement,
paired=T)
t = 20.1662, df = 1385, p-value < 2.2e-16
alternative hypothesis : true difference in means is not equal to 0
95 percent confidence interval :
0.6238259 0.7582701
sample estimates :
mean of the differences
0.691048
```

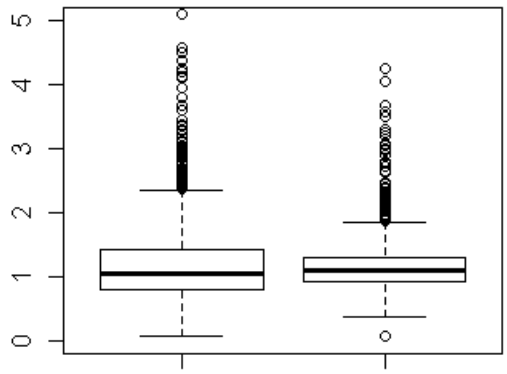


FIGURE 12.2.6 – Diagramme en boîte des distributions des ralentissements en début (1) et milieu (2) de couche

En comparant pour le ralentissement , début et milieu On a une différence significative.

Valeur finale / valeur intermédiaire

t.test(donnees_slowing$fin_ralentissement,donnees_slowing$milieu_ralentissement, paired=T)
t = 20.1662, df = 1385, p-value < 2.2e-16
alternative hypothesis : true difference in means is not equal to 0
95 percent confidence interval :
0.6238259 0.7582701
sample estimates :
mean of the differences
0.691048

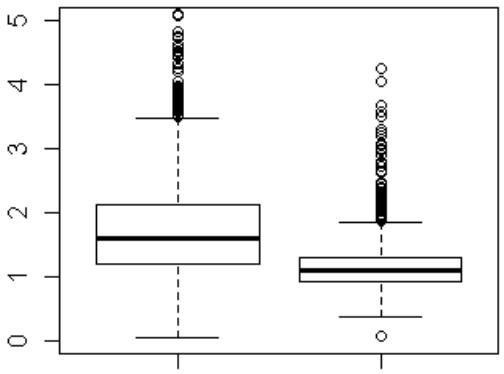


FIGURE 12.2.7 – Diagramme en boîte des distributions des ralentissements en fin (1) et milieu (2) de couche

En comparant pour le ralentissement en fin et milieu, on a une différence significative entre les moyennes. Il est remarquable en comparant les deux diagrammes (12.2.6 et 12.2.7) que cette différence à l'air plus importante que celle entre début et milieu. Ce résultat vient confirmer de manière quantitative ceux obtenus par l'étude descriptive des histogrammes bidimensionnels des transitions inter-couches en section 12.1.1, à la fois pour les ralentissements et pour les proéminences.

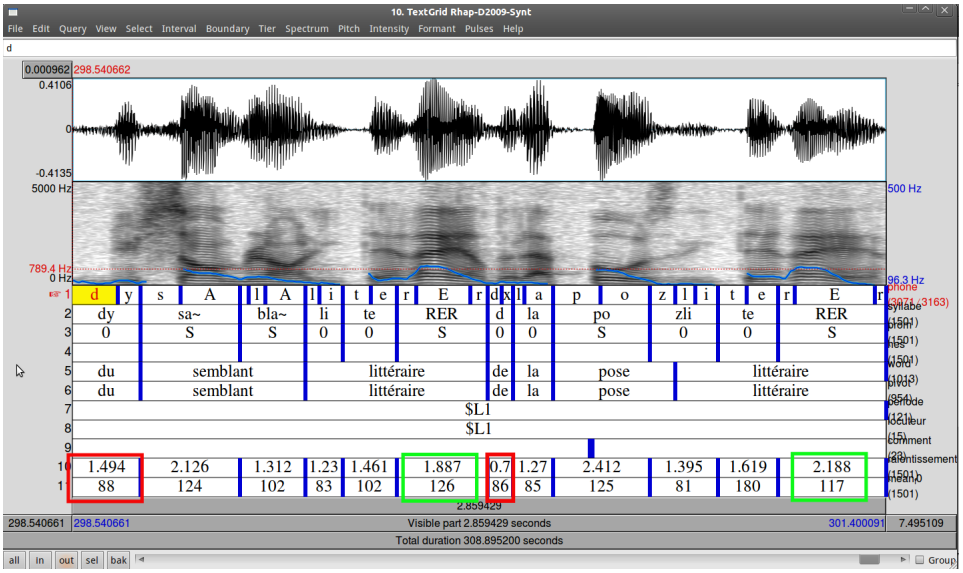


FIGURE 12.2.8 – Exemple d'allongement syllabique et d'augmentation de la f_0 en fin de couche

On montre ainsi que le ralentissement s'accroît en fin de couche (voir figure 12.2.8). On

note toutefois que l'allongement est observé plus volontiers sur le début que sur la fin pour les séquences entre pauses.

Les résultats ne permettent toutefois pas d'établir une corrélation parfaite entre allongement syllabique et durée de f_0 (voir la figure 12.2.9)

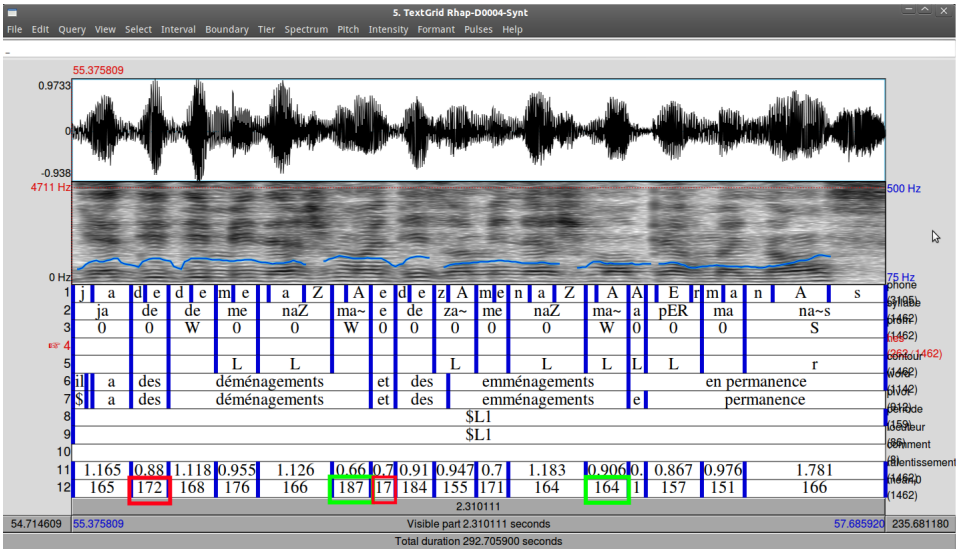


FIGURE 12.2.9 – Exemple d'augmentation de la f_0 alors que le ralentissement n'augmente pas en fin de couche

Enfin, voici deux exemples de figures (12.2.10 et 12.2.11) qui sont des cas classiques avec une augmentation de f_0. On note toutefois que la figure ne comporte pas de contour montant mais seulement une valeur moyenne de sa f_0 plus élevée.

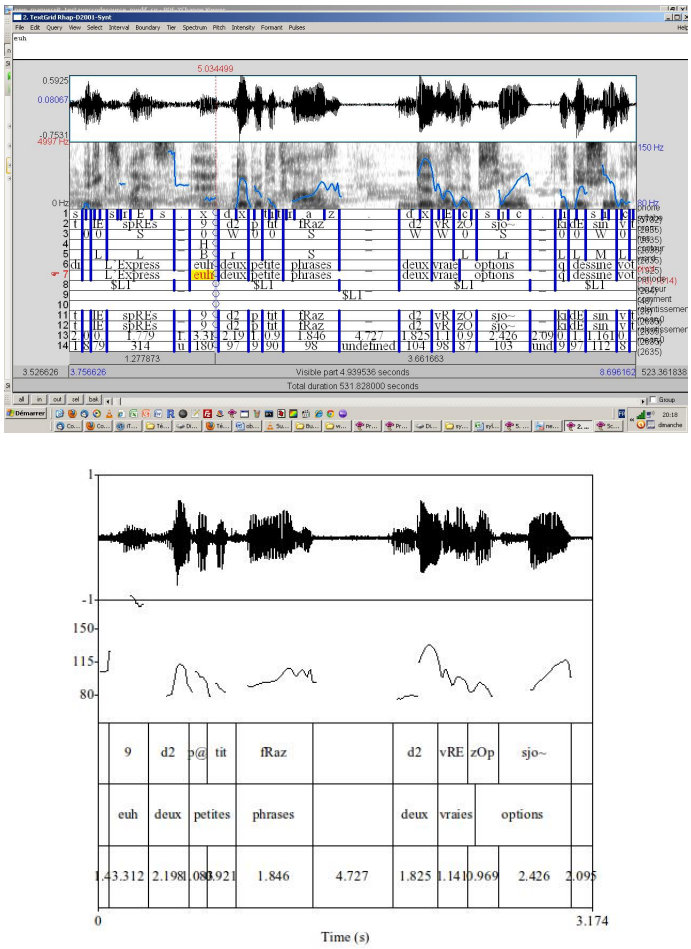


FIGURE 12.2.10 – Contours de f_0 montants en fin de couche de l'entassement "{ deux petites phrases | deux vraies options }"

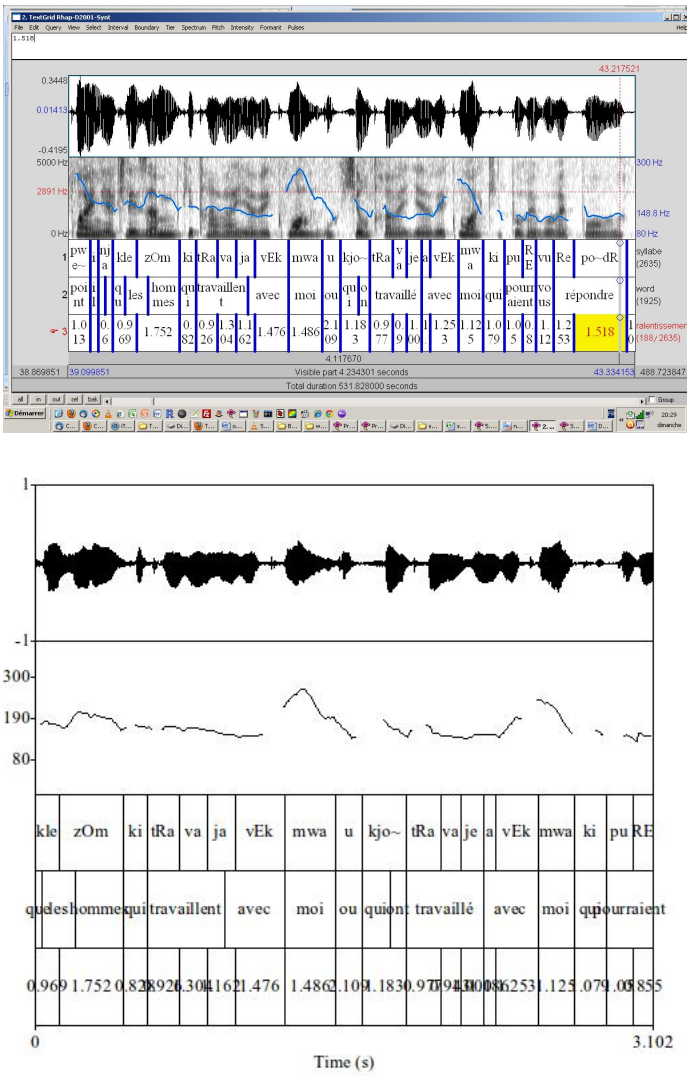


FIGURE 12.2.11 – Valeur moyenne de f_0 montante en fin de couche de l'entassement "{ qui travaillent avec moi | ^ou qui ont travaillé avec moi }"

Chapitre 13

Conclusion

L'étude intono-syntaxique de l'entassement a permis de mettre en évidence l'existence de ce phénomène syntaxique sur le plan prosodique. En effet, l'utilisation des outils présentés en partie II a permis d'extraire pour l'ensemble des syllabes appartenant à des entassements un spectre large de caractéristiques prosodiques telles que la f_0, le ralentissement et les proéminences.

Une étude statistique des ces observations, constituées de plusieurs milliers, voire dizaines de milliers d'occurrences, a permis de mettre fortement en évidence la présence de ruptures prosodiques aux transitions de couches. Ces ruptures se manifestent par un allongement des syllabes en fin de couche, par un accroissement léger de la f_0 et par la transition fréquente d'une forte proéminence à une pause.

Ces conclusions ont été obtenues de deux manières différentes. Une étude des distributions des transitions a tout d'abord été menée. Des représentations bidimensionnelles synthétiques m'ont permis de comparer les transitions inter-couches aux autres transitions du corpus, et en particulier de pouvoir exploiter les valeurs prosodiques catégorielles telles que la proéminence.

Dans un deuxième temps, une étude quantitative est venue confirmer ces résultats. Il me semble remarquable de constater que les différentes analyses et critères prosodiques étudiés semblent tous aller dans le même sens, c'est à dire confirmer d'une manière irréfutable l'existence d'un marquage prosodique du phénomène syntaxique de l'entassement.

Une perspective à envisager serait de reproduire ce type d'étude avec les entassements typés et les autre configurations topologiques. Un travail d'annotation de chaque locuteur en fonction de son sexe est également envisageable et devrait être disponible prochainement.

Quatrième partie

Conclusion Générale

J'ai présenté dans ce mémoire une structure hiérarchique objet enrichie de l'ensemble de données syntaxiques et prosodiques extraites des annotation du corpus Rhapsodie. Dans cette présentation on a proposé une systématisation informatique du système d'annotation syntaxique du corpus Rhapsodie pour son exploitation par le parseur FRMG du laboratoire Alpage et son enrichissement de données prosodiques.

Cette proposition allie des structures de données objet, adaptées au formalisme souhaité, et les différents algorithmes permettant de mettre en œuvre cette proposition [7]. Nous avons vu que cette proposition produit effectivement les structures souhaitées. Il a de plus été démontré que l'implémentation, de la totalité de l'information encodée dans le balisage, dans un multi-arbre est nécessaire, justifiant ainsi le choix théorique d'un structure objet globale. J'ai ainsi montré l'efficacité du système d'annotation syntaxique de Rhapsodie à condition d'opter pour une vision en multi-dimension pour son implémentation et sa pertinence quant à sa représentation arborée.

J'ai démontré l'efficience de cette hiérarchie en donnant un exemple concret de son exploitation pour une étude intono-syntaxique. En effet, j'ai montrée qu'il était possible d'effectuer une analyse prosodique sur des données syntaxiques complexes, l'étude présentée ici n'étant qu'un exemple des possibilités offertes par la structure en multi-arbre. J'ai en effet pu montrer par cette étude qu'un certains nombre de critères prosodiques était particulier au phénomène d'entassement étudié en mettant à profit ces structures de données. Cette étude a été rendue possible par l'exploitation de structure objet présentée avant et par une série des requêtes statistiques.

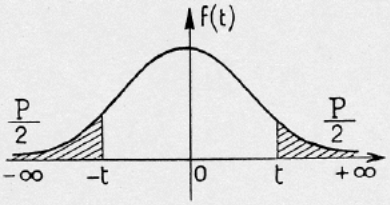
La vision en multi-arbre nous permet d'imaginer encore un plus grand nombre de manipulations et offre la possibilité de l'enrichir d'autres hiérarchies linguistiques (sémantiques, pragmatiques etc.). En effet à partir du moment où une annotation est convertible en une hiérarchie d'information liée à la transcription, il est possible d'intégrer cette hiérarchie au multi-arbre. Il reste toutefois des améliorations à apporter, en particulier un travail sur la conversion du multi-arbre en graphe. Cela permettrait un plus grande souplesse dans les manipulations et parcours possibles.

Un tel formalisme bien que n'étant pas classique, ouvre cependant des perspectives en terme de stockage à grande échelle des données linguistiques. Les bases de données ont évolué et les possibilités offertes par les bases de données orientées objet ou de graphes pour ce type de structures permettrait une plus grande optimisation de traitement des données.

Annexe A

Tests statistiques considérés

A.1 Table du t-test de Student



ν \ P	0,90	0,80	0,70	0,60	0,50	0,40	0,30	0,20	0,10	0,05	0,02	0,01	0,001
1	0,158	0,325	0,510	0,727	1,000	1,376	1,963	3,078	6,314	12,706	31,821	63,657	636,619
2	0,142	0,289	0,445	0,617	0,816	1,061	1,386	1,886	2,920	4,303	6,965	9,925	31,598
3	0,137	0,277	0,424	0,584	0,765	0,978	1,250	1,638	2,353	3,182	4,541	5,841	12,929
4	0,134	0,271	0,414	0,569	0,741	0,941	1,190	1,533	2,132	2,776	3,747	4,604	8,610
5	0,132	0,267	0,408	0,559	0,727	0,920	1,156	1,476	2,015	2,571	3,365	4,032	6,869
6	0,131	0,265	0,404	0,553	0,718	0,906	1,134	1,440	1,943	2,447	3,143	3,707	5,959
7	0,130	0,263	0,402	0,549	0,711	0,896	1,119	1,415	1,895	2,365	2,998	3,499	5,408
8	0,130	0,262	0,399	0,546	0,706	0,889	1,108	1,397	1,860	2,306	2,896	3,355	5,041
9	0,129	0,261	0,398	0,543	0,703	0,883	1,100	1,383	1,833	2,262	2,821	3,250	4,781
10	0,129	0,260	0,397	0,542	0,700	0,879	1,093	1,372	1,812	2,228	2,764	3,169	4,587
11	0,129	0,260	0,396	0,540	0,697	0,876	1,088	1,363	1,796	2,201	2,718	3,106	4,437
12	0,128	0,259	0,395	0,539	0,695	0,873	1,083	1,356	1,782	2,179	2,681	3,055	4,318
13	0,128	0,259	0,394	0,538	0,694	0,870	1,079	1,350	1,771	2,160	2,650	3,012	4,221
14	0,128	0,258	0,393	0,537	0,692	0,868	1,076	1,345	1,761	2,145	2,624	2,977	4,140
15	0,128	0,258	0,393	0,536	0,691	0,866	1,074	1,341	1,753	2,131	2,602	2,947	4,073
16	0,128	0,258	0,392	0,535	0,690	0,865	1,071	1,337	1,746	2,120	2,583	2,921	4,015
17	0,128	0,257	0,392	0,534	0,689	0,863	1,069	1,333	1,740	2,110	2,567	2,898	3,965
18	0,127	0,257	0,392	0,534	0,688	0,862	1,067	1,330	1,734	2,101	2,552	2,878	3,922
19	0,127	0,257	0,391	0,533	0,688	0,861	1,066	1,328	1,729	2,093	2,539	2,861	3,883
20	0,127	0,257	0,391	0,533	0,687	0,860	1,064	1,325	1,725	2,086	2,528	2,845	3,850
21	0,127	0,257	0,391	0,532	0,686	0,859	1,063	1,323	1,721	2,080	2,518	2,831	3,819
22	0,127	0,256	0,390	0,532	0,686	0,858	1,061	1,321	1,717	2,074	2,508	2,819	3,792
23	0,127	0,256	0,390	0,532	0,685	0,858	1,060	1,319	1,714	2,069	2,500	2,807	3,767
24	0,127	0,256	0,390	0,531	0,685	0,857	1,059	1,318	1,711	2,064	2,492	2,797	3,745
25	0,127	0,256	0,390	0,531	0,684	0,856	1,058	1,316	1,708	2,060	2,485	2,787	3,725
26	0,127	0,256	0,390	0,531	0,684	0,856	1,058	1,315	1,706	2,056	2,479	2,779	3,707
27	0,127	0,256	0,389	0,531	0,684	0,855	1,057	1,314	1,703	2,052	2,473	2,771	3,690
28	0,127	0,256	0,389	0,530	0,683	0,855	1,056	1,313	1,701	2,048	2,467	2,763	3,674
29	0,127	0,256	0,389	0,530	0,683	0,854	1,055	1,311	1,699	2,045	2,462	2,756	3,659
30	0,127	0,256	0,389	0,530	0,683	0,854	1,055	1,310	1,697	2,042	2,457	2,750	3,646
40	0,126	0,255	0,388	0,529	0,681	0,851	1,050	1,303	1,684	2,021	2,423	2,704	3,551
80	0,126	0,254	0,387	0,527	0,679	0,848	1,046	1,296	1,671	2,000	2,390	2,660	3,460
120	0,126	0,254	0,386	0,526	0,677	0,845	1,041	1,289	1,658	1,980	2,358	2,617	3,373
∞	0,126	0,253	0,385	0,524	0,674	0,842	1,036	1,282	1,645	1,960	2,326	2,576	3,291

FIGURE A.1.1 – Table t-test STUDENT

A.2 Test de normalité avec le Test du χ^2 d'adéquation

Pour un vecteur x de données, on peut effectuer un test du χ^2 d'adéquation afin de vérifier la normalité de x, c'est à dire de déterminer s'il est raisonnable de considérer que les éléments de x sont distribution selon une loi Gaussienne. Le test du χ^2 d'adéquation est une méthode statistique qui peut répondre à cette question. On compare la distribution empirique de x à la distribution théorique issue d'une loi normale de moyenne et variance m, σ estimées à partir de x.

On formule l'hypothèse (notée H_0 et dite hypothèse nulle) que l'échantillon x observé est issu d'un échantillonnage aléatoire au sein d'une population où la loi de la variable aléatoire est la loi normale $(m\,;\,\sigma)$ et où les tirages successifs des individus sont indépendants. On détermine alors la probabilité p d'obtenir, sous cette hypothèse nulle, un écart égal ou supérieur à celui obtenu avec l'échantillon observé. Cette étape repose de manière sous-jacente sur des résultats mathématiques de probabilité, des conditions sur les effectifs théoriques doivent être remplies pour que la méthode soit valide.

- Si cette probabilité est faible (c'est-à-dire inférieure à un seuil décidé à l'avance qui est généralement de 1% ou de 5%), on rejette l'hypothèse nulle : cela accrédite le fait que l'échantillon observé possède des particularités qui font que sa distribution n'est pas Gaussienne $(m\,;\,\sigma)$. On dit alors que la différence entre la distribution empirique et la distribution normale$(m\,;\,\sigma)$ est significative au seuil de 1% ou de 5%, on peut préciser le degré de signification p.

- Sinon, on ne rejette pas l'hypothèse nulle : les données recueillies ne permettent pas de réfuter l'adéquation de la distribution empirique avec la loi normale $(m\,;\,\sigma)$. L'écart entre la distribution observée et la distribution théorique est alors attribué aux fluctuations d'échantillonnage.

A.3 Table du χ^2

Table de Khi-Deux C^2

v / a	0.1	0.05	0.025	0.01	v / a	0.1	0.05	0.025	0.01
1	2.71	3.84	5.02	6.63	**16**	23.54	26.30	28.84	32.00
2	4.61	5.99	7.38	9.21	**17**	24.77	27.59	30.19	33.41
3	6.25	7.81	9.35	11.34	**18**	25.99	28.87	31.53	34.80
4	7.78	9.49	11.14	13.28	**19**	27.20	30.14	32.85	36.19
5	9.24	11.07	12.83	15.09	**20**	28.41	31.41	34.17	37.57
6	10.64	12.59	14.45	16.81	**21**	29.61	32.67	35.48	38.93
7	12.02	14.07	16.01	18.47	**22**	30.81	33.92	36.78	40.29
8	13.36	15.51	17.53	20.09	**23**	32.01	35.17	38.08	41.64
9	14.68	16.92	19.02	21.67	**24**	33.20	36.41	39.37	42.98
10	15.99	18.31	20.48	23.21	**25**	34.38	37.65	40.65	44.31
11	17.27	19.67	21.92	24.72	**26**	35.56	38.88	41.92	45.64
12	18.55	21.03	23.34	26.22	**27**	36.74	40.11	43.19	46.96
13	19.81	22.36	24.74	27.69	**28**	37.92	41.34	44.46	48.28
14	21.06	23.68	26.12	29.14	**29**	39.09	42.56	45.72	49.59
15	22.31	25.00	27.49	30.58	**30**	40.26	43.77	46.98	50.89

v : le nombre de degré de liberté. a : le risque d'erreur.

FIGURE A.3.1 – Table du χ^2

A.4 Test de Kolmogorov-Smirnov à deux échantillons

Le test de Kolmogorov-Smirnov à deux échantillons est un test non paramétrique qui permet de tester l'hypothèse H_0 selon laquelle deux échantillons x et y observés sont engendrées par la même loi de probabilité. Dans un contexte de comparaison de deux groupes de données, il permet de déterminer si une différence significative est identifiable dans leurs distributions.

Dans ce test, les calculs sur les lois de probabilité se font sur les fonctions de répartition : on mesure l'écart entre les fonctions de répartition empiriques F_x et F_y de x et y. On souhaite tester :

– l'hypothèse $H_0 : F_x = F_y$, contre :

– l'hypothèse $H_1 : F_x \neq F_y$.

Si $x = (x_1, ..., x_n)$ est un échantillon de longueur n, la fonction de répartition empirique $F_x(v)$ associée à cet échantillon est :

$$F_x(v) = \frac{1}{n} \sum_{k=1}^{k=n} 1_{]-\infty,v]}(x_k)$$

$F_x(v)$ est la proportion des observations dont la valeur est inférieure ou égale à v. De la même manière, on calcule F_y. L'écart entre les valeurs observées F_x et F_y peut donc être mesuré par la variable aléatoire :

$$D = \sup_{v \in \mathbb{R}} |F_x(v) - F_y(v)|$$

qui sera appelée variable de décision, ou fonction discriminante, du test. En fonction de D, on peut déterminer la probabilité p d'observer x et y sous l'hypothèse nulle H_0, et de tirer des conclusions de cette probabilité en fonction d'un seuil déterminé (en général 1% ou 5%). On montre de plus que pour $n > 100$, la fiabilité du test est assez bonne.

A.5 Table de Kolmogorov-Smirnov

Table de C de Kolomogorov-Smirnov

n / a	0.10	0.05	0.01	n / a	0.10	0.05	0.01
1	0.950	0.875	0.995	**14**	0.314	0.349	0.418
2	0.776	0.842	0.929	**15**	0.304	0.338	0.404
3	0.642	0.708	0.828	**16**	0.295	0.328	0.392
4	0.564	0.624	0.733	**17**	0.286	0.318	0.381
5	0.510	0.565	0.669	**18**	0.278	0.309	0.371
6	0.470	0.521	0.618	**19**	0.272	0.301	0.363
7	0.438	0.486	0.577	**20**	0.264	0.294	0.356
8	0.411	0.457	0.543	**25**	0.240	0.270	0.320
9	0.388	0.432	0.514	**30**	0.220	0.240	0.290
10	0.368	0.410	0.490	**35**	0.210	0.230	0.270
11	0.352	0.391	0.468	**>35**	$1.22/n^{1/2}$	$1.36/n^{1/2}$	$1.63n^{1/2}$
12	0.338	0.375	0.450				
13	0.325	0.361	1.433				

n : l'effectif total, **a**: le risque d'erreur.

FIGURE A.5.1 – Table de KOLMOGOROV-SMIRNOV

A.6 L'ANOVA à un facteur

L'analyse de variance à un facteur contrôlé ou ANOVA1 a pour objectif de tester l'effet d'un facteur A sur une variable aléatoire continue. Ceci revient à comparer les moyennes de plusieurs populations normales et de même variance à partir d'échantillons aléatoires et indépendants les uns des autres. Chaque échantillon est soumis ou correspond à une modalité du facteur A. Le terme ANOVA indique que la comparaison multiple de moyennes correspond en faite à la comparaison de deux variances.

Le facteur contrôlé A présente p modalités ($1 \leq i \leq p$). On parle aussi de niveaux ou traitements. Le nombre de répétitions j pour une modalité i est noté n_i. Le nombre de répétitions pour chaque modalité du facteur n'est pas forcément le même. La valeur prise par la variable aléatoire Y pour la modalité i du facteur et la répétition j est notée y_{ij} et les valeurs moyennes pour chaque modalité notée $\overline{y}_i$.

Les hypothèses relatives au modèle d'analyse de variances sont nombreuses. L'analyse des résidus e_{ij}, $(y_{ij} - y_i)$ est particulièrement utile pour répondre aux hypothèses de normalité et d'homoscédasticité [1]. Mais dans le cadre d'un modèle à effet fixe, il est équivalent de tester ces hypothèses sur la variable y_{ij}.

L'indépendance entre les différentes valeurs de la variable mesurée y_{ij} est une condition essentielle à la réalisation de l'analyse de variance. Les p échantillons comparés doivent donc être indépendants. L'ensemble des N individus est réparti au hasard (randomisation) entre les p modalités du facteur contrôlé A, ni individus recevant le traitement i.

L'analyse de variance à un facteur teste l'effet d'un facteur contrôlé A ayant p modalités sur les moyennes d'une variable quantitative Y. L'hypothèse nulle testée est la suivante :

il n'y a pas d'effet du facteur A et les p moyennes sont égales à une même moyenne μ.
$H_0 : \mu_1 = \mu_2 = ... = \mu_i = ... = \mu_p = \mu$
alors $y_{ij} = \mu + e_{ij}$
sous H_0 avec e_{ij} variables aléatoires indépendantes suivant une même loi normale $N(0, \sigma)$.

Les résidus e_{ij} correspondent aux fluctuations expérimentales pour chaque valeur de la variable y_{ij} mesurée.

1. On parle d'homoscédasticité lorsque la variance des erreurs stochastiques de la régression est la même pour chaque observation i (de 1 à n observations). La notion d'homoscédasticité s'oppose à celle d'hétéroscédasticité, qui correspond au cas où la variance de l'erreur des variables est différente. (Wikipédia)

Annexe B

Code source complet des scripts Python

B.1 Classe Word

```
class Word :
    """Word class, aimed at representing words, entities separated one from
    the others through a space character. Words are not intervals, because in one interval
    there can be several words. Words are not tokens either, because one token can contain
    several words (vingt deuxieme) or a word can lead to several tokens (au => Ã  le ).
    Finally, Words are not lexemes for the same reasons.
    Word is the main internal representation of the text. Word objects arranged into a tree
    can be mapped to intervals, tokens and lexemes through the adequate functions.

    Properties :
    ----------
    * string    : the raw text of the word
    * pos       : the position (usually in a sequence of words)
    * type      : the type of the word, it's a string, returned by getWordType
    * properties : a dictionnary of properties along with their values. Can be used by user.
                properties maintained by the program are currently :
                => speaker : integer giving the id of the speaker that spoke this word
                => interval : interval object when mapped to a textGrid.
    * tokens : list of tokens to which this word is mapped
    """
    string = ''
    pos = 0
    type = 'standard'


    def __init__(self, string, pos) :
        """Word constructor, parameters are string and pos"""
        string = string.strip()
        if not len(string) :
            return
        self.pos = pos
        self.string = string
        self.type = getWordType(string)
        self.properties = {}
        self.tokens=[]

    def getTokenByString(self, tokenString) :
        """getTokenByString : if self contains a token that has
        tokenString as string, returns it"""
        result = None
        for token in self.tokens :
            if tokenString == token.string : result=token
        return result

    def addToken(self, token) :
        """addToken : adds given token as a linked token"""
        self.tokens.append(token)

    def lexemes(self) :
        result = []
        for tok in self.tokens :
            result.extend(tok.lexemes)
        result = uniqu(result)

    def intervals(self) :
        if self.properties.has_key('interval') :
            return self.properties['interval']
        else :
            return []
```

```
def getWordType(word) :
  """getWordType : inputs word string, returns the type of word.
  change any  True into False if those symbols are not to be handled"""
  if True and (word in ['{','}','{|','|}','|']) :
    return word
  elif True and (word in ['//','//=']) :
    return '//'
  elif True and (word in ['//+']) :
    return word
  elif True and (word in ['ESPERLUETTE']) :
    return '&'
  elif True and (word in ['[',']']) :
    return word
  elif True and (word in ['(',')','(+']) :
    return word
  elif True and (word in ['"']) :
    return word
  elif True and (word in ['<','>','<+', '>+']) :
    return word
  elif True and (word[0] in ['#']) :
    return word[0]
  elif True and (word in ['^']) :
    return word
  elif True and (word[0 :2] in ['$L']) :
    return word[0 :2]
  elif True and (word[0 :2] in ['$-','-$']) :
    return 'overlap'
  else :
    return 'standard'

def listToText(words) :
  """listToText : takes a list of words as input and concatenate their strings to form sentence"""
  return [' '.join([w.string for w in sentence]) for sentence in words]

def handleDiscontinuities(words) :
  """handleDiscontinuities : takes a list of words and rearrange it to handle discontinuities.
  returns the new arranged list"""
  testAgain = True
  while testAgain :
    filteredWords = []
    testAgain = False
    for pos,word in enumerate(words) :
      if word.type != '#' :
        filteredWords.append(word)
      else :
        testAgain = True
        matchString = '#' + word.string
        reprisePos = pos + 1
        while words[reprisePos].string != matchString :
          reprisePos += 1
        stopPos = reprisePos + 1
        while words[stopPos].string != '//' :
          stopPos += 1

        #appending this discontinuity
        filteredWords.extend(words[reprisePos+1 :stopPos+1])

        # then the text between # and ##
        filteredWords.extend(words[pos+1 :reprisePos])

        #then the text after the ## ... //
        filteredWords.extend(words[stopPos+1 :])

        words = filteredWords
        break
  return words

def markSpeakers(words) :
  """markSpeakers : handles speaker words to mark the whole list with the adequate "speaker" property"""
  currentSpeaker = 1
  result = []
  for word in words :
    if word.type == '$L' :
      currentSpeaker = int(word.string[2 :])
    elif word.type is not 'overlap' :
      word.properties['speaker'] = currentSpeaker
      result.append(word)
  return result


def getWords(text) :
  """getWords : splits text given as input and creates the corresponding list of words"""
  symbols = ['"', '^']
  for symbol in symbols :
    text = re.sub('\\'+symbol,' ' + symbol + ' ',text)
  stringWords = text.split()
  words = []
  for pos,stringWord in enumerate(stringWords) :
    words.append(Word(stringWord, pos))
  words = handleDiscontinuities(words)
  words = markSpeakers(words)
  words = [w for w in words if w.type not in ['//+']]
```

return words

B.2 Classe Token

```
class Token :
    """Token Class, aimed at representing tokens, which are elements
    returned by a call to the parser. These elements are
    then linked to lexemes and dependencies. They are different from words,
    because a word can be transformed into several tokens by the parser, or
    several words can be merged into the same token. They are different
    from lexemes because a token can be split into several lexemes.
       During parsing of unrolled sentences, the algorithms automatically
    merges tokens that seem to be the same.

    Properties :
    * string    : the string of the token
    * lexemes   : the lexemes linked to that token
    * words     : the words linked to that token
    * uid    : the unique identifier given to that token. Is only
               set after a matchWithTextGrid
    * ids    : all the ids under which this token appears during the parsing
               of the different unrolled sentences.
               """

    string = ''
    lexemes = []
    words = []
    uid=''

    def __init__(self, string, id) :
        """Token constructor, parameters are string and id"""
        string = string.strip()
        if not len(string) :
            return
        self.string = string
        self.ids = [id]
        self.lexemes = []
        self.words=[]
        self.uid='' #set by parse

    def addId(self,id) :
        """addId : add an other id to an existing token"""
        self.ids.append(id)

    def getLexemeByForm(self, lexemeForm) :
        """getLexemeByForm : if one of the lexemes linked to self has
        form lexemeForm, returns it"""
        result = None
        for lexeme in self.lexemes :
            if lexemeForm in lexeme.forms : result=lexeme
        return result

    def addLexeme(self, lexeme) :
        """addLexeme : add lexeme to the lexemes linked to self"""
        self.lexemes.append(lexeme)

    def intervals(self) :
        """intervals : if linked words are matched with textgrid, gets
        the intervals corresponding to this Token"""
        result = []
        for w in self.words :
            result.extend(w.intervals())
        result=uniqu(result)
        return result

    def t_bounds(self) :
        """tbounds : if linked words are matched with textgrid, returns
        the start and end of boundary intervals"""
        intervals = self.intervals()
        if not len(intervals) :
            return None,None
        start = min([x.xmin() for x in intervals])
        end   = max([x.xmax() for x in intervals])
        return start,end



def findToken(id,tokenList) :
    """findToken : finds all tokens in the list tokenList that have been
    given the provided id"""
    result=[]
    for token in tokenList :
        if id in token.ids : result.append(token)
    return result
```

B.3 Classe Lexeme

```
class Lexeme :
    """Lexeme Class, aimed at representing lexemes, which are the
    fine elements returned by a call to the parser. A token can
    be split into several lexemes.
       During parsing of unrolled sentences, the algorithms automatically
    merges lexemes that are linked to tokens that seem to be the same.

    Properties :
    * forms  : the different forms of the token
    * lemmas     : the different lemmas of the token
    * ids     : all the ids under which this lexeme appears during the parsing
              of the different unrolled sentences.
    * features  : a dictionnary of all the features and their values.
              for each feature, the corresponding entry is a set of the
              different values it had.
    * lemmas     : the lexemes linked to that token
    * words      : the words linked to that token
    * uid     : the unique identifier given to that lexeme. Is only
              set after a matchWithTextGrid
    * depends   : list of the dependencies in which this lexeme is dependent
    * governs   : list of the dependencies in which this lexeme is governor
    * refTokkenIds : list of the ids of the governor of this lexeme
    * token  : token linked to this lexeme.
              """
    uid="
    ids = []
    forms = []
    lemmas = []
    features = {}
    refTokenIds = []
    token = None
    depends = []
    governs = []

    def __init__(self, lexemeNode) :
        """
        Lexeme constructor : creates a lexeme object from a XML lexeme node like :
        ---------------------------------------
        <word form="le" id="lex6" lemma="le">
                <ref idref="tok5"/>
                <features cat="det" countable="-" def="+" dem="-" det="+" gender="masc" lexid="lex6"
                nb="5" number="sg" numberposs="-" poss="-" token="au" wh="-"/>
          </word>

        """
        if lexemeNode is None :
            self.uid="
            self.forms =["]
            self.ids = ['dummyId']
            self.lemmas = ["]
            self.features = {}
            self.refTokenIds=['dummyId']
            self.token = None
            self.governs = []
            self.depends = []
            return

        self.uid=" #is set only by parse
        self.forms = [lexemeNode.getAttribute('form')]
        self.ids = [lexemeNode.getAttribute('id')]
        self.lemmas = [lexemeNode.getAttribute('lemma')]


        #find ref Token
        refNode = lexemeNode.getElementsByTagName('ref')[0]
        self.refTokenIds = [refNode.getAttribute('idref')]
        self.token = None #is set only in the parse function

        #get featureNode
        featureNode = lexemeNode.getElementsByTagName('features')[0]

        #build features dictionary
        Attr =  featureNode.attributes
        self.features={}
        for attrPos in range(Attr.length) :
            self.features[Attr.item(attrPos).name]=[Attr.item(attrPos).value]

        self.depends=[]
        self.governs=[]

    def mergeWith(self,lexeme) :
        """mergeWith : merges self with another lexeme"""
        #puts all content of lexeme into self. Also merges features
        self.forms.extend([x for x in lexeme.forms if x not in self.forms])
        self.ids.extend([x for x in lexeme.ids if x not in self.ids])
        self.lemmas.extend([x for x in lexeme.lemmas if x not in self.lemmas])
        keys=lexeme.features.keys()
        for key in keys :
            if self.features.has_key(key) :
```

```
        self.features[key].extend(lexeme.features[key])
      else :
        self.features[key] = lexeme.features[key]

def words(self) :
  return self.token.words

def t__bounds(self) :
  """tbounds : returns t_bound of linked token"""
  if self.token is None :return None
  return self.token.t__bounds()

def str(self) :
  return self.uid+'('+'|'.join(self.forms)+')'

def prettyPrint(self) :
  print self.str()
  print '  => features:',self.features
  print '  => governs :'
  for govdep in self.governs :
    print '      * '+govdep.str()
  print '  => depends :'
  for depdep in self.depends :
    print '      * '+depdep.str()
```

B.4 Classe Dependancy

```
class Dependency :
  """Dependency Class, aimed at representing dependencies between
  lexemes as returned by Eric's parser.
     A dependency is characterized by its governor and dependent
  lexemes, its function and id.

  Properties :
  * govLexeme : governor lexeme object
  * depLexeme : dependent lexeme object
  * func      : function of the dependency. It's a string
  * uid       : unique identifier given to this dependency
  """
  govLexeme=None
  depLexeme=None
  func="
  uid="

  def ___init___(self, gov, dep,func) :
    """Dependency constructor"""
    self.govLexeme = gov
    self.depLexeme = dep
    self.func=func
    self.uid="#set only by parse
  def isequal(self,dep) :
    """isequal : checks whether two dependencies have the same
    func, governor and dependent lexemes. Returns False if it is
    not the case"""
    if (     (self.govLexeme is dep.govLexeme)
         and (self.depLexeme is dep.depLexeme)
         and (self.func == dep.func)) :
      return True
    else :
      return False

  def str(self) :
    govstr='root'
    if self.govLexeme is not None :
      govstr=self.govLexeme.str()
    depstr = self.depLexeme.str()
    return '['+self.uid+':'+self.func+'='+govstr+'=>' +depstr+']'
```

B.5 Classe Node

```
class Node :
    """Node class. This class is aimed at representing the base structure
    of the tree nodes. A tree node is characterized mainly through
    the Words it contains, its (unique) father node and its several
    sons (Nodes). Appart from these structural information, a Node also
    has some properties such as its type (string).
        The tree structure is build from a texwt or a list of Nodes
    through the use  of the buildTree function.
        Given the tree structure, all kind of operations can be performed.
    the Node class provides many functions designed to manipulate a tree.
        Generally, the functions can have an impact on the whole tree
    structure and more particularly on the downstream tree, i.e. all the
    nodes that have self as an ancestor. Likewise, many search functions
    have been implemented to look for particular information in the tree,
    or to rearrange the tree to merge together some nodes.
        See the member methods for details.

    Properties :
    -----
    * content   : list of Words objects directly contained by this Node
    * sons      : list of Nodes objects that are the sons of this Node
    * father : father Node object (can be None if this is a root node)
    * type      : a string giving the type of the Node
    * marks     : a dictionnary giving some particular properties of the Node
                can be used by user. The properties used by current
                implementation are :
                => annotation : string, when  a tree has been build using
                  buildTree, the root Node is marked with the text used
                  as input to buildTree
                => matchedWithTextGrid : boolean. when the tree has been
                   matched to a TextGrid, is True.
                => pivotTier : tier object (see TextGrid.py) : when the
                  tree has been matched to a TextGrid, root Node is given this
                  property.
                => integrated : used for some particular Node types
                => interrupted : used when parsing piles
                => parsed : when parse has been successfully call, sets to True
    """
    content= []
    sons = []
    type = 'root'
    father = None

    def __init__(self,father = None, type='root') :
        """Node constructor"""
        self.type = type
        self.content = []
        self.sons = []
        self.father = father
        self.marks = {}

    def createSon(self, type) :
        """createSon : create a new son and append it to self.sons"""
        newSon = Node(self,type)
        self.sons.append(newSon)
        return newSon

    def ancestors(self) :
        """ancestors : returns a list of all ancestors of self"""
        result = []
        current = self
        while current.father is not None :
            current=current.father
            result.append(current)
        return result

    def detachIfEmpty(self) :
        """detachIfEmpty : checks whether this son has no content and no son.
        If this is the case, detach it"""
        if (not len(self.sons)) and (not len(self.content)) :
            self.detach()
            return True
        return False

    def intervalsInTier(self, tier,strict=True,recursive=True) :
        intervals = self.intervals(recursive)
        if not len(intervals) :
            return []
        eps = 1E-5

        #aggregating intervals into time segments
        startSlices = []
        stopSlices = []
        for interval in intervals :
            start = interval.xmin()
            stop = interval.xmax()
            startMatch = [pos for pos,x in enumerate(stopSlices) if abs(start - x) < eps]
            stopMatch  = [pos for pos,x in enumerate(startSlices) if abs(stop - x) < eps]
```

```
      if len(startMatch) or len(stopMatch) :
         for match in startMatch :
            stopSlices[match] = stop
         for match in stopMatch :
            startSlices[match] = start
         continue
      startSlices.append(start)
      stopSlices.append(stop)

   #now finding intervals in tier which are contained by at least one of the slices,
   #either strictly are partly
   matching = []
   for interval in tier :
      start = interval.xmin()
      stop = interval.xmax()
      found = False
      for (startslice,stopslice) in zip(startSlices,stopSlices) :
         if strict :
            if ( (start >= startslice) and (stop <= stopslice) ) :
               found = True
         else :
            if (  ( (start < stopSlice)   and (stop > stopSlice  ) )
                        or( (start < startSlice) and (stop > startSlice) ) ) :
               found = True
      if found :
         matching.append(interval)

   """start = min([i.xmin() for i in intervals])
   end = max([i.xmax() for i in intervals])
   if strict :
      matching = [i for i in tier if ( (i.xmin() >= start) and (i.xmax() <= end) )]
   else :
      matching = [i for i in tier if (  ( (i.xmin() < end)   and (i.xmax() > end  ) )
                        or( (i.xmin() < start) and (i.xmax() > start) ) )]"""
   return matching


def clean(self) :
   """clean : clean downstream tree. detach all Nodes whose downstream
   tree does not contain any Word object"""
   processAgain = True
   while processAgain :
      processAgain = False
      for node in self.nodes() :
         processAgain = processAgain or node.detachIfEmpty()

def detach(self) :
   """detach : moves content and sons of self to its father."""
   father = self.father
   self.father = None
   if father is None :
      return
   father.sons = [son for son in father.sons if son is not self]


def mergeUntil(self, others, stopTypes) :
   """mergeUntil : merge all nodes in others and their downstream tree with self until
   nodes of a type in stopTypes is encountered."""
   #for each of the Nodes to merge with self
   for other in others :
      if other.type in stopTypes :
         #simple case : it is of a stoptype : simply attach it
         other.attach(self)
         continue

      #on the contrary case
      # first add its content to self
      self.content.extend(other.content)

      #and, for each of its sons
      for son in other.sons :
         if other.type not in stopTypes :
            #if it is not of a stop type, then look whether
            #this type is already represented. If yes, merge, if
            #no, attach
            siblings= self.getSonsOfTypes([son.type])
            if not len(siblings) :
               son.attach(self)
            else : siblings[0].mergeUntil([son],stopTypes)
         else :
            son.attach(self)


def detachSons(self) :
   """detachSons : detach all sons of self"""
   sons = self.sons[ :]
   for son in sons :
      son.detach()
   return sons

def attach(self,father) :
   """attach : sets father as the new father of self and add self to father.sons"""
```

```
    self.detach()
    self.father = father
    if father is not None :
      father.sons.append(self)

  def getSonsOfTypes(self,types,recursive=False) :
    """getSonsofType : gets all sons whose type is in types"""
    filteredSons = []
    if not recursive :
      toFilter = self.sons
    else :
      toFilter = self.nodes(includeSelf=False)
    for son in toFilter :
      if son.type in types :
        filteredSons.append(son)
    return filteredSons

  def mark(self,string,data=True) :
    """mark : appends entry string with value data to self.marks"""
    self.marks[string] = data

  def unmark(self,string) :
    """unmark : removes string from self.marks if it exists"""
    if self.marks.has_key(string) :
      del self.marks[string]

  def ismarked(self, string) :
    """ismarked : returns True if self has a string entry in its marks """
    if self.marks.has_key(string) :
      return True
    else :
      return False

  def getSonsNotOfTypes(self,types) :
    """getSonsNotOfTypes : gets all sons of self whose type is not in types"""
    filteredSons = []
    for son in self.sons :
      if son.type not in types :
        filteredSons.append(son)
    return filteredSons

  def getContainerOf(self, words) :
    """getContainerOf : returns the Node in the downstream tree of self
    that directly contains all nodes in words. None if no such Node exists"""
    result=None
    if all([x in self.content for x in words]) :
      return self
    else :
      for son in self.sons :
        result = son.getContainerOf(words)
        if result is not None : break
      return result

  def buildRhapsodieXML(self,wordsList=None) :
    """buildRhapsodieXML : Builds complete XML data for Rhapsodie based on that tree
    """
    if not self.ismarked('parsed') :
      print 'Tree must be parsed. Doing nothing.'
      return

    #create main document
    doc=Document()

    #create its sample element
    sample=doc.createElement("sample")
    sample.setAttribute("style","Rhapsodie")
    doc.appendChild(sample)

    #create text element with annotation and add it to sample
    text=doc.createElement("markup_text")
    text.appendChild(doc.createTextNode(self.marks['annotation']))
    sample.appendChild(text)

    if self.ismarked('matchedWithTextGrid') :
      #create words element with type pivot and append all intervals to it
      pivot=doc.createElement("words")
      pivot.setAttribute("type","pivot")
      for interval in self.marks['pivotTier'] :
        #create an interval node and append it
        intervalNode = doc.createElement("word")
        intervalNode.setAttribute("id",interval.uid)
        intervalNode.setAttribute("xmin",str(interval.xmin()))
        intervalNode.setAttribute("xmax",str(interval.xmax()))
        intervalNode.setAttribute("shape",'empty')
        intervalNode.setAttribute("localreg",'empty')
        intervalNode.setAttribute("orthotext ",interval.mark())
        pivot.appendChild(intervalNode)
      sample.appendChild(pivot)

    #create words element with type token and append all tokens to it
    tokens=doc.createElement("words")
    tokens.setAttribute("type","tokens")
```

```
for token in self.tokens() :
    #create a token node
    tokenNode = doc.createElement("word")
    tokenNode.setAttribute('id',token.uid)
    tokenNode.setAttribute('orthotext ',token.string)
    tokenNode.setAttribute("shape",'empty')
    tokenNode.setAttribute("localreg",'empty')

    #for each token, also add start and end attributes
    #because Nicolas and Arthur want it.
    #still, one has to be careful of the fact that tokens
    #can have discontinuous temporal support!
    start,end= token.t__bounds()
    if start is not None :
        tokenNode.setAttribute("start",str(start))
    if end is not None :
        tokenNode.setAttribute("end",str(end))

    if self.ismarked('matchedWithTextGrid') :
        #identifies the intervals to which this token is
        #linked through the words to which it is linked
        refIntervals = []
        for refWord in token.words :
            if refWord.type is not 'standard' :
                continue
            refIntervals.extend(refWord.properties['interval'])
        refIntervals = uniqu(refIntervals)
    else :
        refIntervals=[]

    for refInterval in refIntervals :
        refIntervalNode = doc.createElement("ref")
        refIntervalNode.setAttribute("idref",refInterval.uid)
        tokenNode.appendChild(refIntervalNode)

    tokens.appendChild(tokenNode)
sample.appendChild(tokens)

#create words element with type lexemes and append all lexemes to it
lexemes=doc.createElement("words")
lexemes.setAttribute("type","lexemes")

tagset=doc.createElement("tagset")
tagset.setAttribute("tag","cat")
tagset.setAttribute("name","cat")
lexemes.appendChild(tagset)

for lexeme in sorted(self.lexemes(), key=lambda lex : max([w.pos for w in lex.words()]) ) :
    if (len(lexeme.forms) > 1) or (len(lexeme.lemmas)>1) :
        print 'There may be a problem due to a lexeme having many forms or lemmas! : %s'%' '.join(lexeme.forms)

    #create a lexeme node
    lexemeNode = doc.createElement("word")
    lexemeNode.setAttribute('id',lexeme.uid)
    lexemeNode.setAttribute('orthotext ',lexeme.forms[0])
    lexemeNode.setAttribute('lemma',lexeme.lemmas[0])
    lexemeNode.setAttribute("shape",'empty')
    lexemeNode.setAttribute("localreg",'empty')

    #for each lexeme, also add start and end attributes
    #because Nicolas and Arthur want it.
    #still, one has to be careful of the fact that lexemes
    #can have discontinuous temporal support!
    start,end= lexeme.t__bounds()
    if start is not None :
        lexemeNode.setAttribute("start",str(start))
    if end is not None :
        lexemeNode.setAttribute("end",str(end))

    #append a ref node to it for its ref token
    refTokenNode = doc.createElement("ref")
    refTokenNode.setAttribute("idref",lexeme.token.uid)
    lexemeNode.appendChild(refTokenNode)

    #append a feature node to it for its features
    featuresNode = doc.createElement("features")
    for feature in lexeme.features.keys() :
        if feature == 'lexid' :
            chosenValue = lexeme.uid
        else :
            #gets unique elements keeping order
            uniqueElements = uniqu(lexeme.features[feature][ :])

            if len(uniqueElements) > 1 :
                #specify that this lexeme has a feature having several values
                featuresNode.setAttribute('attention', '+')
                featuresNode.setAttribute('attentionFeature', feature)

                countVector = [lexeme.features[feature].count(value) for value in uniqueElements]
                #si plusieurs valeurs pour une feature donnee, on prend la derniere des valeurs qui apparaissent le plus
                maxi = max(countVector)
                uniqueElements=[x for k, x in enumerate(uniqueElements) if countVector[k]==maxi]
```

```
            chosenValue = uniqueElements[-1]

          print '       There are %d values for feature %s for lexeme %s!'%(len(uniqueElements),feature,lexeme.uid)
            for k,value in enumerate(uniqueElements) :
              print '              value %s'%value,countVector[k]
          else :
            chosenValue = uniqueElements[0]

        featuresNode.setAttribute(feature, chosenValue)
      lexemeNode.appendChild(featuresNode)

      lexemes.appendChild(lexemeNode)

    sample.appendChild(lexemes)

    #create dependencies element with type syntax and append
    #dependencies per UI to it
    dependencies=doc.createElement("dependencies")
    dependencies.setAttribute("type","syntax")
    tagset=doc.createElement("tagset")
    tagset.setAttribute("tag","func")
    tagset.setAttribute("name","syntacticfunctions")
    dependencies.appendChild(tagset)

    for k,ui in enumerate(self.sons) :
      #create a dependancy node for this ui
      dependencyNode = doc.createElement("dependency")
      dependencyNode.setAttribute('id','dep%d'%k)

      #add as an attribute the complete annotation for this iu
      if k == len(self.sons)-1 :
        endPos = max([w.pos for w in ui.words()])
      else :
        endPos = self.sons[k+1].startPos
      iuCompleteText = ' '.join([ w.string for w in wordsList[ui.startPos :endPos] ])
      dependencyNode.setAttribute('markupIU',iuCompleteText)

      #for each dependency within this ui, append it as child
      for dependency in ui.dependencies() :
        linkNode = doc.createElement("link")
        linkNode.setAttribute('depid',dependency.depLexeme.uid)

        #linkNode.setAttribute('juncSTRING',dependency.depLexeme.token.string)#XXXX
        linkNode.setAttribute('func',dependency.func)
        if dependency.govLexeme is not None :
          linkNode.setAttribute('govid',dependency.govLexeme.uid)
          #linkNode.setAttribute('gov__STRING',dependency.govLexeme.token.string)#XXXX
        linkNode.setAttribute('id',dependency.uid)
        dependencyNode.appendChild(linkNode)
      dependencies.appendChild(dependencyNode)
    sample.appendChild(dependencies)

    #pile structure
    pilesDependenciesNode = doc.createElement("constrees")
    pilesDependenciesNode.setAttribute("type","pile_tree")
    pileTree = self.copy()
    pileTree.pileConversion()
    pileTree.prettyPrint()
    for k,pile in enumerate(pileTree.sons[ :]) :
      #create a dependancy node for this ui
      pileDependencyNode = doc.createElement("constree")
      pileDependencyNode.setAttribute('type','pile')
      pileDependencyNode.setAttribute('id','pile%d'%k)
      pile.XMLHierarchyPrint(doc,pileDependencyNode)
      pilesDependenciesNode.appendChild(pileDependencyNode)

    sample.appendChild(pilesDependenciesNode)

    #topological structure
    topoDependenciesNode = doc.createElement("constrees")
    topoDependenciesNode.setAttribute("type","topology_tree")
    topoTree = self.copy()
    topoTree.topologicalConversion()
    for k,ui in enumerate(topoTree.sons) :
      #create a dependancy node for this ui
      topoDependencyNode = doc.createElement("constree")
      topoDependencyNode.setAttribute('type',"topology")
      topoDependencyNode.setAttribute('id','topo%d'%k)

      discParams = [ {'types' :['iu'],  'provokedBy' :
                              {'parenthesis' :'inserting',
                              'graft' :'embedded',
                              'inkernel' :'inserting'} },]

      uiXMLNode = ui.XMLHierarchyPrint(doc,topoDependencyNode,discontinuities=discParams)
      topoDependenciesNode.appendChild(topoDependencyNode)
    sample.appendChild(topoDependenciesNode)


    return doc
```

```
def XMLHierarchyPrint(self,document = None, father=None,noContentWriteforTypes=[],discontinuities = []) :
    """XMLHierarchyPrint : writes to the given XML document the Node.
    the content of words of a type in noContentWriteforTypes is not included.
    discontinuities is a list of dictionnaries of the type :
    discontinuities = [  {'types' :['iu'],'provokedBy' :{'parenthesis' :'insert','graft' :'graft','embedded' :'embedded'} },
                  {'types' :['prekernel','postkernel','kernel','intro'],'provokedBy' :{'inkernel' :'inkernel'} }]
    information"""
    if document is None :
        document=Document()
        father = document

    myEntry = document.createElement("const")
    myEntry.setAttribute("ctype",self.type)

    #handling of the discontinuities
    if len(discontinuities) :
        downstream = self.nodes()
        downstream.remove(self)
        downstreamTypes = [n.type for n in downstream]
        for disc in discontinuities :
            if self.type in disc['types'] :
                matched = ''
                for key in disc['provokedBy'].keys() :
                    if len([t for t in downstreamTypes if t is key]) :
                        matched+='+'+disc['provokedBy'][key]
                matched=matched.lstrip('+')
                if len(matched) :
                    myEntry.setAttribute("complexity",'True')
                    myEntry.setAttribute("complexitytype",matched)
                else :
                    myEntry.setAttribute("complexity",'False')
    #xmin xmax stuff
    intervals = self.intervals()
    if intervals :
        xmin = min([x.xmin() for x in intervals])
        xmax = max([x.xmax() for x in intervals])
    else :
        xmin = None
        xmax = None

    #for dr. Obin
    myEntry.setAttribute("start",str(xmin))
    myEntry.setAttribute("end",str(xmax))
    myEntry.setAttribute("shape",'empty')
    myEntry.setAttribute("localreg",'empty')

    #done for header
    father.appendChild(myEntry)

    #if this type is not in noContentWrite, writes its content
    if self.type not in noContentWriteforTypes :
        for lexeme in self.lexemes(recursive=False) :
            #add each of its lexemes
            newContent = document.createElement('const')
            newContent.setAttribute("ctype","lexeme")
            newContent.setAttribute("idref",lexeme.uid)
            myEntry.appendChild(newContent)

    #for all its sons, do the same thing, sorted in chronological order
    for son in [sorted(self.sons, key=lambda n : n.getSpan()[0])] :
        son.XMLHierarchyPrint(document=document,father=myEntry,discontinuities=discontinuities)
    return myEntry

def prettyPrint(self,level = 0, focusNode=None) :
    """prettyPrint : displays downstream tree"""
    margin = level*'  '
    contentString = ''
    for w in self.content :
        contentString += w.string
        """if w.properties.has_key('interval') :
            contentString+='(%s)'% ','.join([x.uid for x in w.properties['interval']])"""
        if w.tokens is not None :
            contentString+='('+'|'.join([t.uid for t in w.tokens])+')'
        contentString+=' '
    focusString=''
    if focusNode is self :
        focusString='* '
    print margin + '|=> ' + focusString + self.type + ': '+contentString
    for son in self.sons :
        son.prettyPrint(level+1,focusNode)

def getSpan(self) :
    """getSpan : gets the min and max pos of the Nodes downstream self"""
    begin =  1000000
    end = - 1000000
    for element in self.content :
        begin = min(begin, element.pos)
        end = max(end, element.pos)
    for son in self.sons :
        beginSon, endSon = son.getSpan()
        begin = min(begin, beginSon)
        end = max(end,endSon)
```

```
    return begin, end

  def unroll(self) :
    """unroll : returns a list whose elements are the different unrolled sentences
    (lists of words) produced by the tree"""

    #utilitary functions
    def addLists(A,B) :
      #returns [a1 a2 ... an b1 b2 .. bn]
      C=[]
      for a in A :
        if len(a) :
          C.extend([a])
      for b in B :
        if len(b) :
          C.extend([b])
      return C

    def multiplyLists(A,B) :
      #returns [a1 b1 a1 b2 ... a1 bn a2 b1  ... a2 bn ... an b1  ... an bn]
      if not len(B) :
        return A
      C=[]
      for a in A :
        for b in B :
          temp = []
          temp.extend(a)
          temp.extend(b)
          C.extend([temp])
      return C

    absolute = []
    relative = [self.content]
    integrationComaPos = {  'integratedprekernel' :[1],
                  'integratedpostkernel' :[0],
                  'integratedinkernel' :[0, 1],}
    for son in self.sons :
      sonAbs, sonRel = son.unroll()
      absolute=addLists(absolute,sonAbs)
      if son.type == 'layer' :
        #in case of layers, there is a disjunction
        relative = addLists(relative,sonRel)
      elif son.type in ['graft'] :
        #replace relative with XXX and add it to absolute
        newWord = Word('xxx',sum(son.getSpan())/2.0)
        relative = multiplyLists(relative, [[newWord]])
        absolute=addLists(absolute,sonRel)
      elif son.type in ['iu', 'parenthesis','inkernel','prekernel','postkernel','discursivemarker'] :
        absolute=addLists(absolute,sonRel)
        """elif son.type=='intro' :
          #in case of intro, send unrolled content to absolute
          absolute=addLists(absolute,sonRel)"""
      elif son.type in ['integratedprekernel','integratedpostkernel','integratedinkernel'] :
        #insert comas at desired positions
        newWords = []
        for comaPos in integrationComaPos[son.type] :
          #for 0 (left) : before word (= +0) for 1 (right) : after word (+1)
          newWords.append(Word(',',son.getSpan()[comaPos]+comaPos))
        relative = multiplyLists(relative, [newWords])
        relative = multiplyLists(relative, sonRel)
      else :
        relative = multiplyLists(relative, sonRel)


    if self.type == 'root' :
      absolute=addLists(absolute,relative)
      return [sorted(x, key=lambda word : word.pos) for x in absolute]
    else :
      return absolute, relative

  def exclude(self) :
    """detach : detach self and attach its sons and content to its father"""
    father = self.father
    if father is None : return
    self.detach()
    father.content.extend(self.content)
    for son in self.sons :
      son.attach(father)

  def accessibleNodes(self, stopTypes,includeStopNodes=True) :
    """accessibleNode : get nodes that are accessible through a direct pass that does
    not includes nodes of a type included in stopTypes """
    nodes = []
    for son in self.sons :
      if son.type not in stopTypes :
        nodes.append(son)
        nodes.extend(son.accessibleNodes(stopTypes))
      elif includeStopNodes : nodes.append(son)
    return nodes

  def filterHierarchiesOf(self,types,by) :
    """filterHierarchiesOf : merges all downstream Nodes of self of a type in types
```

```
    until Nodes of a type in by is encountered"""
    goOnFiltering=True
    while goOnFiltering :
      goOnFiltering=False
      if self.type in types :
        candidates = [x for x in self.accessibleNodes(by) if x.type in types]
        for candidate in candidates :
          goOnFiltering=True
          candidate.exclude()
          break
      else :
        candidates = [x for x in self.accessibleNodes(by) if x.type in types]
        for candidate in candidates :
          sonsToExclude = [x for x in candidate.accessibleNodes(by,False) ]
          for toExclude in sonsToExclude :
            goOnFiltering=True
            toExclude.exclude()

          if goOnFiltering :
            break
  def absorb(self, others) :
    for other in others :
      self.content.extend(other.content)
      for son in other.sons :
        son.attach(self)

  def groupby(self, what,by) :
    """groupby : group nodes belonging to a type in the list what by any type in the list by   """
    #building list of types that will be kept
    typesToKeep=[x for x in what]
    typesToKeep.extend(by)

    #exclude accessible nodes that are not of a type to keep
    goOnExclusions = True
    while goOnExclusions :
      goOnExclusions = False

      #get accessibles nodes to exclude
      toExclude = [x for x in self.accessibleNodes(by) if x.type not in typesToKeep]
      if len(toExclude) :
        goOnExclusions=True
        fatherOfExcluded = toExclude[0].father
        toExclude[0].detach()
        fatherOfExcluded.content.extend(toExclude[0].content)
        fatherOfExcluded.mergeUntil(toExclude[0].detachSons(),by)

    #now, forbid hierarchies of nodes of type in what by nodes of types in by
    self.filterHierarchiesOf(what,by)

    #identify stopping nodes and detach them :
    accessibles = self.accessibleNodes(by) ;
    stopBranches = [x for x in accessibles if (x.type in by) and (x is not self)]
    for stopNode in stopBranches :
      stopNode.detach()

    #detach all accessible nodes :
    for node in accessibles :
      node.detach()

    #group accessible nodes by type
    for currentType in what :
      grouped=[x for x in accessibles if x.type==currentType]
      if len(grouped) :
        typedSon = self.createSon(currentType)
        #typedSon.absorb(others=grouped)

        #add content and sons of each element of others to typedSon
        for other in grouped :
          typedSon.content.extend(other.content)
          for son in other.sons :
            son.attach(typedSon)

    #for each stop branch,
    for branchNode in stopBranches :
      #attach it back to currentNode
      branchNode.attach(self)

      #then remove its sons of excluded type
      excludedSons=branchNode.getSonsNotOfTypes(typesToKeep)
      while len(excludedSons) :
        for excludedSon in excludedSons :
          #for each such son, detach it and put its content
          #to the branch Node
          excludedSon.detach()
          branchNode.content.extend(excludedSon.content)

          #then merge branch node with his sons until stop types are met
          sons = excludedSon.detachSons()
          branchNode.mergeUntil(sons,by)
        excludedSons=branchNode.getSonsNotOfTypes(typesToKeep)

      #now branch node is guaranteed to have no son of a type
```

```
        #to exclude. Then call groupby on each of its sons
        for son in branchNode.sons :
            son.groupby(what,by)

    #now, forbid again hierarchies of nodes of type in what by nodes of types in what
    self.filterHierarchiesOf(what,by)

def nodes(self,includeSelf=True,) :
    """nodes : return a list of all the nodes that have self as an ancestor."""
    if includeSelf :
        result =  [self]
    else :
        result=[]
    for son in self.sons :
        result += son.nodes(includeSelf=True)
    return result

def words(self,recursive=True) :
    """words : return a list of all the words :
    recursi=True : whose containing node have self as an ancestor
    recursive=False : whose containing node is self"""

    result =  [x for x in self.content]
    if recursive :
        for son in self.sons :
            result.extend(son.words())
    return result

def tokens(self,recursive=True) :
    """tokens : return a list of all the tokens :
    recursive=True : that are linked to at least a node that has self as an ancestor
    recursive=False : that are linked to self"""
    tok = []
    for w in self.words(recursive) :tok.extend(w.tokens)
    return uniqu(tok)

def lexemes(self,recursive=True) :
    """lexemes : return a list of all the lexemes :
    recursive=True : that are linked to at least a node that has self as an ancestor
    recursive=False : that are linked to self"""
    lex = []
    for tok in self.tokens(recursive) :lex.extend(tok.lexemes)
    return uniqu(lex)

def intervals(self, recursive=True) :
    """intervals : return a list of all the intervals :
    recursive=True : that are linked to at least a node that has self as an ancestor
    recursive=False : that are linked to self"""
    ints = []
    for word in self.words(recursive) :
        if word.properties.has_key('interval') :
            ints.extend(word.properties['interval'])
    return uniqu(ints)

def dependencies(self,recursive=True) :
    """dependencies : return a list of all the dependencies :
    recursive=True : that involve at least a node that has self as an ancestor
    recursive=False : that involve self"""

    #the depends list is enough because any dependency has a depending lexeme (contrarily to governing,
    #which may be None)
    dep = []
    for lex in self.lexemes(recursive) :dep.extend(lex.depends)
    return uniqu(dep)

def copy(self) :
    """copy : copies the current hierarchy downstream from self and returns it"""
    result = Node(father = None, type = self.type)
    result.content = self.content[ :]
    result.marks = self.marks.copy()
    for son in self.sons[ :] :
        son.copy().attach(result)
    return result

def remove(self, word,recursive=True) :
    """remove : removes the given word from self and if recursive=True, from all its sons"""
    if word in self.content :
        self.content.remove(word)
    if recursive :
        for son in self.sons :
            son.remove(word,recursive)

def removeWordsOfType(self, wordType,recursive=True) :
    """remove : removes the words of a given type from self and if recursive=True, from all its sons"""
    toRemove = [w for w in self.content if w.type is wordType]
    for word in toRemove :
        self.content.remove(word)
    if recursive :
        for son in self.sons :
            son.removeWordsOfType(wordType,recursive)
```

```
  def log(self,inputText,currentNode,log) :
    """log : outputs tree and waits for user input"""
    print '\n'*500
    print "
    print inputText
    print '-----------------------------------'
    print 'Current Tree:'
    self.prettyPrint(focusNode=currentNode)
    print "
    print log
    raw_input()

  def topologicalConversion(self) :
    """topologicalConversion : alias for a call to groupby to obtain "topological trees" """
    print 'starting topological conversion'
    topologicalTypes = ['prekernel',
                'kernel',
                'postkernel',
                'inkernel',
                'integratedprekernel',
                'integratedpostkernel',
                'integratedinkernel',
                'intro']
    separators = ['iu','embedded','graft','parenthesis','discursivemarker']
    """topologicalTypes = [
                'kernel',

                'inkernel']

    separators = ['iu','embedded','graft','parenthesis','discursivemarker','prekernel','postkernel', 'integrated-
prekernel',
                'integratedpostkernel',
                'integratedinkernel',
                'intro']
    """
    self.groupby(topologicalTypes,separators)
    print 'topologicalConversion'
    #removing & words for topological conversion
    self.removeWordsOfType('&')

  def pileConversion(self) :
    """pileConversion : alias for a call to groupby to obtain a "pile tree" """
    print 'starting topological conversion'
    self.groupby([],['layer','pile','iu'])
    self.groupby(['layer'],['pile'])
    #removing & words for topological conversion
    self.removeWordsOfType('&')


  def filterTextGrid(self, intervals) :
    if not self.ismarked('matchedWithTextGrid') :
      return None
    allIntervals = self.intervals()
    grid = tg.TextGrid()
    print self.marks['pivotTier']
    tier = tg.IntervalTier('piles',self.marks['pivotTier'].xmin(),self.marks['pivotTier'].xmax())
    #tier = tg.IntervalTier(self.marks['pivotTier'].name(),self.marks['pivotTier'].xmin(),self.marks['pivotTier'].xmax())
    for interval in intervals :
      tier.append(interval)
    grid.append(tier)
    return grid

  def matchTextGrid(self, fileName, tierName) :
    """matchTextGrid : matchs word strings with textGrid and adds the corresponding
    intervals as property 'interval' of words"""
    words = sorted([w for w in self.words() if w.type is 'standard'], key=lambda word : word.pos)

    grid = tg.TextGrid()
    grid.read(fileName)

    tierPos = [x for x in range(len(grid)) if grid[x].name()==tierName]
    if not len(tierPos) :
      error('tier not found!')
    tierPos = tierPos[0]

    intervalTextsOriginal = [x.mark() for x in grid[tierPos]]
    intervalTexts = intervalTextsOriginal[ :]

    currentPos = 0
    for k, word  in enumerate(words) :
      if word.type is not 'standard' :
        continue
      filteredString = re.sub("'|-|\/\/\+|&",' ',word.string).split()
      for label in filteredString :
        found = False
        currentPosBegin = currentPos
        while not found :
          if (currentPos<len(intervalTexts)) and (label in intervalTexts[currentPos]) :
            interval = grid[tierPos][currentPos]
            if (word.properties.has_key('interval')) and interval not in word.properties['interval'] :
              word.properties['interval'].append(interval)
            else :
```

```
                word.properties['interval']=[interval]
              intervalTexts[currentPos] = re.sub(label,'',intervalTexts[currentPos],count=1)
              found = True
            else :
              currentPos+=1
              if currentPos - currentPosBegin > 5 :
                print '----------------------------------------'
                print 'Error while looking for a correspondance for word %s'%word.string
            print '   Context in annotation is : %s' % ' '.join([x.string for x in words[max(0,k-5) :min(len(words),k+5)]
])
                        print '   I Began looking for this word at TextGrid position %d :  : %s' % (currentPosBegin, '
'.join(intervalTextsOriginal[max(currentPosBegin-5,1) :currentPos+1]))
                print '----------------------------------------'
                raise error

    #adding unique identifiers to interval objects for Rhapsodie
    for k,interval in enumerate(grid[tierPos]) :
      interval.uid='piv%d'%k

    self.mark('pivotTier',grid[tierPos])
    self.mark('TextGrid',grid)
    self.mark('matchedWithTextGrid')
    print 'Tree matched with textgrid'


  def parse(self) :
    """ parse : Unroll the tree, parses all sentences through Eric's parser,
    then recovers filtered parser's output through Kim's scripts via XML file.
    Puts back all tokens/lexemes/dependencies into the tree
    """
    import rhapsodieEricParsing as eric

    sentences = self.unroll()

    unrolledFileName = 'unrolled_sentences'

    tokens=set()
    lexemes = []
    dependencies=[]

    #Parsing for each unrolled sentence
    for sentence in sentences :
      #1Â°) Get eric parsing result and its filtered version
      #--------------------------
      #word as strings in the sentence
      sentenceStrings = [x.string for x in sentence]

      #writing sentence as a string in file
      secureWrite(listToText([sentence]),unrolledFileName)

      #calling eric, and Kim's code for filtering it
      outxml = eric.parseSentenceFile(unrolledFileName)

      #parsing Kim's output xml file
      doc = minidom.parse(outxml)

      #get "pivot","tokens" and "lexemes" sections
      wordsSections = doc.getElementsByTagName("words")

      #2Â°) For all tokens, identify words linked to them. Create Token
      # objects if they do not exist
      #--------------------------

      #tokens in XML
      tokensObjectsInXML = wordsSections[1].getElementsByTagName("word")
      tokensStringsInXML = [x.getAttribute('text') for x in tokensObjectsInXML]
      tokensIdsInXML = [x.getAttribute('id') for x in tokensObjectsInXML]

      #identify which words match the tokens
      testSymbols = tokensStringsInXML[ :]
      sourceSymbols = sentenceStrings[ :]
      currentPosInSource = 0
      correspondances=[]
      for test in testSymbols :
        test=re.sub("'|-|\?|\.|\:|\!|\,|\;",' ',test)
        currentCorrespondances=[]
        splitted = test.split()
        #print '|'.join(testSymbols)
        for current in splitted :
          current=current.strip('_')
          found = False
          while not found :
            #print '|'+current.lower()+'|',sourceSymbols, currentPosInSource
            if current.lower() in sourceSymbols[currentPosInSource].lower() :
              #print 'found!'
              #sourceSymbols[currentPosInSource]=re.sub(current,'',sourceSymbols[currentPosInSource])
              currentCorrespondances.append(currentPosInSource)
              found = True
            else :
              currentPosInSource+=1
        correspondances.append(currentCorrespondances)
```

```
    #for each token,
    for tokenPos,corr in enumerate(correspondances) :
      #there will be a token object, but it may not be a new one (in case of piles)
      newToken = None

      #look whether the corresponding word(s) already point to a token having the same string.
      #ASSUMPTION (!!) is that this word will not lead to tokens having the same string and
      #needing to be handled differently (seems reasonable)
      for wordPos in corr :
        newToken = sentence[wordPos].getTokenByString(tokensStringsInXML[tokenPos])

      if newToken is None :
        newToken = Token(tokensStringsInXML[tokenPos],tokensObjectsInXML[tokenPos].getAttribute('id'))
        for wordPos in corr :
          sentence[wordPos].addToken(newToken)
      else :
        #even if token exists, it may not be linked to all words in its correspondance
        for wordPos in corr :
          if sentence[wordPos].getTokenByString(tokensStringsInXML[tokenPos]) is None :
            sentence[wordPos].addToken(newToken)

        newToken.addId(tokensIdsInXML[tokenPos])

      #add this token to the set of tokens
      tokens.add(newToken)

    #3Â°) For all lexemes, identify tokens linked to them. Create Lexeme
    # objects if they do not exist
    #--------------------------------
    lexemesObjectsInXML = wordsSections[2].getElementsByTagName("word")
    for lexemeNode in lexemesObjectsInXML :
      #create a new Lexeme object containing all information
      tempLexeme = Lexeme(lexemeNode)

      #now, test whether this Lexeme was actually already created.

      #first get actual ref token object
      refToken = [x for x in  tokens if tempLexeme.refTokenIds[0] in x.ids]
      if not len(refToken) :
        error('reference Token has not been found!')
      refToken = refToken[0]

      #then try to get lexeme using the form of newLexeme
      currentLexeme = refToken.getLexemeByForm(tempLexeme.forms[0])
      if currentLexeme is not None :
        currentLexeme.mergeWith(tempLexeme)
      else :
        currentLexeme=tempLexeme
        refToken.addLexeme(currentLexeme)
        lexemes.append(currentLexeme)

    #4Â°) Add all dependencies such as :
    # <dependencies type="syntax">
    #   <tagset name="syntacticfunctions" tag="func"/>
    #     <dependency id="dep50">
    #       <!--je crois que j'en suis Â  la vingt deuxiÃšme depuis que je suis lÃ -->
    #       <link depid="lex570" func="subject" govid="lex571" id="func570"/>
    #       <link depid="lex571" func="root" id="func571"/>
    #       <link depid="lex574" func="clg" govid="lex575" id="func574"/>
    #       <link depid="lex582" func="vmod" govid="lex581" id="func582"/>
    #       </dependency>
    #  </dependencies>
    #----------------------
    #get "dependencies" node
  dependencyNodesInXML = doc.getElementsByTagName("dependencies")[0].getElementsByTagName('dependency')
    if len(dependencyNodesInXML) :
      dependencyNodesInXML=dependencyNodesInXML[0].getElementsByTagName('link')
    for dependencyNode in dependencyNodesInXML :
      govId   = dependencyNode.getAttribute('govid')
      depId   = dependencyNode.getAttribute('depid')
      depFunc = dependencyNode.getAttribute('func')
      if len(govId) :
        govLexeme = [x for x in lexemes if govId in x.ids]
        if not len(govLexeme) :
          error('governing lexeme not found!')
        govLexeme=govLexeme[0]
      else :
        govLexeme=None

      depLexeme = [x for x in lexemes if depId in x.ids]
      if not len(depLexeme) :
        error('dependent lexeme not found!')
      depLexeme=depLexeme[0]
      newDependency = Dependency(govLexeme, depLexeme,depFunc)
      if any([newDependency.isequal(x) for x in dependencies]) :
        continue
      dependencies.append(newDependency)

      depLexeme.depends.append(newDependency)
      if govLexeme is not None :
        govLexeme.governs.append(newDependency)
```

```
      # 5Â°)create a token and a lexeme for all words that have no linked token
      #   Set 'cat' as :
      #  'j' (jonction) when word is in an introcoord Node
      #  'i' (interjection) when word is in a discursivemarker Node
      #  'unknown' otherwise
      #  Also create a dependency from root to them.
      #---------------------------------------------------
      nonLinkedWords = [x for x in self.words() if not len(x.tokens)]
      print ' '.join([w.string for w in nonLinkedWords]),'|NONLINKEDWORDS'
      identifiant=10000
      artificalLexemes=[]
      for w in nonLinkedWords :
        fatherNode=self.getContainerOf([w])
        newToken = Token(w.string,'dummyId%d'%identifiant)
        identifiant+=1
        w.addToken(newToken)
        newLexeme = Lexeme(None)
        newLexeme.forms=[w.string]
        newLexeme.lemmas=[w.string]
        if (fatherNode is not None) and (fatherNode.type is 'intro') and (fatherNode.father is not None) and (fath-
erNode.father.type is 'layer') :
          #add feather 'cat' set to 'J' if lexeme is an intro of a layer
          newLexeme.features['cat']=['J']
        elif (fatherNode is not None) and (fatherNode.type is 'discursivemarker') :
          newLexeme.features['cat']=['I']
        else :
          newLexeme.features['cat']=['unknown']
        newToken.addLexeme(newLexeme)
        newDependency = Dependency(None, newLexeme,'root')
        newLexeme.depends.append(newDependency)

      # 6Â°)filter dependencies with respect to pile structure
      #---------------------------------------------------
      pileTree = self.copy()
      pileTree.pileConversion()
      pileNodes = [node for node in pileTree.nodes() if node.type=='pile']

      #gets all intro lexemes
      introNodes = [node for node in self.nodes() if node.type=='intro']
      introLexemes=[]
      for introNode in introNodes :
        introLexemes.extend(introNode.lexemes())

      #then loop on piles
      for pile in pileNodes :
        #for each pile, find all lexemes belonging to its parent
        outLexemes = pile.father.lexemes(recursive=False)
        dependencies = pile.father.dependencies()

        #for each pile, the para dependency is initially empty
        previousInDeps = []

        for layernum,layer in enumerate(pile.sons) :
          inLexemes = layer.lexemes(recursive=True)
          if not len(inLexemes) :
            continue
         toFilterIn = [dep for dep in dependencies if (dep.depLexeme in inLexemes) and (dep.govLexeme in outLexemes)
]

          if layernum :
            #not the first layer, all dependencies with
            #an inLexeme as depLexeme
            #and an outLexeme as govLexeme is invisible
            for dep in toFilterIn :
              dep.func += '_invisible'

          if layernum != len(pile.sons)-1 :
            #not the last layer, all dependencies with
            #an inLexeme as govLexeme
            #and an outLexeme as depLexeme is invisible
              toFilterOut = [dep for dep in dependencies if (dep.govLexeme in inLexemes) and (dep.depLexeme in
outLexemes) ]
            for dep in toFilterOut :
              dep.func += '_invisible'

          #and we add a para dependency if needed
          if len(previousInDeps) :
            previousDepFuncs = uniqu([x.func for x in previousInDeps])
            #para dependencies are put between words in successive layers being the depLexeme of dependencies of the same
type
            for inDep in toFilterIn :
              if re.sub('_invisible',",inDep.func) in previousDepFuncs or inDep.func in previousDepFuncs :
               paraGovLexemes = [x.depLexeme for x in previousInDeps if re.sub(x.func,",inDep.func) in ['','_invisible']]
                  paraGovLexemes = [x for x in paraGovLexemes if x is not None ]
                  paraDepLexeme = inDep.depLexeme
                  if len(paraGovLexemes) :
                    newDep = Dependency(paraGovLexemes[0], paraDepLexeme,'para')
                    paraGovLexemes[0].governs.append(newDep)
                    paraDepLexeme.depends.append(newDep)

                    #add a junc lexeme from the para gov to the intro and from the intro to the para dep
                    intros = [l for l in inLexemes if l in introLexemes]
```

```
                if len(intros) :
                    newDepIntro = Dependency(paraGovLexemes[0], intros[0],'junc')
                    paraGovLexemes[0].governs.append(newDepIntro)
                    intros[0].depends.append(newDepIntro)

                    newDepIntro = Dependency(intros[0],paraDepLexeme,'junc')
                    intros[0].governs.append(newDepIntro)
                    paraDepLexeme.depends.append(newDepIntro)

        previousInDeps=toFilterIn


# 5Â˚) a posteriori linking backward the tokens to the words and the
# lexemes to the tokens. Also creating unique ids for tokens,
# lexemes and dependencies
#----------------------------------------
#link each lexeme with its parent token, create uid for tokens
for k,tok in enumerate(self.tokens()) :
    tok.uid = 'tok%d'%k
    for lex in tok.lexemes : lex.token = tok

#link each token with its parents words
for word in self.words() :
    for tok in word.tokens :
        if word not in tok.words : tok.words.append(word)

#create uid for lexemes. Sorting them in ascending order of appearance of their LAST linked word
sortedLexemes =  sorted(self.lexemes(), key=lambda lex : max([w.pos for w in lex.words()]) )
for k,lex in enumerate(sortedLexemes) :
    lex.features['index']=['%d'%k]
    lex.uid = 'lex%d'%k

#create uid for dependencies
for k,dep in enumerate(self.dependencies()) :
    dep.uid = 'func%d'%k

self.mark('parsed')
```

B.6 Construction de l'arbre

```
def buildTree(text,display=False) :
    """buildTree : parsing algorithm that builds the analysis tree from
    either a texte (string) or a list of words."""
    #--------------------------
    #Initialization :
    #--------------------------

    if not isinstance(text,list) :
        text=re.sub("'","' ",text)
        words = getWords(text)
    else :
        words=text

    #tree init
    root = Node()
    currentNode = root.createSon('iu')
    currentNode.startPos = 0

    #pos init
    currentPos = 0

    associatedTypes = {  '[' :'graft',']' :'graft', '[+' :'graft',
                '(' :'parenthesis', ')' :'parenthesis', '(+' :'parenthesis',
                '{' :'pile', '}' :'pile',
                '<' :'prekernel','<+' :'integratedprekernel',
                '>' :'postkernel','>+' :'integratedpostkernel',
                'parenthesis' :'inkernel','graft' :'embedded'}
    try :
        #main loop on each word of the text
        while currentPos < len(words) :
            #get current word
            currentWord = words[currentPos]
            print currentPos, currentWord.string

            if display : root.log(text,currentNode,'New word: '+currentWord.string+' of type '+currentWord.type)

            #depending on type, perform specific action
            if currentWord.type in ['standard','&'] :
                if display : root.log(text,currentNode,'Normal or & word '+currentWord.string+' is appended to currentNode')
                currentNode.content.append(currentWord)
            elif currentWord.type in ['(','(+'] :
                if display : root.log(text,currentNode,'Creating a parenthesis. First find ancestor')
                while currentNode.type not in ['iu','graft','layer','parenthesis'] :
                    currentNode=currentNode.father
            if display :root.log(text,currentNode,"Found ancestor of type in ['iu','graft','layer','parenthesis']. Now creating
a son of it")
                currentNode = currentNode.createSon(associatedTypes[currentWord.type])
                if currentWord.type == '(+' : currentNode.mark('integrated')

            elif currentWord.type =='[' :
                if display : root.log(text,currentNode,'Creating a son ')
                currentNode = currentNode.createSon(associatedTypes[currentWord.type])

            elif currentWord.type == '{' :
                currentNode = currentNode.createSon('pile')
                if display : root.log(text,currentNode,'Creating a pile son ')
                currentNode = currentNode.createSon('layer')
                if display : root.log(text,currentNode,'Creating a layer son ')
            elif currentWord.type in ['<','<+'] :
                if display : print('Prekernel symbol. Several cases. is currentNode of type *kernel*?'),raw_input()
                if 'kernel' in currentNode.type :
                    currentNode.type=associatedTypes[currentWord.type]
                    currentNode=currentNode.father
                    if display : root.log(text,currentNode,'Yes, type has been changed and currentNode is its father')
                else :
                        if display : print('No. The we must move everything not of type intro nor *kernel* to a new son of type
prekernel')  ,raw_input()

                movedSons = currentNode.getSonsNotOfTypes(['intro','kernel','prekernel','postkernel','integratedprekernel','integratedpostkernel'])
                    for son in movedSons :
                        son.detach()
                    movedContent = currentNode.content[ :]
                    currentNode.content = []
                    if display : root.log(text,currentNode,'Content has been temporarily detached')
                    newKernelSon = currentNode.createSon(associatedTypes[currentWord.type])
                    if display : root.log(text,currentNode,'Son has been created. Now attaching content to it.')
                    for son in movedSons :
                        son.attach(newKernelSon)
                    newKernelSon.content = movedContent
                    if display : root.log(text,currentNode,'Done.')
            elif (currentWord.type in ['>','>+']) :
                if display : print 'Postkernel symbol. Going to container ancestor and creating a new son',raw_input()
                while currentNode.type not in ['iu','graft','layer','parenthesis'] :
                    currentNode=currentNode.father
                      if display : root.log(text,currentNode,"Found ancestor of type in ['iu','graft','layer','parenthesis']. Now
creating a son of it")
                currentNode = currentNode.createSon(associatedTypes[currentWord.type])
                if display : root.log(text,currentNode,'Done.')
            elif (currentWord.type == '"') :
```

```
          if display :print("Discursive marker symbol, finding the next one, and insert all words in between to a new son
of type discursivemarker"),raw_input()
          currentPos += 1
          beginPos = currentPos
          while (words[currentPos].type != '"') :
            currentPos+=1
          newSon = currentNode.createSon('discursivemarker')
          newSon.content.extend(words[beginPos :currentPos])
          if display : root.log(text,currentNode,'Done.')
        elif currentWord.type == '^' :
          if display :print("Intro symbol, insert the next word to a new son of type intro"),raw_input()
          introNode = currentNode.createSon('intro')
          currentPos += 1
          nextWord = words[currentPos]
          if nextWord.type != 'standard' :
            #should be standard
            print("Word after intro should be standard"),raw_input()
            raise error
          introNode.content.append(nextWord)
          if display : root.log(text,currentNode,'Done.')
        elif currentWord.type == '//' :
          if display :print("End of ui symbol. Getting container ancestor."),raw_input()
          while currentNode.type not in ['iu','graft','layer','parenthesis'] :
            currentNode=currentNode.father
          if display : root.log(text,currentNode,"Found ancestor of type in ['iu','graft','layer','parenthesis']. Is it of
type ui?")
          if currentNode.type == 'iu' :
            currentNode = currentNode.father.createSon('iu')
            if display : root.log(text,currentNode,"Yes. Created a brother of type ui.")
          else :
            if display :print("No. We need to insert a new node of type ui"),raw_input()
            oldFather = currentNode.father
            currentNode.detach()
            appendedNode = oldFather.createSon(currentNode.type)
            currentNode.attach(appendedNode)
            currentNode.type = 'iu'
            if display : root.log(text,currentNode,"And now create a brother of type ui.")
            currentNode = appendedNode.createSon('iu')
            if display : root.log(text,currentNode,'Done.')
          currentNode.startPos = currentPos+1
        elif  (currentWord.type in [')', ']','}']) :
          if display : print('Closing parenthesis or brackets. First find containing ancestor'),raw_input()
          oldCurrentNode = currentNode
          while currentNode.type != associatedTypes[currentWord.type] :
            currentNode = currentNode.father
          if display : root.log(text,currentNode,'Containing ancestor found.')
          if (currentWord.type == ')') and currentNode.ismarked('integrated') :
         if display :print("We are closing a parenthesis that was marked as integrated. Change its type to integratedinkernel"),raw_input()

            currentNode.type = 'integratedinkernel'
            currentNode.unmark('integrated')
          elif (   (currentWord.type in [']',')'])
             and (   len(currentNode.content)
                 or len(currentNode.getSonsNotOfTypes(['iu']))  )  ) :
            #if does not contain ui, replace () and [] with inkernel and graft
            #respectively
            currentNode.type = associatedTypes[currentNode.type]
             if display : root.log(text,currentNode,'CurrentNode is of type [] or () and does not contain ui sons. Change
its type accordingly')
          if (  (currentWord.type in [']'] )
            and ('kernel' not in currentNode.father.type)) :
              if display : print("We are closing a [] whose father is not a kernel."),raw_input()
              A = currentNode.father
              currentNode.detach()
              kernels = A.getSonsOfTypes(['kernel'])
              if not len(kernels) :
                if display : print("No kernel son, creating a new kernel as the father."),raw_input()
                father=A.createSon('kernel')
                father.content = [x for x in A.content]
                A.content = []
              else :
                if display : print("There was a kernel unclue. Moving currentNode as a son of it ."),raw_input()
                father = kernels[0]
              currentNode.attach(father)
              if display : root.log(text,currentNode,'Done.')
          currentNode = currentNode.father
          if display : root.log(text,currentNode,'Finally moved to father. Done')

        elif currentWord.type == '|' :
          if display : print('New Layer. First, find containing pile'),raw_input()
          while currentNode.type != 'pile' :
            currentNode = currentNode.father
          if display : root.log(text,currentNode,'Done, now create a new layer son of it.')
          currentNode = currentNode.createSon('layer')
          if display : root.log(text,currentNode,'Done')
        elif currentWord.type == '|}' :
          if display : print('interrupted pile. Find containing pile and mark it as interrupted'),raw_input()
          while currentNode.type != 'pile' :
            currentNode = currentNode.father
          currentNode.mark('interrupted')
          currentNode = currentNode.father
          if display : root.log(text,currentNode,'Done. Now got father as currentNode')
```

```
        elif currentWord.type == '{|' :
         if display : print('Entering an interrupted pile. There should be a son of currentNode of type pile that is marked
as interrupted'),raw_input()
           while currentNode.type != 'root' :
             currentNode = currentNode.father
           #pileSons = [x for x in currentNode.nodes() if (x.type=='pile' and x.ismarked('interrupted'))  ]
           pileSons = [x for x in currentNode.nodes() if (x.type=='pile' and x.ismarked('interrupted'))  ]
           #pileSons = [x for x in currentNode.getSonsOfTypes(['pile']) if x.ismarked('interrupted')]
           if not len(pileSons) :
             #there should be at least a pile
             print("Probleme de reouverture de pile a une mauvaise position."),raw_input()
             raise error
           if display : print('Found it. unmark it and go into it in a new layer son of it'),raw_input()
           #pileSons[-1].unmark('interrupted')
           currentNode = pileSons[-1].createSon('layer')
           if display : root.log(text,currentNode,'Done.')
         currentPos += 1
    except :
      print "-----------------------------"
      print "PARSING ERROR while parsing symbol " + words[currentPos].string + " at position " + str(currentPos)
      print ""
      print "  The context was:"
     context = ' '.join([words[pos].string for pos in range(max(0, currentPos - 10), min(len(words)-1,currentPos+10))])
      print context
      print "-----------------------------"
      sys.exit(0)

    if display :
      root.log(text,currentNode,'Cleaning tree.')
    root.clean()

    if display :
        root.log(text,currentNode,'Now creating kernels for nodes that are not of type *kernel*, that do have content and
that do not have kernel sons.')

    for node in root.nodes()[ :] :
      if ( (node.type not in ['intro','prekernel', 'kernel','postkernel','integratedprekernel',
                  'integratedpostkernel','inkernel','integratedinkernel','discursivemarker'])
        and (node.content) and (not node.getSonsOfTypes('kernel'))) :
        #gather content of nodes that are not *kernel to a kernel son if it doesn't exist
        kernel = node.createSon('kernel')
        kernel.content = [x for x in node.content]
        node.content = []

    introNodes = [n for n in root.nodes() if n.type is 'intro']
    for intro in introNodes :
      if intro.father is None : continue
      introBrothers = [n for n in intro.father.getSonsOfTypes(['intro']) if n is not intro]
      if not len(introBrothers) :continue
      for introBrother in introBrothers :
        intro.content.extend(introBrother.content)
        introBrother.detach()

    root.mark('annotation',text)
    if display :
      root.log(text,currentNode,'Finished.')
    return root,words
```

B.7 Récolte des données statistiques

```
def stats_memoire() :
  import yaml


  def encodeSymbol(symb,dico) :
    #cherche la marque de l'interval dans le dictionnaire
    if symb.mark() not in dico :
      #s'il n'y est pas, on l'ajoute
      dico.append(symb.mark())
    #on renvoit la position du symbole dans le dictionnaire
   return dico.index(symb.mark()) # ATTENTION, je suppose que le dico reste augmentÃ© Ã  chaque appel, ca semble
vrai apres tests

  def encodeFloat(interval,defaultNum=-1) :
    try :
      result = float(interval.mark())
    except :
      result = defaultNum
    return result

  def removeDuplicates(intervals) :
    result = []
    cles = []
    for interval in intervals :
      cle = interval.xmin()*1E8+interval.xmax()
      if cle not in cles :
        result.append(interval)
        cles.append(cle)
    return result

  annotsDir = './data/input_programme'
  TextGridDir = './data/TextGridAvecProsodieSyntaxe'

  files = os.listdir(annotsDir)

  syllInPiles = []
  allSyll = []

  #allocation des dictionnaires
  #----------
  localRegDict = []
  shapeDict = []
  promDict = []
  hesDict = []
  sylDict = []
  groupTypeDict = []
  footTypeDict = []
  packageTypeDict = []

  for ifile,fileName in enumerate(files) :
    #preparation de l'arbre et matching au textgrid
    #----------
    baseFilename=os.path.splitext(os.path.basename(fileName))[0]
    print 'Handling file ', baseFilename
    tree,annotWords=rhapsTest(annotsDir+'/'+fileName,parse=False,
                    TextGridName=TextGridDir+'/'+baseFilename+'.TextGrid',
                    XMLName=None)

    tree.pileConversion()
    tree.prettyPrint()


    # RÃ©cupÃ©ration des tiers
    #----------

    #recuperation de l'objet textgrid
    if not tree.ismarked('matchedWithTextGrid') :
      error('Tree not attached to textgrid, aborting.')
    tg = tree.marks['TextGrid']

    #chercher la tier syllabe
    sylTier = [t for t in tg if t.name()=='syllabe']
    if not len(sylTier) :
      error('Syllabe tier not found; aborting.')
    sylTier = sylTier[0]

    #chercher la tier ralentissement
    ralTier = [t for t in tg if t.name()=='ralentissement_diphones']
    if not len(ralTier) :
      error('Ralentissement tier not found; aborting.')
    ralTier = ralTier[0]

    #chercher la tier mean_f0
    f0Tier = [t for t in tg if t.name()=='mean_f0']
    if not len(f0Tier) :
      error('mean_f0 tier not found; aborting.')
    f0Tier = f0Tier[0]

    #chercher la tier syllableLocalReg
```

```
localRegTier = [t for t in tg if t.name()=='syllableLocalReg']
if not len(localRegTier) :
  error('syllableLocalReg tier not found; aborting.')
localRegTier = localRegTier[0]

#chercher la tier syllableShape
shapeTier = [t for t in tg if t.name()=='syllableShape']
if not len(shapeTier) :
  error('syllableShape tier not found; aborting.')
shapeTier = shapeTier[0]

#chercher la tier prom
promTier = [t for t in tg if t.name()=='prom']
if not len(promTier) :
  error('prom tier not found; aborting.')
promTier = promTier[0]

#chercher la tier hes
hesTier = [t for t in tg if t.name()=='hes']
if not len(hesTier) :
  error('hes tier not found; aborting.')
hesTier = hesTier[0]

#chercher la tier groupType
groupTypeTier = [t for t in tg if t.name()=='groupType']
if not len(groupTypeTier) :
  error('groupType tier not found; aborting.')
groupTypeTier = groupTypeTier[0]

#chercher la tier footType
footTypeTier = [t for t in tg if t.name()=='footType']
if not len(footTypeTier) :
  error('footType tier not found; aborting.')
footTypeTier = footTypeTier[0]

#chercher la tier packageType
packageTypeTier = [t for t in tg if t.name()=='packageType']
if not len(packageTypeTier) :
  error('packageType tier not found; aborting.')
packageTypeTier = packageTypeTier[0]


# Extraction de toutes les syllabes et construction de la table
#---------------------------------------
#recherche des intervalles par syllabe inclues dans cette iu
syllabes = removeDuplicates(tree.intervalsInTier(sylTier));
ralentissements = removeDuplicates(tree.intervalsInTier(ralTier))
f0s = removeDuplicates(tree.intervalsInTier(f0Tier))
localRegs = removeDuplicates(tree.intervalsInTier(localRegTier))
shapes = removeDuplicates(tree.intervalsInTier(shapeTier))
proms = removeDuplicates(tree.intervalsInTier(promTier))
hess = removeDuplicates(tree.intervalsInTier(hesTier))

for isyl, (syl,ral,f0,localReg,shape,prom,hes) in enumerate(zip(syllabes,ralentissements,f0s,localRegs,shapes,proms,hess)) :
    #creation de l'entree pour cette syllabe
    entry = [ifile,                    #1 numero du fichier
        isyl,                  #2 position de la syllabe en cours dans le fichier
        encodeFloat(f0),           #3 mean f0
        encodeFloat(ral),          #4 ralentissement
        encodeSymbol(syl,sylDict),     #5 syllabe
        encodeSymbol(prom,promDict),   #6 prom
        encodeSymbol(hes,hesDict),     #7 hes
        encodeSymbol(shape,shapeDict),     #8 shape
        encodeSymbol(localReg,localRegDict),#9 localReg
        syl.xmin(),                #10 xmin
        syl.xmax(),                #11 xmax
        ]
    allSyll.append(entry)


# Extraction de toutes les syllabes dans des piles et construction de la table
#---------------------------------------

#prend tous les noeuds de l'arbre
nodes = tree.nodes()

#extrait les piles
piles = [n for n in nodes if n.type=='pile']

for pospile,pile in enumerate(piles) :
  pileAncestors = [n for n in pile.ancestors() if n.type is 'pile']
  pileSons = [n for n in pile.nodes(includeSelf=False) if n.type is 'pile']

  #normalement tous les enfants d'une pile sont des layers,
  #mais je les cherche comme ca pour etre bien sur que
  #c'en sont
  layers = [n for n in pile.sons if n.type=='layer']

  for poslayer,layer in enumerate(layers) :

    #recherche des intervalles par syllabe inclues dans cette iu
    syllabes = removeDuplicates(layer.intervalsInTier(sylTier));
```

```
    ralentissements = removeDuplicates(layer.intervalsInTier(ralTier))
    f0s = removeDuplicates(layer.intervalsInTier(f0Tier))
    localRegs = removeDuplicates(layer.intervalsInTier(localRegTier))
    shapes = removeDuplicates(layer.intervalsInTier(shapeTier))
    proms = removeDuplicates(layer.intervalsInTier(promTier))
    hess = removeDuplicates(layer.intervalsInTier(hesTier))

    nsyls = len(syllabes)

for isyl, (syl,ral,f0,localReg,shape,prom,hes) in enumerate(zip(syllabes,ralentissements,f0s,localRegs,shapes,proms,hess)) :
    #creation de l'entree pour cette syllabe
    entry = [ifile,                    #1 numero du fichier
        pospile,                  #2 numero de pile
        poslayer,                 #3 layer en cours
        isyl,                     #4 position de la syllabe en cours dans le layer
        encodeFloat(f0),          #5 mean f0
        encodeFloat(ral),         #6 ralentissement
        len(pileAncestors),       #7 nb de parents pile
        len(pileSons),            #8 nb d'enfants pile
        int(pile.ismarked('interrupted')),  #9 pile interrompue ************************
        len(layers),              #10 nombre de layers de la pile
        nsyls,                    #11 nombre de syllabes dans layer
        encodeSymbol(syl,sylDict),      #12 syllabe
        encodeSymbol(prom,promDict),    #13 prom
        encodeSymbol(hes,hesDict),      #14 hes
        encodeSymbol(shape,shapeDict),  #15 shape
        encodeSymbol(localReg,localRegDict),#16 localReg
        syl.xmin(),               #17 xmin
        syl.xmax(),               #18 xmax
        ]
    syllInPiles.append(entry)

secureWrite(yaml.dump(allSyll),'allSyll.yaml')
secureWrite(yaml.dump(syllInPiles),'syllInPiles.yaml')
secureWrite(yaml.dump(localRegDict),'localRegDict.yaml')
secureWrite(yaml.dump(shapeDict),'shapeDict.yaml')
secureWrite(yaml.dump(promDict),'promDict.yaml')
secureWrite(yaml.dump(hesDict),'hesDict.yaml')
secureWrite(yaml.dump(sylDict),'sylDict.yaml')
secureWrite(yaml.dump(groupTypeDict),'groupTypeDict.yaml')
secureWrite(yaml.dump(footTypeDict),'footTypeDict.yaml')
secureWrite(yaml.dump(packageTypeDict),'packageTypeDict.yaml')
tree.prettyPrint()
```

B.8 Script principal et fonctions annexes

```
import re
import sys
import codecs
import os
from xml.dom import minidom
from xml.dom.minidom import Document
import TextGrid as tg
from nltk.featstruct import unify


def secureRead(fileName) :
    """secureRead : reads content of the file, but through detection of the encoding using chardet.
    replace all this stuff with a simple read if needed"""
    import chardet
    rawHeader=open(fileName,"r")
    encoding_pronostic =  chardet.detect(rawHeader.read())
    rawHeader.close()
   print "assuming encoding " + encoding_pronostic['encoding'] + ' with confidence ' + str(encoding_pronostic['confidence']*100)+'%'
    fHandle = codecs.open(fileName, 'r',encoding_pronostic['encoding'])
    text=fHandle.read()
    fHandle.close()
    return text

def secureWrite(text,fileName,encoding='utf-8') :
    """secureWrite : writes to file using specified encoding. See following webpage for a list :
     http ://docs.python.org/library/codecs.html#standard-encodings (first column) """
    fHandle = codecs.open(fileName, 'w',encoding)

    if isinstance(text,list) :
        #if the given text is a list of data, writes each data followed by a new line
        for sentence in text :
            fHandle.write(sentence+'\n') ;
    else :
        fHandle.write(text)
    fHandle.close()

def uniqu(A) :
    """uniqu : get unique elements in list A while preserving order"""
    result = []
    for x in A :
        if x not in result :result.append(x)
    return result


def mainTests() :
    """parsing tests"""

   entrees = [u"""euh" { deux petites phrases | deux vraies options } qui dessinent { votre route //+ | une route qui tÃ©moigne
{ d'une certaine | d'une bonne | d'une trÃ¨s bonne } conduite } //"""]

    for entree in entrees[ :] :
        root,words = buildTree(entree,True)
        liste=root.unroll()
        print '----------------------'
        print entree
        print '----------------------'
        print 'Complete tree: pretty print:'
        root.prettyPrint()

        topologicalTree = root.copy()
        topologicalTree.topologicalConversion()
        print ' '
        print 'Topological tree: pretty print'
        topologicalTree.prettyPrint()

        pileTree = root.copy()
        pileTree.pileConversion()
        print ' '
        print 'Pile tree: pretty print'
        pileTree.prettyPrint()

        print '----------------------'
        print 'Unrolled sentences:'
        textList = listToText(liste)
        for x in textList :
            print x
        print ' '
        print ' '
    sys.exit(0)



def rhapsTest(filename,parse=True, TextGridName=None,XMLName=None) :
    """rhapsTest : main rhapsodie function. Builds tree from annotation
    in filename, matches with filename.TextGrid, parses sentences using
    Eric and builds final XML.
    """
    text=secureRead(filename)
    print text
```

```
    tree,annotWords=buildTree(text)
    if TextGridName is not None :
        tree.matchTextGrid(TextGridName,'pivot')
    if parse :
        import rhapsodieEricParsing as eric
        tree.parse()
    if XMLName is not None :
        print 'Tree : XML:'
        xmlDoc = tree.buildRhapsodieXML(wordsList=annotWords)
        fHandle = codecs.open(XMLName,'w','utf-8')
        fHandle.write(xmlDoc.toprettyxml(indent="      "))
        fHandle.close()

    return tree,annotWords


if __name__ == "__main__" :

    #code a executer pour rÃ©colter les donnÃ©es
    #stats_memoire2()
    #sys.exit(0)

    #code a executer pour faire des tests de construction d'arbres
    mainTests()
    sys.exit(0)

    #code Ã  executer pour le traitement complet du corpus Rhapsodie
    annotsDir = './data/input_programme'
    TextGridDir = './data/TextGrid'
    resultsDir = './fichiers_ok'

    files = os.listdir(annotsDir)

    tokensCount=[]
    iusCount=[]
    pilesCount = []
    embeddedgraftsCount = []
    PreKernelCount = []
    PostKernelCount = []
    tokensPerIu = []
    pilesPerIu = []
    PreKernelPerIu = []
    PostKernelPerIu = []
    tokensPerIuGlobal = 0
    pilesPerIuGlobal = 0
    PreKernelPerIuGlobal = 0
    PostKernelPerIuGlobal = 0

    totalEmbedStrings= ['parts of annotation including grafts or embedded:',"]
    introStrings = []
    PreKernelStrings= []
    PostKernelStrings= []


    for ifile,fileName in enumerate(files) :
        baseFilename=os.path.splitext(os.path.basename(fileName))[0]
        print 'Handling file ', baseFilename
        tree,annotWords=rhapsTest(annotsDir+'/'+fileName,parse=True,
                        TextGridName=TextGridDir+'/'+baseFilename+'.TextGrid',
                        XMLName=resultsDir+'/'+baseFilename+'.xml')


        #append all intro strings to total intro strings
        intronodes = [n for n in tree.nodes() if n.type is 'intro']
        for intro in intronodes :
            introString = ' '.join([w.string for w in sorted(intro.content,key=lambda word :word.pos)])
            if introString not in introStrings :
                introStrings.append(introString)
        introStrings.sort()
        secureWrite(introStrings, 'intro_total.txt')

        print 'statistics : number of tokens...'

        #number of tokens
        tokensCount.append(len(tree.tokens()))
        allNodes = tree.nodes()

        print 'statistics : intro and discursive markers...'

        #output files containing all content of nodes of a certain type
        typesToList=['intro','discursivemarker','pile','embedded','graft','prekernel','postkernel']
        for type in typesToList :
            nodes = [n for n in allNodes if n.type == type]
            wordsStrings=[]
            for node in nodes :
                wordsStrings.append(' '.join([w.string for w in sorted(node.content,key=lambda word :word.pos)]))
            secureWrite(sorted(wordsStrings), 'fichiers_ok/output_'+fileName+type+'.txt')

        print 'statistics : getting all piles...'

        #getting all piles
        pilesNodes = [n for n in allNodes if n.type == 'pile']
```

```
pileIntervals = []
for pile in pilesNodes :
   pileIntervals.extend(pile.intervals())
pileIntervals = uniqu(pileIntervals)

#outputs text grid with additional tier for piles
grid = tree.filterTextGrid(pileIntervals)

newGrid = tg.TextGrid()
newGrid.read(TextGridDir+'/'+baseFilename+'.TextGrid')
newGrid.append(grid[0])
newGrid.write('fichiers_ok/'+baseFilename+'_with_piles_tier.TextGrid')


#piles count for this file
pilesCount.append(len(pilesNodes))

#getting all embedded/grafts
print 'statistics : number of embedded/grafts...'
embeddedGraftNodes = [n for n in allNodes if n.type in ['embedded','graft']]
#[] count for this file
embeddedgraftsCount.append(len(embeddedGraftNodes))

#getting all prekernels
print 'statistics : number of prekernels...'
PreKernelNodes = [n for n in allNodes if n.type in ['prekernel']]
#[] count for this file
PreKernelCount.append(len(PreKernelNodes))

#getting all postkernels
print 'statistics : number of postkernels...'
PostKernelNodes = [n for n in allNodes if n.type in ['postkernel']]
#[] count for this file
PostKernelCount.append(len(PostKernelNodes))

print 'statistics : getting all ius...'
#getting all ius
iuNodes = [n for n in allNodes if n.type == 'iu']
#iu count for this file
iusCount.append(len(iuNodes))

#initializes all embed&graft sentences to empty
embedStrings= []
embedContent=False

#initializes all prekernel sentences to empty
PreKernelStrings= []
PreKernelContent=False

#initializes all postkernel sentences to empty
PostKernelStrings= []
PostKernelContent=False

#initializes tokens and piles per iu data
tokensPerIu.append({'mean' :0,'detail' :[]})
pilesPerIu.append({'mean' :0,'detail' :[]})
PreKernelPerIu.append({'mean' :0,'detail' :[]})
PostKernelPerIu.append({'mean' :0,'detail' :[]})


sumToken=0
sumPiles = 0
sumPreKernel = 0
sumPostKernel = 0
#per iu :
for iiu,iu in enumerate(iuNodes) :
   print '      iu %d/%d : getting all nodes'%(iiu,len(iuNodes))
   nodesInIu= iu.nodes()

   #if iu contains nodes of type graft or embedded, add it to output
   if len([n for n in nodesInIu if n.type in ['graft','embedded']]) :
      print '      iu %d/%d : ui contains [], adding it to embedStrings'%(iiu,len(iuNodes))
      posmin,posmax = iu.getSpan()
      posmax = min(posmax,len(annotWords)-1)
      while (posmax < len(annotWords)) and (annotWords[posmax].type != '//') :
         posmax+=1
      embedStrings.append(' '.join([w.string for w in annotWords[(posmin+1) :(posmax+1)]]))
      embedStrings.append(")
      embedContent=True

   #if iu contains nodes of type prekernel, add it to output
   if len([n for n in nodesInIu if n.type in ['prekernel']]) :
      print '      iu %d/%d : ui contains <, adding it to PreKernelStrings'%(iiu,len(iuNodes))
      posmin,posmax = iu.getSpan()
      posmax = min(posmax,len(annotWords)-1)
      while (posmax < len(annotWords)) and (annotWords[posmax].type != '//') :
         posmax+=1
      PreKernelStrings.append(' '.join([w.string for w in annotWords[(posmin+1) :(posmax+1)]]))
      PreKernelStrings.append(")
      PreKernelContent=True

   #if iu contains nodes of type prekernel, add it to output
```

```
    if len([n for n in nodesInIu if n.type in ['postkernel']]) :
      print '     iu %d/%d : ui contains >, adding it to PostKernelStrings'%(iiu,len(iuNodes))
      posmin,posmax = iu.getSpan()
      posmax = min(posmax,len(annotWords)-1)
      while (posmax < len(annotWords)) and (annotWords[posmax].type != '//') :
        posmax+=1
      PostKernelStrings.append(' '.join([w.string for w in annotWords[(posmin+1) :(posmax+1)]]))
      PostKernelStrings.append('');
      PostKernelContent=True

    #count tokens in iu
    print '     iu %d/%d : getting all tokens'%(iiu,len(iuNodes))
    tokensPerIu[-1]['detail'].append(len(iu.tokens()))
    sumToken+=tokensPerIu[-1]['detail'][-1]

    #count piles in iu
    print '     iu %d/%d : getting all piles'%(iiu,len(iuNodes))
    piles = [n for n in nodesInIu if n.type=='pile']
    pilesPerIu[-1]['detail'].append(len(piles))
    sumPiles+=len(piles)

    #count prekernels in iu
    print '     iu %d/%d : getting all prekernel'%(iiu,len(iuNodes))
    prekernel = [n for n in nodesInIu if n.type=='prekernel']
    PreKernelPerIu[-1]['detail'].append(len(prekernel))
    sumPreKernel+=len(prekernel)

    #count postkernels in iu
    print '     iu %d/%d : getting all postkernel'%(iiu,len(iuNodes))
    postkernel = [n for n in nodesInIu if n.type=='postkernel']
    PostKernelPerIu[-1]['detail'].append(len(postkernel))
    sumPostKernel+=len(postkernel)

#compute mean number of piles and tokens per iu
tokensPerIu[-1]['mean'] = sumToken/float(len(iuNodes))
pilesPerIu[-1]['mean'] = sumPiles/float(len(iuNodes))
PreKernelPerIu[-1]['mean'] = sumPreKernel/float(len(iuNodes))
PostKernelPerIu[-1]['mean'] = sumPostKernel/float(len(iuNodes))
#outputs graft&embed data
print 'statistics : writing embedStrings'
if embedContent :
    tempString = ['parts of annotation including grafts or embedded:',"]
    tempString.extend(embedStrings)
    secureWrite(tempString, 'fichiers_ok/output_'+fileName+'_graft&embedd'+'.txt')

totalEmbedStrings.extend(embedStrings)
secureWrite(totalEmbedStrings, 'graft&embedd_total.txt')


tokensPerIuGlobal = sum([ sum(x['detail']) for x in tokensPerIu]) / float(sum(iusCount))
pilesPerIuGlobal =  sum([ sum(x['detail']) for x in  pilesPerIu]) / float(sum(iusCount))
PreKernelPerIuGlobal = sum([ sum(x['detail']) for x in PreKernelPerIu]) / float(sum(iusCount))
PostKernelPerIuGlobal =  sum([ sum(x['detail']) for x in  PostKernelPerIu]) / float(sum(iusCount))

txt = []
txt+=['Summary of analysis']
txt+=['-------------' ]
txt+=['Number of tokens (global) : %d'%sum(tokensCount)]
txt+=['Number of ius    (global) : %d'%sum(iusCount)]
txt+=['Number of piles  (global) : %d'%sum(pilesCount)]
txt+=['Number of []     (global) : %d'%sum(embeddedgraftsCount)]
txt+=['Number of prekernels     (global) : %d'%sum(PreKernelCount)]
txt+=['Number of postkernels    (global) : %d'%sum(PostKernelCount)]
txt+=['Mean nb of tokens per iu  : %f'%tokensPerIuGlobal]
txt+=['Mean nb of piles per iu   : %f'%pilesPerIuGlobal]
txt+=['Mean nb of prekernels per iu  : %f'%PreKernelPerIuGlobal]
txt+=['Mean nb of postkernels per iu   : %f'%PostKernelPerIuGlobal]
txt+=["]
txt+=['Summary per file']
txt+=['-------------']
for k,ffilename in enumerate(files[0 :ifile+1]) :
    baseFilename=os.path.basename(ffilename)
    txt+=['  file '+baseFilename+':']
    txt+=['     Number of tokens     : %d'%tokensCount[k]]
    txt+=['     Number of ius        : %d'%iusCount[k]]
    txt+=['     Number of piles      : %d'%pilesCount[k]]
    txt+=['     Number of embedded+graft: %d'%embeddedgraftsCount[k]]
    txt+=['     Number of prekernels: %d'%PreKernelCount[k]]
    txt+=['     Number of postkernels: %d'%PostKernelCount[k]]
    txt+=['     Mean nb of tokens per iu: %f'%tokensPerIu[k]['mean']]
    txt+=['     Mean nb of piles per iu : %f'%pilesPerIu[k]['mean']]
    txt+=['     Mean nb of prekernels per iu: %f'%PreKernelPerIu[k]['mean']]
    txt+=['     Mean nb of postkernels per iu : %f'%PostKernelPerIu[k]['mean']]
    """txt+=['       Detail per iu :']
    for iiu in range(len(tokensPerIu[k]['detail'])) :
        txt+=['          iu %d : %d tokens, %d piles'%(iiu,tokensPerIu[k]['detail'][iiu],pilesPerIu[k]['detail'][iiu])]
    """
    txt+=["]

secureWrite(txt,'Summary.txt')
```

B.9 Construction des données Matlab depuis Python

```
import yaml
from arbres_printemps import *
from numpy import *
import pylab as pl
from scipy import *

#charger les donnees pour toutes les syllabes et les enregistrer dans un fichier texte
allSylls = array(yaml.load(secureRead('allSyll.yaml')))
(N,D) = allSylls.shape
print 'traitement de la matrice de resultats, de taille %d x %d'%(N,D)
result = []
for n in range(N) :
   result.append('\t'.join([str(x) for x in allSylls[n, :]]))
print 'ecriture de allSylls.txt'
secureWrite(result,'allSylls.txt')


#charger les donnees et les enregistrer dans un fichier texte
sylls = array(yaml.load(secureRead('syllInPiles.yaml')))
(N,D) = sylls.shape
print 'traitement de la matrice de resultats, de taille %d x %d'%(N,D)
result = []
for n in range(N) :
   result.append('\t'.join([str(x) for x in sylls[n, :]]))
print 'ecriture de syllsInPiles.txt'
secureWrite(result,'syllsInPiles.txt')



#charger les dictionnaires

localRegDict = yaml.load(secureRead('localRegDict.yaml'))
shapeDict = yaml.load(secureRead('shapeDict.yaml'))
promDict = yaml.load(secureRead('promDict.yaml'))
hesDict = yaml.load(secureRead('hesDict.yaml'))
sylDict = yaml.load(secureRead('sylDict.yaml'))
groupTypeDict = yaml.load(secureRead('groupTypeDict.yaml'))
footTypeDict = yaml.load(secureRead('footTypeDict.yaml'))
packageTypeDict = yaml.load(secureRead('packageTypeDict.yaml'))

#creation du script matlab de chargement des donnees
strings = []
strings.append('dicos={ {\"+ '\',\".join(sylDict)+'\'},')
strings.append('{\"+ '\',\".join(promDict)+'\'},')
strings.append('{\"+ '\',\".join(hesDict)+'\'},')
strings.append('{\"+ '\',\".join(shapeDict)+'\'},')
strings.append('{\"+ '\',\".join(localRegDict)+'\'},')
strings.append('{\"+ '\',\".join(groupTypeDict)+'\'},')
strings.append('{\"+ '\',\".join(footTypeDict)+'\'},')
strings.append('{\"+ '\',\".join(packageTypeDict)+'\'}};')
strings.append('load allSylls.txt');
strings.append('load syllsInPiles.txt');
strings.append('fieldNameAll = {\'ifile\',\'isyl\',\'f0\',\'rallentissement\',\'syllabe\',\'proeminence\',\'HES\',\'forme\',\'registre
local\',\'xmin\',\'xmax\'};');
strings.append('fieldName = {\'ifile\',\'ipile\',\'ilayer\',\'isyl\',\'f0\',\'rallentissement\',\'pileAncestors\',\'pileSons\',\'interrompue\',\'nb
local\',\'xmin\',\'xmax\'};');
strings.append('dicoCorrespondant   = [ 0,      0,        0,      0,    0,    0,      0,              0,      0,0,          0,
1,          2,          3,      4,          5,              0,      0   ];');
strings.append('dicoCorrespondantAll   = [ 0,      0,        0, 0,     1,    2,    3,      4,              5,      0,0,          0,
1,          2,          3,      4,          5,              0,      0   ];');

secureWrite(strings,'loaddata.m')

sys.exit(0)
```

B.10 Script de manipulation des TextGrids

```
# classes for Praat TextGrid data structures, and HTK .mlf files
# Kyle Gorman <kgorman@ling.upenn.edu>
# Modifications : Antoine Liutkus <antoine@liutkus.net> and Julie Beliao <
    julie@beliao.fr>

# TODO: documentation

import codecs
import chardet

class mlf:
    """ read in a HTK .mlf file. iterating over it gives you a list of
    TextGrids """

    def __init__(self, file):
        self.__items = []
        self.__n = 0
        text = open(file, 'r')
        text.readline() # get rid of header
        while 1: # loop over text
            name = text.readline()[1:-1]
            if name:
                grid = TextGrid()
                phon = IntervalTier('phones')
                word = IntervalTier('words')
                wmrk = ''
                wsrt = 0.
                wend = 0.
                while 1: # loop over the lines in each grid
                    line = text.readline().rstrip().split()
                    if len(line) == 4: # word on this baby
                        pmin = float(line[0]) / 10e6
                        pmax = float(line[1]) / 10e6
                        phon.append(Interval(pmin, pmax, line[2]))
                        if wmrk:
                            word.append(Interval(wsrt, wend, wmrk))
                        wmrk = line[3]
                        wsrt = pmin
                        wend = pmax
                    elif len(line) == 3: # just phone
                        pmin = float(line[0]) / 10e6
                        pmax = float(line[1]) / 10e6
                        phon.append(Interval(pmin, pmax, line[2]))
                        wend = pmax
                    else: # it's a period
                        word.append(Interval(wsrt, wend, wmrk))
                        self.__items.append(grid)
                        break
                grid.append(phon)
                grid.append(word)
                self.__n += 1
            else:
                text.close()
                break

    def __iter__(self):
        return iter(self.__items)

    def __len__(self):
        return self.__n

    def __str__(self):
        return '<MLF instance with %d TextGrids>' % self.__n

class TextGrid:
    """ represents Praat TextGrids as list of different types of tiers """

    def __init__(self, name = None):
        self.__tiers = []
        self.__n = 0
        self.__xmin = None
        self.__xmax = None
        self.__name = name # this is just for the MLF case
        self.__encoding = 'utf-8' #default


    def __str__(self):
        return '<TextGrid with %d tiers>' % self.__n
```

```
def __iter__(self):
    return iter(self.__tiers)

def __len__(self):
    return self.__n

def __getitem__(self, i):
    """ return the (i-1)th tier """
    return self.__tiers[i]

def xmin(self):
    return self.__xmin

def xmax(self):
    return self.__xmax

def append(self, tier):
    self.__tiers.append(tier)
    if self.__xmax is not None:
        self.__xmax = max(self.__xmax, tier.xmax())
    else:
        self.__xmax = tier.xmax()
    if self.__xmin is not None:
        self.__xmin = min(self.__xmin, tier.xmin())
    else:
        self.__xmin = tier.xmin()
    self.__n += 1

def read(self, file):
    """ read TextGrid from Praat .TextGrid file """
    rawHeader=open(file,"r")
    encoding_pronostic =  chardet.detect(rawHeader.read())
    rawHeader.close()
    #print "assuming encoding " + encoding_pronostic['encoding'] + ' with
        confidence ' + str(encoding_pronostic['confidence']*100)+'%'
    self.__encoding = encoding_pronostic['encoding']
    text = codecs.open(file, 'r',encoding_pronostic['encoding'])
    text.readline() # header crap
    text.readline()
    text.readline()
    self.__xmin = float(text.readline().strip(' \t').split()[2])
    self.__xmax = float(text.readline().rstrip().split()[2])
    text.readline()
    m = int(text.readline().rstrip().split()[2]) # will be self.__n soon
    text.readline()
    for i in range(m): # loop over grids
        print i
        text.readline()
        meuh = text.readline()
        print meuh
        if meuh.rstrip().split()[2] == '"IntervalTier"':
            inam = text.readline().rstrip().split()[2][1:-1]
            imin = float(text.readline().rstrip().split()[2])
            imax = float(text.readline().rstrip().split()[2])
            itie = IntervalTier(inam, imin, imax) # redundant FIXME
            n = int(text.readline().rstrip().split()[3])
            try:
                for j in range(n):
                    text.readline().rstrip().split() # header junk
                    jmin = float(text.readline().rstrip().split()[2])
                    jmax = float(text.readline().rstrip().split()[2])
                    datatxt = text.readline()
                    jmrk = datatxt.split('"')[1]
                    itie.append(Interval(jmin, jmax, jmrk))
            except IndexError:
                pass
            self.append(itie)
        else: # pointTier or TextTier
            inam = text.readline().rstrip().split()[2][1:-1]
            imin = float(text.readline().rstrip().split()[2])
            imax = float(text.readline().rstrip().split()[2])
            itie = PointTier(inam, imin, imax) # redundant FIXME
            n = int(text.readline().rstrip().split()[3])
            try:
                for j in range(n):
                    text.readline().rstrip() # header junk
                    jtim = float( text.readline().rstrip().split()[2])
                    datatxt = text.readline()
                    jmrk = datatxt.split('"')[1]
                    #jmrk = text.readline().rstrip().split()[2][1:-1]
```

```
                itie.append(Point(jtim, jmrk))
            except IndexError:
                pass
            self.append(itie)
        text.close()

    def write(self, text):
        import sys
        """ write it into a text file that Praat can read """
        print self.__encoding
        text = codecs.open(text, 'w','utf-8')

        text.write('File type = "ooTextFile"\n')
        text.write('Object class = "TextGrid"\n\n')
        text.write('xmin = %f\n' % self.__xmin)
        text.write('xmax = %f\n' % self.__xmax)
        text.write('tiers? <exists>\n')
        text.write('size = %d\n' % self.__n)
        text.write('item []:\n')

        for (tier, n) in zip(self.__tiers, range(1, self.__n + 1)):
            text.write('\titem [%d]:\n' % n)
            if tier.__class__ == IntervalTier:
                text.write('\t\tclass = "IntervalTier"\n')
                text.write('\t\tname = "%s"\n' % tier.name())
                text.write('\t\txmin = %f\n' % tier.xmin())
                text.write('\t\txmax = %f\n' % tier.xmax())
                text.write('\t\tintervals: size = %d\n' % len(tier))
                for (interval, o) in zip(tier, range(1, len(tier) + 1)):
                    text.write('\t\t\tintervals [%d]:\n' % o)
                    text.write('\t\t\t\txmin = %f\n' % interval.xmin())
                    text.write('\t\t\t\txmax = %f\n' % interval.xmax())
                    text.write('\t\t\t\ttext = "%s"\n' % interval.mark())
            else: # PointTier
                text.write('\t\tclass = "TextTier"\n')
                text.write('\t\tname = "%s"\n' % tier.name())
                text.write('\t\txmin = %f\n' % tier.xmin())
                text.write('\t\txmax = %f\n' % tier.xmax())
                text.write('\t\tpoints: size = %d\n' % len(tier))
                for (point, o) in zip(tier, range(1, len(tier) + 1)):
                    text.write('\t\tpoints [%d]:\n' % o)
                    text.write('\t\t\ttime = %f\n' % point.time())
                    text.write('\t\t\tmark = "%s"\n' % point.mark())
        text.close()

class IntervalTier:
    """ represents IntervalTier as a list plus some features: min/max time,
    size, and tier name """

    def __init__(self, name = None, xmin = None, xmax = None):
        self.__n = 0
        self.__name = name
        self.__xmin = xmin
        self.__xmax = xmax
        self.__intervals = []

    def __str__(self):
        return '<IntervalTier "%s" with %d points>' % (self.__name, self.__n)

    def __iter__(self):
        return iter(self.__intervals)

    def __len__(self):
        return self.__n

    def __getitem__(self, i):
        """ return the (i-1)th interval """
        return self.__intervals[i]

    def xmin(self):
        return self.__xmin

    def xmax(self):
        return self.__xmax

    def name(self):
        return self.__name

    def append(self, interval):
        self.__intervals.append(interval)
        if self.__xmax is not None:
```

```
            self.__xmax = max(self.__xmax,interval.xmax())
        else:
            self.__xmax = interval.xmax()
        if self.__xmin is not None:
            self.__xmin = min(self.__xmin,interval.xmin())
        else:
            self.__xmin = interval.xmin()
        self.__n += 1

    def read(self, file):
        text = open(file, 'r')
        text.readline() # header junk
        text.readline()
        text.readline()
        self.__xmin = float(text.readline().rstrip().split()[2])
        self.__xmax = float(text.readline().rstrip().split()[2])
        self.__n = int(text.readline().rstrip().split()[3])
        for i in range(self.__n):
            text.readline().rstrip() # header
            imin = float(text.readline().rstrip().split()[2])
            imax = float(text.readline().rstrip().split()[2])
            imrk = text.readline().rstrip().split()[2].replace('"', '') # txt
            self.__intervals.append(Interval(imin, imax, imrk))
        text.close()

    def write(self, file):
        text = open(file, 'w')
        text.write('File type = "ooTextFile"\n')
        text.write('Object class = "IntervalTier"\n\n')
        text.write('xmin = %f\n' % self.__xmin)
        text.write('xmax = %f\n' % self.__xmax)
        text.write('intervals: size = %d\n' % self.__n)
        for (interval, n) in zip(self.__intervals, range(1, self.__n + 1)):
            text.write('intervals [%d]:\n' % n)
            text.write('\txmin = %f\n' % interval.xmin())
            text.write('\txmax = %f\n' % interval.xmax())
            text.write('\ttext = "%s"\n' % interval.mark())
        text.close()

class PointTier:
    """ represents PointTier (also called TextTier for some reason) as a list
    plus some features: min/max time, size, and tier name """

    def __init__(self, name = None, xmin = None, xmax = None):
        self.__n = 0
        self.__name = name
        self.__xmin = xmin
        self.__xmax = xmax
        self.__points = []

    def __str__(self):
        return '<PointTier "%s" with %d points>' % (self.__name, self.__n)

    def __iter__(self):
        return iter(self.__points)

    def __len__(self):
        return self.__n

    def __getitem__(self, i):
        """ return the (i-1)th tier """
        return self.__points[i]

    def name(self):
        return self.__name

    def xmin(self):
        return self.__xmin

    def xmax(self):
        return self.__xmax

    def append(self, point):
        self.__points.append(point)
        self.__xmax = point.time()
        self.__n += 1

    def read(self, file):
        text = open(file, 'r')
        text.readline() # header junk
        text.readline()
```

```
        text.readline()
        self.__xmin = float(text.readline().rstrip().split()[2])
        self.__xmax = float(text.readline().rstrip().split()[2])
        self.__n = int(text.readline().rstrip().split()[3])
        for i in range(self.__n):
            text.readline().rstrip() # header
            itim = float(text.readline().rstrip().split()[2])
            imrk = text.readline().rstrip().split()[2].replace('"', '') # txt
            self.__points.append(Point(imrk, itim))
        text.close()

    def write(self, file):
        text = open(file, 'w')
        text.write('File type = "ooTextFile"\n')
        text.write('Object class = "TextTier"\n\n')
        text.write('xmin = %f\n' % self.__xmin)
        text.write('xmax = %f\n' % self.__xmax)
        text.write('points: size = %d\n' % self.__n)
        for (point, n) in zip(self.__points, range(1, self.__n + 1)):
            text.write('points [%d]:\n' % n)
            text.write('\ttime = %f\n' % point.time())
            text.write('\tmark = "%s"\n' % point.mark())
        text.close()

class Interval:
    """ represent an Interval """
    def __init__(self, xmin, xmax, mark):
        self.__xmin = xmin
        self.__xmax = xmax
        self.__mark = mark
        self.uid=''

    def __str__(self):
        return '<Interval "%s" %f:%f>' % (self.__mark, self.__xmin, self.__xmax)

    def xmin(self):
        return self.__xmin

    def xmax(self):
        return self.__xmax

    def duration(self):
        return self.__xmax-self.__xmin

    def mark(self):
        return self.__mark

class Point:
    """ represent a Point """
    def __init__(self, time, mark):
        self.__time = time
        self.__mark = mark

    def __str__(self):
        return '<Point "%s" at %f>' % (self.__mark, self.__time)

    def time(self):
        return self.__time

    def mark(self):
        return self.__mark
```

Annexe C

Code source des scripts Matlab

C.1 Script principal des analyses

```
clear all
loaddata

%affichage 3D ou 2D
relief = 0;

%fichiers à tester
ifiles=-1;

%donnees à tester : inPiles = 0 => toutes les syllabes du corpus, à 1 =>
%celles dans les piles
%-------------------------------------------------------------------------
inPiles = 0;

%champs à tester
%---------------

% 1 numero du fichier
% 2 numero de pile
% 3 layer en cours
% 4 position de la syllabe en cours dans le layer
% 5 mean f0
% 6 ralentissement
% 7 nb de parents pile
% 8 nb d'enfants pile
% 9 pile interrompue
% 10 nombre de layers de la pile
% 11 nombre de syllabes dans layer
% 12 syllabe
% 13 prom
% 14 hes
% 15 shape
% 16 localReg
% 17 xmin
% 18 xmax
reffields = [6];
fields    = [6];


%test topologie pile avec enfants/parents/interrompue ou pas de test
%-------------------------------------------------
enfants = 0;
parents = -1;
interrompue = 1;

%nombre de points pour les histogrammes
%-------------------------------------------
Nh = 100;
Nh_diff = 300;

%tests debut-fin-milieu layer: vide pour pas de test, sinon les champs concernés
    à
%tester
%-----------------------------
tests_debut_mid_fin = [5  6];
```

```
%echelle de l'affichage, sous la forme [xmin xmax ymin ymax], si besoin :
%-----------------------------------------
%echelle = [0 5 0 5]; % pour les ralentissements
%echelle = [0 400 0 400];  % pour les f0

%restriction à un layer de fin (commence à 0)
%-----------------------------------------
endLayerTest = -1           %pour les tests inter
samelayer_endLayerTest = -1 %pour les tests intra

% type de test, soit ttest2 soit kstest2:
stat_test = 'ttest2';

%horizon analyse
%-----------------------------------------
horizon = 1


%config x-y
%-----------------------------------------
pos_x = 'input'
pos_y = 'output';


analyseString = {'inter-couches','intra-couches','toutes syllabes'};
for ifile =ifiles
    clear difference
    load couleurs %chargement des couleurs CC, un dérivé maison de 'summer'
    clf
    nFields = length(fields);
    for ifield = 1:length(fields)
        clear Vx
        clear Vy

        for analyse = [0, 1, 2]
            switch analyse
                case 0
                    endlayer_pos = endLayerTest;
                case 1
                    endlayer_pos = samelayer_endLayerTest;
            end

            if (analyse == 0) || (analyse == 1)
                [x,y,dict] = chercheDonneesDansPiles(enfants,parents,ifile,
                    analyse, ...
                        horizon,endlayer_pos,[reffields(ifield), fields(ifield)
                            ]);
            elseif (analyse == 2)
                nomChamp = fieldName{reffields(ifield)};
                Iref = find(ismember(fieldNameAll, nomChamp)==1);
                nomChamp = fieldName{fields(ifield)};
                Ifield = find(ismember(fieldNameAll, nomChamp)==1);

                if isempty(Iref)||isempty(Ifield)
                    continue;
                end

                [x,y,dict] = chercheDonnees(ifile,horizon,[Iref, Ifield]);
            end

            R = size(x,1);

            %affichage des résultats
            refdico = dict{1};
            testdico = dict{2};

            figure(analyse+1)
            clf
            subplot(nFields,1,ifield);

            switch pos_x
                case 'input'
                    XX = x(:,1);
                    xlabelstr= [ fieldName{reffields(ifield)} ' contexte gauche'
                        ];
                case 'output'
                    XX = y(:,1);
                    xlabelstr= [ fieldName{reffields(ifield)} ' contexte droite'
                        ];
            end
```

```
        switch pos_y
            case 'input'
                YY = x(:,2);
                ylabelstr= [ fieldName{fields(ifield)} ' contexte gauche'];
            case 'output'
                YY = y(:,2);
                ylabelstr =[ fieldName{fields(ifield)} ' contexte droite'];
        end

        if ~exist('Vx')
            Vx = [];
        end
        if ~exist('Vy')
            Vy = [];
        end

        if relief
            [h,H,Vx,Vy] = coolhist3D(XX,YY,refdico,testdico,Nh,Vx,Vy);
        else
            [h,H,Vx,Vy] = coolhist2D(XX,YY,refdico,testdico,Nh,Vx,Vy);
        end

        if exist('echelle') && ~isempty(echelle)
            axis(echelle)
        end
        if ~relief
            colorbar
        end

        if isempty(refdico) && isempty(testdico)
            % les deux séries sont non catégorielles, préparation de
            % l'étude de leur différence
            difference{analyse+1}= XX - YY;
            nboccur(analyse+1) = R;
        end

        axis xy;
        grid on
        colormap(CC)
        xlabel(xlabelstr);
        ylabel(ylabelstr);
        if relief
            zlabel('Proportion (%)');
        end

        if ifile >= 0
            stringFile = sprintf('Echantillon %d',ifile+1);
        else
            stringFile = sprintf('Tous les échantillons');
        end

        title(sprintf('%s, %s, %s. (%d occurrences,horizon=%d)',stringFile,
            analyseString{analyse+1},...
                            fieldName{fields(ifield)}, R,horizon));
    end
    if exist('difference')
        figure(10)
        clf
        NN = length(difference);
        AX = [];
        for analyse = 1:NN
            subplot(NN,1,analyse)
            [HH,VV] = hist(difference{analyse},Nh_diff );
            HH = HH/sum(HH(:))*100;
            bar(VV,HH);
            xlabel(sprintf('\\Delta%s',fieldName{fields(ifield)}));
            ylabel('proportion (%)');
            grid on
            title(sprintf('Transitions de %s %s (%d occurrences)',fieldName{
                fields(ifield)},...
                                                analyseString{analyse},
                                                    nboccur(analyse)));
            AX(analyse,:) = axis;
        end

        VFin = [min(AX(:,1)), max(AX(:,2)), 0, max(AX(:,4))];
        for n = 1:NN
            subplot(NN,1,n)
            axis(VFin);
        end
```

```
            tests_a_faire = [1 2; 1 3; 2 3];
            for i = 1:size(tests_a_faire,1)
                if strcmp(stat_test ,'kstest2')
                    [h,pvalue] = kstest2(difference{tests_a_faire(i,1)},
                        difference{tests_a_faire(i,2)});
                    testName = 'Test de  Kolmogorov-Smirnov';
                elseif strcmp(stat_test ,'ttest2')
                    [h,pvalue] = ttest2(difference{tests_a_faire(i,1)},
                        difference{tests_a_faire(i,2)});
                    testName = 'Test de  student';
                end
                if h==1
                    testString = sprintf('Hypothèse de même distribution pour
                        différence \"%s\" et \"%s\" rejetée (%s). p-value=%0.15f
                        ',...
                                            analyseString{tests_a_faire(i,1)
                                                },analyseString{
                                                tests_a_faire(i,2)},testName
                                                ,pvalue);
                else
                    testString = sprintf('Hypothèse de même distribution pour
                        différence \"%s\" et \"%s\" non rejetée (%s). p-value
                        =%0.15f',...
                                            analyseString{tests_a_faire(i,1)
                                                },analyseString{
                                                tests_a_faire(i,2)},testName
                                                ,pvalue);
                end
                disp(testString)
            end


        end
    end

    pause(0.02)
    drawnow
end

if ~isempty(tests_debut_mid_fin)
    Itri = 1:size(syllsInPiles,1);

    if enfants>=0
        Itri = find(syllsInPiles(:,8)==enfants);
    end
    if parents>0
        Itri = Itri(find(syllsInPiles(Itri,7)==parents));
    end
    groupes = cell(size(Itri));
    for i =1:length(groupes)
        if syllsInPiles(Itri(i),4)==0
            groupes{i}='debut de couche';
        elseif syllsInPiles(Itri(i),4)==(syllsInPiles(Itri(i),11)-1)
            groupes{i} ='fin de couche';
        else
            groupes{i}='milieu de couche';
        end
    end

    debuts = Itri(find(syllsInPiles(Itri,4)==0));
    fins = Itri(find(syllsInPiles(Itri,4)==(syllsInPiles(Itri,11)-1)));
    autres = setdiff(Itri,union(debuts,fins));

    for test = tests_debut_mid_fin
        anova1(syllsInPiles(Itri,test),groupes)
        grid on
    end
end
```

C.2 Recherche des données dans la matrice

```
function [inputs, outputs,dictionnaires] = chercheDonneesDansPiles(enfants,
    parents,...
                            ifiles,sameLayer, horizon,endLayerTest,fields)

loaddata

dictionnaires={};
for field = fields
    if dicoCorrespondant(field)
        dictionnaires{end+1} = dicos{dicoCorrespondant(field)}
    else
        dictionnaires{end+1} = {};
    end
end

inputs = [];
outputs = [];

if ifiles>0
    files = ifiles;
else
    files = unique(syllsInPiles(:,1))';
end

for file=files
    Ifile = find(syllsInPiles(:,1)==file);
    piles = unique(syllsInPiles(Ifile,2))';

    for pile = piles
        Ipile = find(syllsInPiles(Ifile,2)==pile);

        if enfants>=0
            Ipile = Ipile(find(syllsInPiles(Ifile(Ipile),8)==enfants));
        end
        if parents>0
            Ipile = Ipile(find(syllsInPiles(Ifile(Ipile),7)==parents));
        end

        layers = unique(syllsInPiles(Ifile(Ipile),3))';

        if endLayerTest>0
            endLayersToTest = layers(find(layers==endLayerTest));
        else
            endLayersToTest = layers;
        end

        for endLayer = endLayersToTest
            if sameLayer
                ILayerSyllables = find(syllsInPiles(Ifile(Ipile),3)==endLayer);
                nSylls = length(ILayerSyllables);

                if (nSylls<2*horizon)
                    continue
                end

                pos=1;
                while pos+2*horizon-1 <= nSylls
                    before = pos:(pos+horizon-1);
                    after = (pos+horizon):(pos+2*horizon - 1);

                    startReference = median(syllsInPiles(Ifile(Ipile(
                        ILayerSyllables(before))),fields),1);
                    endReference = median(syllsInPiles(Ifile(Ipile(
                        ILayerSyllables(after))),fields),1);
                    inputs = [inputs; startReference];
                    outputs = [outputs; endReference];
                    pos=pos+1;
                end
            else
                startLayer = endLayer - 1;
                IstartLayerSyllables = find(syllsInPiles(Ifile(Ipile),3)==
                    startLayer);
                IendLayerSyllables = find(syllsInPiles(Ifile(Ipile),3)==endLayer
                    );

                if (length(IstartLayerSyllables)<horizon) || (length(
                    IendLayerSyllables)<horizon)
                    continue
```

```
                end

                startReference = median(syllsInPiles(Ifile(Ipile(
                    IstartLayerSyllables(end-horizon+1:end))),fields),1);
                endReference = median(syllsInPiles(Ifile(Ipile(
                    IendLayerSyllables(1:horizon))),fields),1);
                inputs = [inputs; startReference];
                outputs = [outputs; endReference];
            end
        end


    end
end
```

C.3 TextGrid Loader

```
function DATA = TGLoader(fileName)
%Reads textgrids with ONLY ONE TIER
fileName
DATA = [];

fid = fopen(fileName,'r');

%Read Introduction Lines
InputText=textscan(fid,'%s',3,'delimiter','\n');
Intro=InputText{1};

%Checks whether file is textgrid
if ~strcmp(Intro{1}, 'File type = "ooTextFile"')
    disp('File is no textgrid, aborting.');
    fclose(fid)
    return;
end

DATA.Intro = Intro;

InputText=textscan(fid,'xmin = %f ',1,'delimiter','\n');
DATA.FileStart =  InputText{1};

InputText=textscan(fid,'xmax = %f ',1,'delimiter','\n');
DATA.FileEnd =  InputText{1};

InputText=textscan(fid,'%s',8,'delimiter','\n','whitespace',' ');
DATA.UnusedStuff=InputText{1};


InputText=textscan(fid,'intervals: size = %d',1,'delimiter','\n');
DATA.NIntervals = InputText{1};

DATA.IntervalData = [];

LastRead = 0;
index = 1;
% pouet = []
% meuh = []

while LastRead < DATA.NIntervals
    InputText=textscan(fid,'intervals [%d]:',1,'delimiter','\n');
    if isempty(InputText{1})
        break;
    end
    DATA.IntervalData(index).Position = InputText{1};

    LastRead = DATA.IntervalData(index).Position;

    InputText=textscan(fid,'xmin = %f ',1,'delimiter','\n');
    DATA.IntervalData(index).xmin = InputText{1};

    InputText=textscan(fid,'xmax = %f ',1,'delimiter','\n');
    DATA.IntervalData(index).xmax = InputText{1};

    InputText=textscan(fid,'%s',2,'delimiter','"') ;

    DATA.IntervalData(index).text = InputText{1}{2};
    InputText=textscan(fid,'%s',1,'delimiter','\n');


%     pouet(end+1) = LastRead;
%     meuh(end+1) = index;
    index = index+1;
%     clf
%     plot(pouet,'b')
%     hold on
%     plot(meuh,'r')
%     drawnow
end

fclose(fid);
DATA.NIntervals = index-1;
%
% if pouet(end) ~= meuh(end)
%     pause
% end
```

C.4 TextGrid Writer

```
function TGWriter(DATA, fileName)
%Writes to textgrids read by TGLoader


fid = fopen(fileName,'w');

for index = 1:length(DATA.Intro)
    fprintf(fid, '%s\n', DATA.Intro{index});
end

fprintf(fid,'xmin = %f  \n', DATA.FileStart);
fprintf(fid,'xmax = %f  \n', DATA.FileEnd);

for index = 1:length(DATA.UnusedStuff)
    fprintf(fid, '%s\n', DATA.UnusedStuff{index});
end

fprintf(fid,'        intervals: size = %d \n', DATA.NIntervals);

for index = 1:DATA.NIntervals

    fprintf(fid,'        intervals [%d]:\n', index);
    fprintf(fid,'            xmin = %f  \n',DATA.IntervalData(index).xmin);
    fprintf(fid,'            xmax = %f  \n',DATA.IntervalData(index).xmax);
    fprintf(fid,'            text = "%s" \n',DATA.IntervalData(index).text);
end
fprintf(fid,'\n');

fclose(fid);
```

C.5 Write TextGrid's transcription to file

```
%Writes a transcription from a DATA structure given by TGLoader to a text
%file.
function WriteTranscriptionToFile(DATA, fileName)

%Will change line each time a new loc is encountered ('$' character)
CHANGE_LINE_CHARACTER = '$';

%Sets an empty string as the beginning transcription
Strings = {''};

%loops over intervals
for index = 1:DATA.NIntervals

    %Gets text of current interval
    CurrentString = DATA.IntervalData(index).text;

    %if string is empty, continue
    if isempty(CurrentString)
        continue;
    end

    %if string starts with a character matching that of the
    %CHANGE_LINE_CHARACTER, adding a new line
    if CurrentString(1) == CHANGE_LINE_CHARACTER
        Strings{end+1} = '';
        Strings{end+1} = CurrentString(1:3);
        Strings{end+1} = '';
        CurrentString(1:4) = [];
    end

    %now adding the current string to the current line (last one)
    if ~isempty(Strings{end})&&(Strings{end}(end) ~= '''')
        Strings{end} = [Strings{end} ' ' CurrentString];
    else
        Strings{end} = [Strings{end} CurrentString];
    end
end

%writing to text file
fid = fopen(fileName, 'w');
for index = 1:length(Strings)
    fprintf(fid, '%s\n',Strings{index});
end
fclose(fid);
```

Index

analyse en grille, 79
Analyse prosodique, 15
Arborator, 57
arbre, 63
attribut, 39

Balisage, 15
binômes irréversibles, 81

classe, 39
co-composition, 80
Co-hyponyme, 83
Confirmation, 85, 86
conjonction, 82
Connexité rectionnelle, 17
coordination, 79, 81
Coordination compositionnelle, 82
Coordination hyperonymique, 83
Coordination intensive, 84
Coordination non relationnelle, 83
coordination non-compositionnelle, 83
Coordinations additives, 82
Coordinations alternatives, 82
corpus, 37
Correction, 86
Critère de proximité, 56

Dépliage, 55
Désambiguïsation, 56
Disfluence, 80
Disfluences, 84
disjonction, 82
Donnée Structurées, 60
durée, 15
DyALog, 41

Entassement, 16
entassement, 79

features, 43
fonction, 39
FRMG, 15, 41

grammaire TAG/TIG, 41
grammaires de construction, 90

hiérarchie, 63
Hyperonyme, 83

instances, 39
Intensification, 84

jonction, 79
Juxtaposition, 83

lexique LeFFF, 41
Liste dialogique, 84
liste paradigmatique, 79

méthode, 39
Méthodes d'instance, 43
macro-syntaxe, 79
micro-syntaxe, 79
micro-syntaxique, 17
Morphème, 90
multi-arbre, 57, 61, 66

Négociation, 85

Parenthétique, 85
Parseur, 38
proéminence, 97
programmation orientée objet, 39
Projection, 69
projection, 58

référent, 83
Réfutation, 86
Répétition, 80
répétitions, 84
racine, 63
ralentissement, 97
Rection, 17
rection, 79
Reformulation, 85
reformulation, 85
Registres locaux, 15
repliage, 56

Structure arborescente, 15
Synonyme, 83

Syntaxe de dépendance, 15
syntaxe de dépendance, 79
Syntaxe de l'écrit, 15
Système d'annotation, 15

Topologie des entassements, 87
transcription, 37

UML, 45
Unité Illocutoire, 17
Unité micro-syntaxique, 17
Unité Rectionnelle, 17

valeurs de F0, 15
variable, 39

Bibliographie

[1] A. Abeillé and D. Godard. La syntaxe de la coordination (special issue). *Langages*, 122, 2005.

[2] C. Astesano. *Rythme et accentuation en français : invariance et variabilité stylistique*. Paris, 2001.

[3] M. Avanzi, A. Lacheret-Dujour, and B. Victorri. Analor : A tool for semi-automatic annotation of french prosodic structure. In *Speech Prosody*, 2008.

[4] M. Avanzi, N. Obin, A. Lacheret, and B. Victorri. Vers une modélisation continue de la structure prosodique. le cas des proéminences accentuelles. *Journal of French Languages Studies*, 1-21, 2011.

[5] M-E. Beckman and J. Pierrehumbert. Intonational structure in japanese and english. *Phonology Yearbook*, 3 :255–309, 1986.

[6] J. Belião. Création d'un multi-arbre à partir d'un texte balisé : l'exemple de l'annotation d'un corpus d'oral spontané. In *RECITAL*, 2012.

[7] J. Belião and A. Liutkus. Rapport technique provisoire des algorithmes utilisés pour le parsing d'un corpus de français oral annoté. Technical report, HAL : halshs-00682283 version 1, 2012.

[8] G. Beller, C. Veaux, G. Degottex, N. Obin, P. Lanchantin, and X. Rodet. Ircam corpus tools : SystÃšme de gestion de corpus de parole. *TAL*, 2009.

[9] C. Benzitoun, A. Dister, K. Gerdes, S. Kahane, and R. Marlet. annoter du des textes tu te demandes si c'est syntaxique tu vois. *The 28th Conference on Lexis and Grammar*, Arena Romanistica 4, Presses de l'Université de Bergen :16–27, 2009.

[10] C. Benzitoun, A. Dister, K. Gerdes, S. Kahane, P. Pietrandrea, and F. Sabio. "tu veux couper là faut dire pourquoi. propositions pour une segmentation syntaxique du français parlé". *Actes du Congrès Mondial de Linguistique française, La Nouvelle Orléans.*, 2010.

[11] A. Berrendonner. *Morpho-syntaxe, pragma-syntaxe et ambivalences sémantiques.* Macro-syntaxe et macro-sémantique : Actes du colloque d'Aarhus, pages 23-41, 2002.

[12] M. Bilger. Coordination : analyses syntaxiques et annotations. *Recherches sur le français parlé*, 15 :255–272, 1999.

[13] C. Blanche-Benveniste. Syntaxe, choix du lexique et lieux de bafouillage. *Documentation et Recherche en Linguistique Allemande Contemporaine*, pages 123–157, 1987.

[14] C. Blanche-Benveniste. Un modèle d'analyse syntaxique 'en grilles' pour les productions orales. *Anuario de Psicologia Liliane Tolchinsky (coord.) Barcelona*, vol. 47 :11–28, 1990.

[15] C. Blanche-Benveniste. Le semblamble et le dissemblable en syntaxe. *Recherches sur le français parlé*, 13 :7–33, 1995.

[16] C. Blanche-Benveniste. Approches de la langue parlée en français. *Paris : Ophrys*, 1997.

[17] C. Blanche-Benveniste, M. Bilger, C. Rouget, and K. Van den Eynd. Le français parlé. études grammaticales. *Paris, CNRS éditions.*, 1990.

[18] C. Blanche-Benveniste, B. Borel, J. Deulofeu, J. Durand, A. Giacomi, C. Loufrani, B. Meziane, and N. Pazery. Des grilles pour le français parlé. *Recherches sur le français parlé*, 2 :163–205, 1979.

[19] P. Boersma and D. Weenink. Praat - doing phonetics by computer, 1997-2012.

[20] E. Bonvino, F. Masini, and P. Pietrandrea. List constructions : a semantic network. In *Troisième Conférence Internationale de l'AFLiCo, Nanterre*, 2009.

[21] K. Carlson. The effects of parallelism and prosody in the processing of gapping structures. *Language and Speech*, 43 (3) :229–259, 2001.

[22] I-D. Craig. *Object-Oriented Programming Languages : Interpretation.* Undergraduate Topics in Computer Science, 2007.

[23] E. Cresti. Corpus di italiano parlato. *Florence, Accademia della Crusca.*, 2000.

[24] M. de Fornel and J-M. Marandin. L'analyse grammaticale des auto-réparations. *Le Gré des Langues*, 10 :8–68, 1996.

[25] E. de La Clergerie. *DyALog Primer : Building tabular parsers and programs*, 2004.

[26] E. de La Clergerie, B. Sagot, L. Nicolas, and M-L. Guénot. Frmg : évolutions d'un analyseur syntaxique tag du franạis. *11th International Conference on Parsing Technologies (IWPT'09)*, 2009.

[27] J. Deulofeu. *Recherches sur les formes de la prédication dans les énoncés assertifs en français contemporain (le cas des énoncés introduits par le morphème que).* PhD thesis, Université Paris 3, 1999.

[28] E. Delais-Roussarie J-M. Marandin H. Yoo F. Mouret, A. Abeillé. Aspects prosodiques des constructions coordonnées du français. In *JEP*, 2008.

[29] C. Féry. German intonational patterns. *Niemeyer*, Tubingen, 1993.

[30] C. Féry. *Indian languages as intonational phrase languages.* Aakar Publisher Delhi, 2010.

[31] C. Féry and G. Kentner. The prosody of embedded coordinations in german and hindi.

[32] C. Féry and F. Kugler. Pitch accent scaling on given, new, and focused constituents in german. *Journal of Phonetics*, 36(4) :680–703, 2008.

[33] C. Féry and F. Schubo. Hierarchical structures in the intonation of recursive sentences. *Postdam*, 2010.

[34] C-J. Fillmore, P. Kay, and M. Catherine O'Connor. Regularity and idiomaticity in grammatical constructions : the case of let alone. *Language*, 64 (3) :501–538, 1988.

[35] L. Frazier, K. Carlson, and C. Clifton. Prosodic phrasing is central to language comprehension. *Trends in Cognitive Sciences*, 10 (6) :244–249, 2006.

[36] C. Gendrot, M. Adda-Decker, and C. Schmid. Comparaison de parole journalistique et de parole spontanée : analyses de séquences entre pauses. In *Actes de la conférence conjointe JEP-TALN-RECITAL 2012, volume 1 : JEP*, pages 649–656, Grenoble, France, June 2012. ATALA/AFCP.

[37] K. Gerdes. Arborator : A tool for collaborative dependency annotation. *http ://arborator.ilpga.fr/vakyartha/*, 2009-2012.

[38] K. Gerdes and S. Kahane. Speaking in piles : Paradigmatic annotation of french spoken corpus. *Proceedings of the Fifth Corpus Linguistics Conference, Liverpool.*, 2009.

[39] A. Goldberg. Constructions. a construction grammar approach to argument structures. *The University of Chicago Press*, 1995.

[40] J-Ph. Goldman. Easyalign : an automatic phonetic alignment tool under praat. In *InterSpeech*, 2011.

[41] A. Grobet and A-C. Simon. Différents critères de définition des unités prosodiques maximales. *Cahiers de Linguistique française*, 23 :143–163, 2001.

[42] A. Grobet and A-C. Simon. Différents critères de définition des unités prosodiques maximales. *Cahiers de Linguistique france*, 23, submitted.

[43] M-L. Guénot. La coordination considérée comme un entassement paradigmatique : description, formalisation et intégration. *TALN, Leuven*, 1 :178–187, 2006.

[44] M. Haspelmath. Coordination. *in T. Shopen (ed.) Language typology and syntactic description. Complex constructions.*, 2 :1–51, 2007.

[45] B. Hurch. Studies on reduplication. *Empirical Approaches to Language Typology*, 28, 2005.

[46] G. Jefferson. List construction as a task and resource. *in G. Psathas (eds) Interactional competence*, pages 63–92, 1991.

[47] A. Joshi, L. Levy, and M. Takahashi. Tres adjunct grammars. *Journal of the computer and system sciences*, 10 :1 :136–163, 1975.

[48] S. Kahane. De l'analyse en grille à la modélisation des entassements. *(à paraitre) Hommage à Claire Blanche-Benveniste, Presses de l'université de Provence.*, in S. Caddeo, M.-N. Roubaud, M. Rouquier, F. Sabio, 2012. in S. Caddeo, M.-N. Roubaud, M. Rouquier, F. Sabio, Hommage à Claire Blanche-Benveniste, Presses de l'université de Provence.

[49] S. Kahane and P. Pietrandrea. Les parenthétiques comme "unités illocutoires associées" une approche macrosyntaxique. *in M. Avanzi & J. Glikman (éd.)*, Les Verbes Parenthétiques : Hypotaxe, Parataxe ou Parenthèse ?, 2011.

[50] S. Kahane and P. Pietrandrea. Typologie des entassements en français. In *Actes de la conférence Linx*, 2012.

[51] P. Kay and C-J. Fillmore. Grammatical constructions and linguistic generalizations : The what's x doing y ? *Language*, 57 :1–33, 1999.

[52] A-S. Kroch and A-K. Joshi. The linguistic relevance of tree adjoining grammar. Technical report, University of Pennsylvania, 1985.

[53] A. Lacheret-Dujour and F. Beaugendre. La prosodie du français. *CNRS Langage, Paris*, 1999.

[54] A. Lacheret-Dujour, S. Kahane, P. Pietrandrea, M. Avanzi, and B. Victorri. Oui mais elle est où la coupure, là ? Quand syntaxe et prosodie s'entraident ou se complètent. *Langue française, Paris-Larousse*, 170 :61–80, 2011.

[55] D. R. Ladd. Declination : a review and some hypothesis. *Phonology Yearbook in Ewen C.J. & Anderson J.M. (éds)*, 1 :53–74, 1984.

[56] K. Lambrecht. Formulaicity, frame semantics, and pragmatics in german binomial expressions. *Language*, 60 (4) :753–796, 1984.

[57] W. Levelt. Monitoring and self-repair in speech. *Cognition*, 14 :41–104, 1983.

[58] C. Loufrani and M-N. Roubaud. La notion d'approximation : langage ordinaire, langage pathologique. *Recherches sur le français parlé*, 10 :131–142, 1990.

[59] Y. Malkiel. Studies in irreversible binomials. *Lingua*, 8 :113–160, 1959.

[60] P. Martin. Winpitch 2000 : a tool for experimental phonology and intonation research. In *Prosody (Workshop)*, 2000.

[61] F. Masini. Binomial constructions : inheritance, specification and subregularities. *Lingue e Linguaggio*, 2 :207–232, 2006.

[62] F. Masini and P. Pietrandrea. Magari. *Cognitive Linguistics*, 21(1) :75–121, 2010.

[63] C. Mauri. *Coordination relations in the languages of Europe and beyond*. Berlin, 2008.

[64] P. Mertens. *L'intonation du français. De la description linguistique à la reconnaissance automatique*. PhD thesis, KULeuven, 1987.

[65] P. Mertens. The prosogram : Semi-automatic transcription of prosody based on a tonal perception model. In *Speech Prosody*, 2004.

[66] L-A. Michaelis and K. Lambrecht. The exclamative sentence type in english. *In A. Goldberg, ed. Conceptual Structure, Discourse and Language*, pages 375–389, 1996.

[67] M-A. Morel and L. Danon-Boileau . Grammaire de l'intonation. *Ophrys*, Paris-Gap, 1998.

[68] M-A. Morel and L. Danon-Boileau. Grammaire de l'intonation. In *Paris-Gap, Ophrys*, 1998.

[69] N. Obin. *MeLos : Analysis and Modelling of Speech Prosody and Speaking Style*. Thèse de doctorat, Ircam-UPMC, Paris, 2011.

[70] N. Obin, A. Lacheret-Dujour, C. Veaux, X. Rodet, and A-C. Simon. A method for automatic and dynamic estimation of discourse genre typology with prosodic features. *Interspeech*, 2008.

[71] M. Overstreet. And stuff und so : Investigating pragmatics expressions in english and german. *Journal of Pragmatics*, 37 :1845–1864, 2005.

[72] U. Patil, G. Kentner, A. Gollrad, F. Kugler, C. Féry, and S. Vasishth. Focus, word order and intonation in hindi. *Journal of South Asian Linguistics*, 1 :55–72, 2008.

[73] P. Pietrandrea. List constructions. 2012.

[74] P. Pietrandrea and S. Kahane. La typologie des entassements. 2012.

[75] R Development Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[76] Rhapsodie. Site du projet rhapsodie, corpus prosodique de référence en français parlé. *http ://rhapsodie.risc.cnrs.fr*, 2012.

[77] M. Rossi. L'intonation. le système du français : description et modélisation. In *Paris-Gap, Ophrys*, 1999.

[78] B. Sagot. The lefff , a freely available and large-coverage morphological and syntactic lexicon for french. In *LREC'10*, 2010.

[79] J. Sauvage-Vincent and A. Lacheret. Protocole de codage prosodique : Annotation des proéminences, disfluences et périodes dans le projet rhapsodie. Technical report, Université Paris Ouest Nanterre La Défense, 2012.

[80] A. Schafer, S. Speer, P. Warren, and S-D. White. Intonational disambiguation in sentence production and comprehension. *Journal of Psycholinguistic Research*, 29 (2) :169–182, 2000.

[81] M. Selting. Lists as embedded structures and the prosody of list construction as an interactional resource. *Journal of Pragmatics*, 39 :483–526, 2007.

[82] S-M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8 :333–343, 1985.

[83] Keng Siau and Terry A. Halpin, editors. *Unified Modeling Language : Systems Analysis, Design and Development Issues*. Idea Group, 2001.

[84] A-C. Simon and A. Grobet. Réinitialisations (resetting) et unités prosodiques maximales : une évidence ? In *Journées Prosodie, Grenoble*, 2001.

[85] T. Stolz. (wort-)iteration : (k)eine universelle konstruktion. *in K. Fischer and A. Stefanowitsch (eds.) Konstruktionsgrammatik. Von der Anwendung zur Theorie*, pages 105–32, 2006.

[86] T. Stolz. Das ist doch keine reduplikation ! uber falsche freunde bei der suche nach richtigen beispielen. *in A. Ammann and A. Urdze (eds.) Wiederholung, Parallelismus, Reduplikation. Strategien der multiplen Strukturanwendung, Bochum, Universitatsverlag Dr. N. Brockmeyer*, pages 47–80, 2007.

[87] T. Stolz. Total reduplication vs. echo-word formation in language contact situations. *in P. Siemund and N. Kintana (eds.), Language Contact and Contact Languages. Hamburg Studies on Multilingualism*, 7 :107–32, 2008.

[88] D. Tannen. Talking voices : repetition, dialogue, and imagery in conversational discourse. *Cambridge University Press*, 1991 (2007).

[89] L. Tesnière. *éléments de syntaxe structurale*. Librairie C. Klincksieck, 1959.

[90] J. Vaissière. *Le français, langue à frontières par excellence dans Frontièr, du linguistique au sémiotique*. Ouvrage collectif initié par Nelly Andrieux-Reix, dirigé et édité par Dominique Delomier en collaboration avec Mary-Annick Morel, 2010.

[91] B. Walchli. Co-compounds and natural coordination. In *Oxford University Press*, 2005.

[92] G. Yule. Speakers' topics and major paratones. *Lingua*, 52 :33–47, 1980.