Apprentissage de grammaires catégorielles : transducteurs d'arbres et clustering pour induction de grammaires catégorielles

Noémie-Fleur Sandillon-Rezer

N° d'ordre : 4940

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Noémie-Fleur SANDILLON-REZER**

Pour obtenir le grade de

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Apprentissage de Grammaires Catégorielles

Transducteurs d'arbres et clustering pour induction de grammaires catégorielles

---

Soutenance prévue le : 09/12/2013

Devant la commission d'examen composée de :

Christian RETORÉ	Professeur	Directeur de Thèse
Richard MOOT	Chargé de Recherche CNRS	Co-Directeur de Thèse
Annie FORET	Maître de Conférences	Rapporteur
Mark STEEDMAN	Professeur	Rapporteur
Laurence DANLOS	Professeur	Examinateur
Alexis NASR	Professeur	Examinateur
Géraud SÉNIZERGUES	Professeur	Examinateur
Tim van de CRUYS	Chargé de Recherche CNRS	Examinateur

– 2013 –

Remerciements

Finir un doctorat me semble quelque chose de terrible et fantastique. J'ai l'impression que ces trois années sont passées trop vite, et que le début de thèse était encore hier : hier encore, je cherchais un stage de recherche, qui m'a poussé vers mon sujet actuel.

Je souhaiterais donc remercier Richard Moot, qui me suit depuis ce fameux stage, pour son soutien et ses idées tout au long de ces années. Cela a été un plaisir aussi bien qu'un enrichissement intellectuel de passer tout ce temps à travailler sous sa direction ; j'ai aussi adoré nos discussions moins formelles, qui m'ont appris énormément. Je n'aurais pas pu faire cette thèse sans l'aide et les conseils de Christian Retoré. Il a toujours su prendre de son temps pour m'éclairer et discuter : je sais que je peux toujours compter sur lui.

Géraud Sénizergues m'a, tant par ses cours que par les échanges que nous avons eu, poussée vers la recherche. Je sais que s'il ne m'avait pas intéressée à l'informatique fondamentale, je n'aurais pas voulu partir vers ce domaine. J'espère de tout coeur devenir un jour un enseignant tel que lui, capable faire éclore des vocations comme il l'a fait avec moi.

Je remercie aussi le jury : Laurence Danlos, Annie Foret, Alexis Nasr, Géraud Sénizergues, Mark Steedman et Tim van de Cruys, pour avoir accepté d'évaluer mon travail et de relire celui-ci. Leurs commentaires de pré-soutenance m'ont aidée à améliorer ce manuscrit, et les discussions ayant suivi celle-ci m'ont donné de nombreuses idées et pistes pour continuer ces travaux de recherche.

A mes relecteurs moins officiels qui ont pris le temps de lire le manuscrit sans faire partie du domaine, ni parfois même du monde de la recherche, je présente mes remerciements les plus sincères. Toute erreur restant dans ce manuscrit ne saurait leurs être imputée.

Je remercie également toutes les personnes, famille aussi bien qu'amis, qui ont assisté à la soutenance. Je sais que le sujet a pu leur paraître complexe, mais les voir tous m'a rendu plus sûre de moi devant le jury.

Tout au long de cette thèse, j'ai été soutenue par de nombreuses personnes. Ainsi, je tiens à saluer Jérôme, Florent, Luc, Raphaël et Alyx, puis Tom, Simon et Jérôme (pas le même), qui ont été les cobureaux les plus merveilleux que l'on puisse imaginer. Bien sûr, je remercie tous les doctorants que j'ai pu croiser au LaBRI, ceux avec qui je me suis retrouvée en conférence, tels qu'Ophélie, Anaïs, Simon, Jérôme et Pierre, les membres de l'AFoDIB et ceux du LaBRuIt.

Ces trois dernières années, j'ai découvert la musique et j'ai pu continuer à faire travailler mon imagination : Les (not) Metallic Smurfs*, les NFOR, les membres de LTA et DT&C, les étudiants et le personnel de la* Soucoupe*, Salomé ainsi que ma table de jeu de rôle m'ont permis de réserver quelques plages au rêve.*

Je garde ce paragraphe pour mes parents et ma soeur, qui m'ont toujours soutenue dans mes choix, quels qu'ils soient, et qui m'ont réellement aidée pendant les périodes les plus dures de rédaction et de préparation de soutenance. Mon meilleur ami Michaël a eu la patience de me supporter depuis le début de nos études à Bordeaux. J'espère de tout coeur que nous pourrons continuer à nous voir aussi régulièrement qu'actuellement.

Simon a rendu ces six dernières années merveilleuses. Il a contribué à mon épanouissement personnel et intellectuel, et je sais qu'il ne réalise pas à quel point il m'a poussée à avancer, ne serait-ce que pour arriver à son niveau. Le simple fait d'avoir un homme aussi fantastique à mes côtés me rends heureuse : nous irons au bout du monde ensemble.

Résumé

De nos jours, il n'est pas rare d'utiliser des logiciels capables d'avoir une conversation, d'interagir avec nous (systèmes questions/réponses pour les SAV, gestion d'interface ou simplement Intelligence Artificielle - IA - de discussion). Ceux-ci doivent comprendre le contexte ou réagir par mot-clefs, mais générer ensuite des réponses cohérentes, aussi bien au niveau du sens de la phrase (sémantique) que de la forme (syntaxe). Si les premières IA se contentaient de phrases toutes faites et réagissaient en fonction de mots-clefs, le processus s'est complexifié avec le temps. Pour améliorer celui-ci, il faut comprendre et étudier la construction des phrases.

Nous nous focalisons sur la syntaxe et sa modélisation avec des grammaires catégorielles. L'idée est de pouvoir aussi bien générer des squelettes de phrases syntaxiquement correctes que vérifier l'appartenance d'une phrase à un langage, ici le français (il manque l'aspect sémantique). On note que les grammaires AB peuvent, à l'exception de certains phénomènes comme la quantification et l'extraction, servir de base pour la sémantique en extrayant des λ-termes.

Nous couvrons aussi bien l'aspect d'extraction de grammaire à partir de corpus arborés que l'analyse de phrases. Pour ce faire, nous présentons deux méthodes d'extraction et une méthode d'analyse de phrases permettant de tester nos grammaires.

La première méthode consiste en la création d'un transducteur d'arbres généralisé, qui transforme les arbres syntaxiques en arbres de dérivation d'une grammaire AB. Appliqué sur les corpus français que nous avons à notre disposition, il permet d'avoir une grammaire assez complète de la langue française, ainsi qu'un vaste lexique. Le transducteur, même s'il s'éloigne peu de la définition usuelle d'un transducteur descendant, a pour particularité d'offrir une nouvelle méthode d'écriture des règles de transduction, permettant une définition compacte de celles-ci. Nous transformons actuellement 92,5% des corpus en arbres de dérivation.

Pour notre seconde méthode, nous utilisons un algorithme d'unification en guidant celui-ci avec une étape préliminaire de *clustering*, qui rassemble les mots en fonction de leur contexte dans la phrase. La comparaison avec les arbres extraits du transducteur donne des résultats encourageants avec 91,3% de similarité.

Enfin, nous mettons en place une version probabiliste de l'algorithme CYK pour tester l'efficacité de nos grammaires en analyse de phrases. La couverture obtenue varie entre 84,6% et 92,6%, en fonction de l'ensemble de phrases pris en entrée. Les probabilités, appliquées aussi bien sur le type des mots lorsque ceux-ci en ont plusieurs que sur les règles, permettent de sélectionner uniquement le "meilleur" arbre de dérivation.

Tous nos logiciels sont disponibles au téléchargement sous licence `GNU GPL`.

Mots clefs : Apprentissage automatique sur corpus, grammaires catégorielles, analyse de phrases, transducteur d'arbres.

LaBRI
(UMR CNRS 5800)
351, cours de la Libération
33405 Talence cedex
France

Learning Categorial grammars

Abstract

Nowadays, we have become familiar with software interacting with us using natural language (for example in question-answering systems for after-sale services, human-computer interaction or simple discussion bots). These tools have to either react by keyword extraction or, more ambitiously, try to understand the sentence in its context. Though the simplest of these programs only have a set of pre-programmed sentences to react to recognized keywords (these systems include Eliza but also more modern systems like Siri), more sophisticated systems make an effort to understand the structure and the meaning of sentences (these include systems like Watson), allowing them to generate consistent answers, both with respect to the meaning of the sentence (semantics) and with respect to its form (syntax).

In this thesis, we focus on syntax and on how to model syntax using categorial grammars. Our goal is to generate syntactically accurate sentences (without the semantic aspect) and to verify that a given sentence belongs to a language - the French language. We note that AB grammars, with the exception of some phenomena like quantification or extraction, are also a good basis for semantic purposes.

We cover both grammar extraction from treebanks and parsing using the extracted grammars. On this purpose, we present two extraction methods and test the resulting grammars using standard parsing algorithms.

The first method focuses on creating a generalized tree transducer, which transforms syntactic trees into derivation trees corresponding to an AB grammar. Applied on the various French treebanks, the transducer's output gives us a wide-coverage lexicon and a grammar suitable for parsing. The transducer, even if it differs only slightly from the usual definition of a top-down transducer, offers several new, compact ways to express transduction rules. We currently transduce 92.5% of all sentences in the treebanks into derivation trees.

For our second method, we use a unification algorithm, guiding it with a preliminary clustering step, which gathers the words according to their context in the sentence. The comparision between the transduced trees and this method gives the promising result of 91.3% of similarity.

Finally, we have tested our grammars on sentence analysis with a probabilistic CYK algorithm and a formula assignment step done with a supertagger. The obtained coverage lies between 84.6% and 92.6%, depending on the input corpus. The probabilities, estimated for the type of words and for the rules, enable us to select only the "best" derivation tree.

All our software is available for download under `GNU GPL` licence.

Keywords : Language learning, categorial grammar, parsing, tree transducers

Table des matières

Chapitre 1

Introduction

En 1900 déjà, L. Frank Baum créait *Le Magicien d'Oz* et ses personnages novateurs, dont "l'homme en fer blanc", qui peut être considéré comme le premier robot parlant. Le terme "robot" est apparu cependant une vingtaine d'années plus tard, dans la pièce de théâtre de science-fiction *Rossum's Universal Robots*, de Karel Čapek, pour désigner un travailleur, un esclave. Il a fallu attendre les années 50 pour que, grâce à Asimov et son *Cycle des robots*, l'image de robots, intelligents capables d'interaction parfaite avec l'homme, aussi bien par le biais des actes que de la parole, devienne une image répandue.

Cependant, la réalité scientifique est toute autre. Pour l'instant nous sommes encore bien loin des performances dont on imaginait ces robots capables il y a de cela plus d'un siècle, surtout au niveau de la modélisation d'une langue. Si des prouesses quant aux mouvements et déplacements que peuvent effectuer ces robots ont été réalisées, on est encore loin des prouesses imaginées dans le domaine de la science-fiction quant aux fonctions cognitives, et notamment le langage. Sans perdre espoir cependant, il faut reconnaître les progrès qui ont été faits dans le domaine. De nos jours, il n'est pas rare d'utiliser des logiciels capables d'avoir une conversation ou d'interagir avec nous, tels que les systèmes de questions/réponses pour les services après-vente, les gestions vocales d'interface ou simplement les Intelligences Artificielles de discussion pour le loisir. Ceux-ci doivent comprendre le contexte ou réagir par mot-clefs, mais aussi générer par la suite des réponses cohérentes en temps réel, aussi bien au niveau du sens de la phrase (sémantique) que de la forme (syntaxe). Si les premières Intelligences Artificielles se contentaient de phrases toutes faites ou de phrases à trous et réagissaient en fonction de mots-clefs donnés par l'utilisateur dans sa conversation, le processus s'est complexifié avec le temps. Pour améliorer celui-ci, il faut comprendre et étudier la construction des phrases.

Pour notre part, nous avons décidé de nous focaliser sur la syntaxe des phrases et sa modélisation avec des grammaires catégorielles. Les grammaires catégorielles présentent l'avantage de représenter élégamment les règles d'une

langue et de permettre de reconnaître l'appartenance d'une phrase à un langage en assignant à chaque mot une catégorie. De plus, ces catégories permettent d'avoir une première approche de la sémantique des mots, via les λ-termes [1]. Nous sommes partis du constat qu'il existait des mines d'informations syntaxiques sur le français, sous forme de corpus arborés ([Abeillé *et al.*, 2003; Candito et Seddah, 2012]) et que celles-ci devaient pouvoir servir de guide pour formaliser une grammaire de la langue française. Si l'idée d'adapter des algorithmes déjà existants pour les données d'entrée que nous avions était notre point de départ, nous nous sommes rapidement rendu compte que convertir les données pour qu'elles correspondent au bon format et ajuster les algorithmes existants ne garantissait pas un résultat satisfaisant.

Nous avons donc décidé de créer nos propres algorithmes d'extraction de grammaires catégorielles, plus précisément de grammaires AB, à partir des corpus du français. Les annotations syntaxiques précises des corpus nous ont guidé tout au long de notre travail d'extraction. Ces grammaires, ensuite, sont utilisées pour l'analyse de phrases, de manière à savoir si elles appartiennent ou non à une langue, ce qui est le but de toute extraction de grammaire.

Les grammaires catégorielles

Les grammaires catégorielles permettent de reconnaître facilement les phrases des non-phrases, c'est à dire de savoir si un mot appartient à un langage formel ou si une phrase appartient à un langage naturel. Leurs origines remontent à Ajdukiewicz [1935] et Bar-Hillel [1953], et elles ne contenaient que deux règles : l'élimination à droite et l'élimination à gauche. Le principe était d'associer aux mots d'une phrase des catégories, ou types, et en utilisant les deux règles d'élimination de trouver la catégorie finale correspondant au tout. Si la catégorie finale était s, pour *sentence*, la phrase appartenait au langage et s'il était impossible de combiner les types d'une quelconque manière ou si le résultat n'était pas s, la phrase était considérée comme fausse.

Les grammaires AB (pour Ajdukiewicz et Bar-Hillel) ont été étendues par Lambek [1958], en ajoutant les règles d'introduction et le produit. Peu à peu, d'ailleurs, ce modèle a été modifié et amélioré, pour répondre à des usages plus spécifiques d'une langue ou d'une autre. Ainsi, les grammaires de lambek non-associatives ont été proposées par Lambek [1961], et en ont découlé le calcul de Lambek-Grishin [Bernardi et Moortgat, 2010] ou encore les grammaires de Lambek multimodales [Moortgat et Oehrle, 1994].

Quelle que soit la formalisation des grammaires catégorielles que l'on décide d'employer, il convient de trouver un équilibre entre la complexité des dériva-

1. Les catégories associées aux mots permettent d'avoir le type du λ-terme correspondant à ceux-ci

tions et l'apport des règles d'introduction et le produit (utilisé principalement dans les travaux de [de Groote, 1999; Moot, 2013; Morrill, 2013]).

Pour notre part, nous avons pris le parti d'utiliser des grammaires AB, car cela nous semblait capital que nos résultats soient compatibles avec des algorithmes d'apprentissage reconnus ([Buszkowski et Penn, 1990; Kanazawa, 1998]). De plus, les structures d'entrée que nous avions, c'est à dire des corpus arborés (initialement uniquement le corpus de Paris VII [Abeillé *et al.*, 2003] puis aussi le corpus Séquoia [Candito et Seddah, 2012]), donnaient toutes les informations dont nous pouvions avoir besoin pour en extraire une grammaire AB. Les règles d'introduction, sans parler du produit, ne sont pas utilisables avec les corpus en l'état, car il manque de nombreuses informations, telles que les traces. Les grammaires AB sont devenues alors un choix naturel, car cadrant plus avec nos structures d'entrées. Il faut garder à l'esprit cependant que certains phénomènes linguistiques ne peuvent pas être représentés facilement avec ces grammaires, tels que les inversions, les extractions ou encore la quantification, d'autant plus que nous ne pouvons pas retrouver cette information dans les corpus arborés.

Inférence grammaticale

Les algorithmes d'inférence grammaticale permettent, à partir d'une structure donnée, d'apprendre une grammaire. Il en existe de très nombreux, comme nous le verrons plus tard, mais on peut séparer ces algorithmes en trois catégories, en fonction des structures prises en entrée.

Méthodes n'utilisant pas de structure de départ. Les méthodes développées par Adriaans [2001] et Adriaans *et al.* [2000] partent de phrases sans structure et permettent d'apprendre soit une grammaire hors contexte soit une grammaire AB. Leur algorithme d'apprentissage utilise des matrices qui, pour chaque mot et groupe de mots, notent le contexte dans lequel ceux-ci apparaissent ; cela marche dans de nombreux cas, mais il est délicat pour de telles grammaires d'inférer correctement des cas complexes comme l'attachement des syntagmes prépositionnels. Etant donné que ces informations sont annotées correctement dans les corpus tels que ceux de Paris VII [Abeillé *et al.*, 2003] ou Séquoia [Candito et Seddah, 2012], il nous semblait logique de les exploiter.

Méthodes avec structures partielles. Elles sont décrites par des algorithmes tels que ceux de Buszkowski et Penn [1990] ou Kanazawa [1998], et sont clairement dans un paradigme de l'apprentissage à la limite de Gold [1967]. Les structures d'entrées, décrites plus en détail dans l'état de l'art, sont sous

forme d'arbres (ou FA-structures) et la sortie des algorithmes sont une grammaire rigide dans le cas de Buszkowski et Penn ou une grammaire k-valuée dans le cas de Kanazawa. Les grammaires rigides, hélas, ne sont pas représentatives des langues naturelles et, lorsque $k \geq 2$, résoudre l'algorithme d'unification est NP-dur [Costa-Florêncio, 2001]. Cependant, même une grammaire 2-valuée ne peut pas représenter une langue naturelle : les expériences d'extraction de grammaires montrent que le nombre maximal de catégories par mots est souvent élevé, avec de nombreux mots courants (typiquement les conjonctions de coordination) dont le nombre de type dépasse quarante [Hockenmaier et Steedman, 2007; Sandillon-Rezer et Moot, 2011].

Méthodes avec structures totalement définies. Ces méthodes utilisent des structures totalement définies, comme celle de Hockenmaier [2003], qui donne comme résultat une grammaire catégorielle combinatoire. L'idée utilisée est que le corpus arboré utilisé en entrée, qui contient des informations syntaxiques à chaque nœud (avec les mots sur les feuilles), permet de guider l'étape d'extraction de grammaire. La méthode d'Hockenmaier utilise le corpus de Penn [Marcus *et al.*, 1993] qui couvre 49 208 phrases.

Contributions et organisation du mémoire

Ce mémoire suit la progression logique de notre travail. Après une présentation des travaux antérieurs dans le domaine de l'apprentissage ainsi qu'un rappel des notions clefs des grammaires catégorielles, nous présentons nos deux méthodes d'apprentissage de grammaires AB ainsi que notre analyseur de phrases.

Chapitre 1 - Etat de l'art. Ce chapitre est découpé en trois parties. Tout d'abord, nous présentons en détail les grammaires AB [Ajdukiewicz, 1935; Bar-Hillel, 1953] et leurs évolutions que sont les grammaires de Lambek [Lambek, 1958], les grammaires de Lambek multimodales [Moortgat, 1997], les grammaires catégorielles abstraites [de Groote, 2001] et les grammaires catégorielles combinatoires [Steedman, 2001]. Ces outils sont communément utilisés pour reconnaître les phrases des non-phrases, par rapport à une langue naturelle. La seconde partie de l'état de l'art se focalise donc naturellement sur les différents algorithmes d'apprentissage suivant le modèle de Gold [1967], plus particulièrement l'algorithme de Buszkowski et Penn [1990] qui apprend une grammaire rigide, c'est à dire dont les mots du lexique ont au plus un type. Nous expliquons aussi le fonctionnement de l'algorithme de Kanazawa [1998], qui lui apprend, par unfication, des grammaires k-valuées (on autorise les mots du lexique à avoir k types au plus). Enfin, nous présentons les corpus sur lesquels

nous avons effectué nos expérimentations, c'est à dire le corpus de Paris VII [Abeillé *et al.*, 2003], le corpus Séquoia [Candito et Seddah, 2012] et, utilisé uniquement dans l'étape d'analyse de phrases, le corpus de l'Est Républicain [Gaiffe et Nehbi, 2009]. Les deux premiers présentent les phrases sous forme d'arbres syntaxiques alors que le dernier consiste en une collection de phrases.

Chapitre 2 - Transducteur pour inférence grammaticale. Ici, nous présentons l'évolution des transducteurs d'arbres descendants que nous utilisons. La différence majeure entre un transducteur usuel et notre transducteur réside dans la manière plus compacte que nous avons d'écrire les règles, sans perdre aucune caractéristique du transducteur. La paramétrisation des règles qui permet de les factoriser, la récursivité qui autorise les nœuds à avoir un nombre arbitraire de fils et le système de priorité qui nous assure d'avoir toujours les mêmes résultats en sortie, (en un mot il assure le déterminisme de notre transducteur). Outre l'aspect théorique du transducteur, nous expliquons le fonctionnement des règles que nous avons mises en place, puis donnons les résultats sur les deux corpus arborés.

Chapitre 3 - Inférence grammaticale via clustering. Dans l'idée de créer une méthode d'apprentissage suivant le modèle de Gold, nous nous sommes penchés sur les algorithmes d'unification. Nous apportons deux modifications majeures aux algorithmes usuels vus dans l'Etat de l'art. La première est la quantité d'informations contenues dans les arbres pris en entrée par notre algorithme. Au lieu de savoir simplement quel nœud sera le foncteur et l'argument, nous fixons certains types, en nous fondant sur les résultats donnés par le lexique du transducteur. Ainsi, les arbres contiendront aussi bien des variables que des types déjà totalement définis, ou un mélange des deux. La seconde modification est l'utilisation de vecteurs extraits des arbres syntaxiques des phrases pour effectuer une étape de clustering. Le clustering nous permet d'unifier les mots en priorité en fonction de la similitude des contextes dans lesquels ils apparaissent. L'implémentation et les essais de cette méthode sont ensuite discutés. De même, la preuve de convergence à la Gold est donnée.

Chapitre 4 - Analyse de phrases. L'extraction de grammaire n'a réellement d'intérêt que si l'on teste ensuite le résultat dans l'analyse à grande échelle de phrases. C'est pour cela que nous avons mis en place une implémentation probabiliste de l'algorithme CYK. Etant donné que les arbres de dérivation sont déjà binaires, nous pouvons extraire de ceux-ci une grammaire hors contexte probabiliste, correspondant à une grammaire AB, déjà en forme normale de Chomsky. Les probabilités sur les règles sont calculées en fonction de l'occurrence d'une règle par rapport à toutes les règles qui ont la même racine. Le typage des phrases s'effectue majoritairement avec le Supertagger

[Moot, 2010a; Clark et Curran, 2007], ce qui permet d'avoir aussi une probabilité sur les types donnés aux mots, en fonction d'un paramètre β réglé par l'utilisateur. Les probabilités des règles et des types des mots permettent de régler le problème de génération de très nombreux arbres de dérivation. Nous testons ensuite les différentes grammaires extraites avec nos deux méthodes précédentes et évaluons la justesse de chacune.

Chapitre 2

Etat de l'art

2.1 Grammaires catégorielles

Les grammaires catégorielles permettent de reconnaître les phrases des non-phrases, que ce soient des phrases d'un langage formel (on parlera alors de mots) ou d'un langage naturel. Le principe sera de donner des catégories, ou types, aux mots, de telle sorte que lorsque l'on assemble les bons mots la phrase finale a pour type s (pour *sentence*).

2.1.1 Grammaires AB

Les grammaires AB ont été créées par Ajdukiewicz [1935] et Bar-Hillel [1953], ce qui leur a valu leur nom. Ce sont des grammaires algébriques, comme vu dans [Bar-Hillel, 1953] (la preuve est donnée en premier lieu dans [Bar-Hillel, 1953], une version plus moderne est présentée dans [Moot et Retoré, 2012]). Ce sont les premières grammaires catégorielles connues, et elles permettent déjà d'analyser et de représenter la syntaxe d'une langue. Elles sont constituées de catégories, associées aux mots, et de deux règles permettant d'assembler les catégories.

Les catégories peuvent être aussi complexes qu'on le souhaite, en partant d'un petit ensemble de base, noté $\mathcal{P}$. L'ensemble de toutes les catégories possibles sera noté $\mathcal{L}$ et composé ainsi :

$$\mathcal{L} := \mathcal{P} \mid \mathcal{L}\backslash\mathcal{L} \mid \mathcal{L}/\mathcal{L}$$

C'est à dire qu'à partir de notre ensemble de catégories basiques, on peut composer via le / et le \ toute catégorie possible, quelle que soit sa complexité (c'est à dire son nombre de / et \). De même, rien ne nous empêche de rajouter autant de catégories de base à l'ensemble $\mathcal{P}$.

Initialement $\mathcal{P} = \{s, np, n\}$ pour respectivement :

s **:** une phrase complète, telle que *"Jean a lu un livre triste"*.

np **:** un groupe nominal ou un nom propre, tels que *"un livre"*, *"un livre triste"* ou *"Jean"*.

n **:** un nom commun comme *"livre"*. Par extension, on va utiliser aussi cette catégorie pour un groupe nominal dont on a déjà retiré le déterminant comme *"livre triste"*.

Les deux règles d'élimination à droite et à gauche, représentées par le / et le \ :

$$A/B \quad B \to A \qquad B \quad B\backslash A \to A$$

Ces règles peuvent être représentées aussi comme des règles de déduction naturelle :

$$\frac{B \quad B\backslash A}{A}\,[\backslash E] \qquad \frac{A/B \quad B}{A}\,[/E]$$

La phrase :

John	aime	Mary
np	$(np\backslash s)/np$	np

est correcte car le verbe "aime" prend en argument deux phrases nominales (représentées par *np* pour noun phrase), et que "John" et "Mary" ont le type *np*.

Dans le cas présent, on comprend alors que le verbe "aime" cherche en premier lieu un groupe nominal *np* à sa droite, ce qui lui est donné par "Mary". Le groupe de mots "aime Mary" devient alors de type np/s, c'est à dire que pour devenir une phrase à part entière il lui faut quelque chose typé *np* à gauche. Lorsqu'on ajoute "John" de type *np*, on obtient le type *s*, ce qui prouve que l'on a choisi les bons types des mots.

L'ensemble des types associés aux mots représentera une grammaire catégorielle, sous forme d'un lexique. Ledit lexique donne les informations structurelles nécessaires pour savoir si une phrase est correcte ou non.

Les dérivations, quant à elles, peuvent être représentées comme une preuve de logique ou comme un *arbre de dérivation*. C'est cette seconde forme que nous utiliserons le plus souvent dans ce mémoire.

Les arbres de dérivation représentent les règles appliquées pour analyser la phrase. Ils sont binaires et leur racine est habituellement étiquetée avec un des types précisant que la phrase est correcte [1]. Les mots composant la phrase sont précisés sur les feuilles, ainsi que leur type.

1. Habituellement, ce type est uniquement *s*. Nous verrons plus tard que nous utilisons aussi le type *txt* pour différencier les phrases ayant une ponctuation finale de celles qui n'en ont pas, les deux étant considérées comme correctes.

En guise d'exemple, nous allons nous focaliser sur la phrase *"CSF a créé un journal"*. Le lexique correspondant est :

CSF	np
a	$(np\backslash s)/(np\backslash s)$
créé	$(np\backslash s)/np$
un	np/n
journal	n

Le type complexe $(np\backslash s)/np$, associé à *créé*, signifie que le participe passé prend en argument à sa droite un nœud np (son objet), pour créer un nouveau type, $np\backslash s$. Celui-ci sera pris en argument du verbe auxiliaire a, pour donner comme nouveau type $np\backslash s$.

Ce lexique permet de dériver la phrase, et nous présentons à la fois l'arbre de dérivation que nous utiliserons ensuite (voir figure 2.1) et la version en déduction naturelle (figure 2.2). Chaque feuille de l'arbre de dérivation correspond à une entrée lexicale (indiquée sur la figure 2.2 par la règle unaire *Lex*), et chaque sous-arbre local correspond à une des règles d'élimination à droite ou à gauche. La racine est étiquetée s, ce qui montre que la dérivation est correcte.

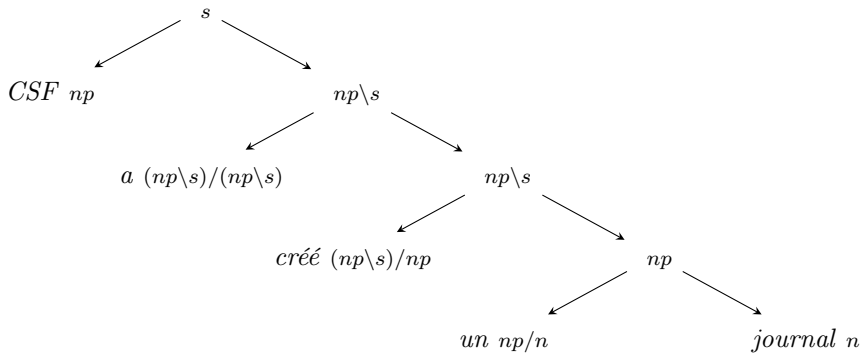


FIGURE 2.1 – Arbre de dérivation de *"CSF a créé un journal"*.

On remarque, en utilisant les types de base donnés par Lambek, que *"CSF*

$$\dfrac{\dfrac{\dfrac{\text{CSF}}{np}\,[Lex] \quad \dfrac{\dfrac{\text{a}}{(np\backslash s)/(np\backslash s)}\,[Lex] \quad \dfrac{\dfrac{\text{créé}}{(np\backslash s)/np}\,[Lex] \quad \dfrac{\dfrac{\text{un}}{np/n}\,[Lex] \quad \dfrac{\text{journal}}{n}\,[Lex]}{np}\,[/E]}{np\backslash s}\,[/E]}{np\backslash s}\,[\backslash E]}{s}$$

FIGURE 2.2 – Dérivation en déduction naturelle de *"CSF a créé un journal"*.

créé un journal" peut être considérée comme une phrase. Pour éviter ceci, nous avons pris le parti de rajouter des types : ainsi, le participe passé est noté $np\backslash s_p$, ou, dans le cas de l'exemple figure 2.1, $(np\backslash s_p)/np$, de manière à ce qu'une telle phrase ne soit pas considérée comme correcte [2].

2.1.2 Calcul de Lambek

Lambek [1958] a repris les travaux de ses prédécesseurs sur les grammaires AB et a rajouté des règles d'introduction ainsi que de produit, permettant une plus grande souplesse dans l'analyse syntaxique des phrases. Même si Buszkowski [1996] montre que l'on peut traduire des grammaires de Lambek en grammaires AB (avec une variant du résultat de Pentus [1992]) en préservant la sémantique, ceci donne des grammaires AB beaucoup plus grandes que les grammaires de Lambek. On peut alors dire, comme le fait Oehrle [1994], que les grammaires de Lambek factorisent mieux l'extraction et la quantification.

Ses règles peuvent être représentées sous forme de règles de déduction naturelle :

$$\frac{B \quad B\backslash A}{A}\,[\backslash E] \qquad \frac{A/B \quad B}{A}\,[/E]$$

$$\frac{\begin{array}{c}[B]\\ \vdots \\ \mathrm{A}\end{array}}{B\backslash A}\,[\backslash I]_3 \qquad \frac{\begin{array}{c}[B]\\ \vdots \\ \mathrm{A}\end{array}}{A/B}\,[/I]_4$$

$$\frac{A \quad B}{A \bullet B}\,[\bullet I] \qquad \frac{A \bullet B \qquad \begin{array}{c}[A][B]\\ \vdots \\ \mathrm{C}\end{array}}{C}\,[\bullet E]_5$$

On peut aussi représenter les règles sous forme de calcul de séquent. Ce style de notation est appelé le style Gentzen [Gentzen, 1935] :

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B\backslash A}{\Gamma, \Delta \vdash A}\,[\backslash E] \qquad \frac{\Gamma \vdash A/B \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A}\,[/E]$$

$$\frac{B, \Gamma \vdash A}{\Gamma \vdash B\backslash A}\,[\backslash I] \qquad \frac{\Gamma, B \vdash A}{\Gamma \vdash A/B}\,[/I]_{\ 6}$$

2. Les catégories de base sont, usuellement, s, np, n auxquelles sont parfois rajoutées pp pour les prépositions. Nous expliquerons plus tard quelles catégories nous utilisons.

3. B est l'hypothèse libre la plus à gauche.

4. B est l'hypothèse libre la plus à droite

5. Il n'y a pas d'hypothèse libre entre A et B.

$$\frac{\Delta \vdash A \qquad \Gamma \vdash B}{\Gamma, \Delta \vdash A \bullet B}\ [\bullet I] \qquad \frac{\Delta \vdash A \bullet B \qquad \Gamma, A, B, \Sigma \vdash C}{\Gamma, \Delta, \Sigma \vdash C}\ [\bullet E]$$

$$\frac{}{A \vdash A}\ [lex]$$

Elles ont, par rapport aux grammaires AB, des caractéristiques supplémentaires :

Transitivité : $x \to y \wedge y \to z \Rightarrow x \to z$ (lorsqu'une expression de type x a aussi le type y et qu'une expression de type y a aussi le type z, alors une expression de type x a aussi le type z).

Monotonicité : $x/y \quad y/z \to x/z$ et $x\backslash y \quad y\backslash z \to x\backslash z$.

Associativité : $(x \bullet y) \bullet z \Leftrightarrow x \bullet (y \bullet z)$ et $(x\backslash y)/z \Leftrightarrow x\backslash(y/z)$

Les règles d'introduction permettent de traiter élégamment des cas tels que l'extraction ou la quantification. Par exemple, si on prend la proposition *"le livre que Marie a lu"*, on peut remarquer que le groupe verbal *"a lu"* se rapporte au livre. Si on veut que le type du verbe et son participe passé soient les mêmes que lorsque l'on écrit *"Marie a lu le livre"*, il faut introduire une catégorie *np*. La dérivation correspondante sera donc :

$$\dfrac{\dfrac{le}{np/n}\ [lex] \quad \dfrac{\dfrac{livre}{n}\ [lex] \qquad n\backslash n}{n}\ [\backslash E]}{np}\ [/E] \qquad \dfrac{\dfrac{que}{(n\backslash n)/(s/np)}\ [lex] \qquad \dfrac{s}{s/np}\ [/I]}{}\ [/E] \qquad \dfrac{\dfrac{Marie}{np}\ [lex] \qquad np\backslash s}{}\ [\backslash E] \qquad \dfrac{\dfrac{a}{(np\backslash s)/(np\backslash s_p)}\ [lex] \qquad np\backslash s_p}{}\ [/E] \qquad \dfrac{\dfrac{lu}{(np\backslash s_p)/np}\ [lex] \qquad [np]}{}\ [/E]$$

Cette structure peut être transformée en arbre, les règles d'introduction étant présentées comme des nœuds unaires, coindexés avec leurs hypothèses, comme on peut voir figure 2.3.

Au niveau des structures permettant de représenter les dérivations, l'associativité rend la représentation plus complexe et nécessite des graphes, étant donné qu'il faut un lien entre la règle d'introduction et son hypothèse. Le problème de coindéxation qui en découle est expliqué par Tiede [1998].

Les grammaires de Lambek peuvent aussi représenter le typage d'un λ-terme. Ainsi, le verbe *"manger"*, qui est généralement de catégorie $(np\backslash s)/np$, correspondra à un λ-terme de type $(e \to e \to t)$, pour dire qu'il a besoin de deux arguments. Bien que cette manière de gérer la sémantique soit basique, cela permet déjà d'en avoir une première approche.

6. Γ ne doit pas être vide

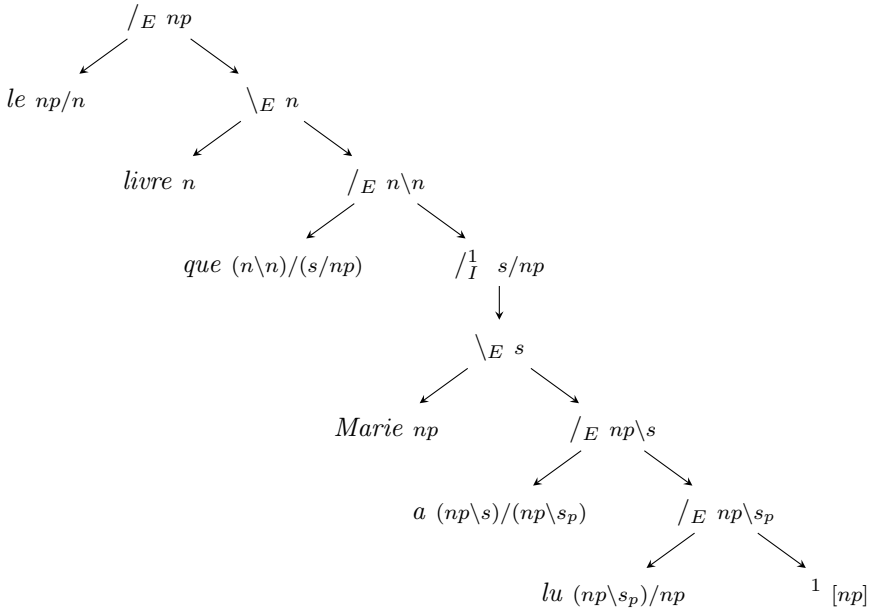


FIGURE 2.3 – Arbre de dérivation de *"Le livre que Marie a lu"*, correspondant à la dérivation précédente.

Les grammaires catégorielles permettent de formaliser efficacement la syntaxe d'une langue, et d'en ébaucher la sémantique. Si les grammaires AB proposent exclusivement des applications, correspondant aux règles d'élimination, les grammaires de Lambek quant à elles ont aussi des abstractions, grâce aux règles d'introduction. Ce sont des grammaires lexicalisées, hors-contexte [Pentus, 1997], dont le point de départ est un ensemble de primitives restreintes. Leur apprentissage, comme nous le verrons dans la section suivante, est complexe et c'est pour cela que nous avons mis au point un outil simplifiant cette étape (voir chapitre 3) et donnant à l'utilisateur le plein contrôle sur l'apprentissage par le biais de règles de transduction d'arbres.

Des extensions ont été créées à partir des grammaires de Lambek.

Grammaires Catégorielles Multimodales Ce sont des extensions des grammaires de Lambek (voir [Moortgat, 1997; Moot et Puite, 2002; Moortgat et Morrill, 1991; Moortgat et Oehrle, 1994]). Le nom "multimodal" vient du fait que les opérateurs / et \ deviennent $/_i$ et $\backslash_i$. De plus, certaines règles structurelles ne peuvent s'appliquer que dans des cas choisis à l'avance par l'utilisateur, ce qui donne un contrôle lexical aux règles.

De plus deux constructeurs unaires, $\Diamond$ et $\Box$ sont ajoutés. Ils donnent eux aussi un contrôle lexical sur les règles structurelles, et ajoutent aussi une inférence logique supplémentaire entre les formules. Les contraintes peuvent être

des contraintes de dérivations [7] mais aussi des contraintes sur la portée des quantificateurs [Bernardi et Moot, 2003].

L'exemple précédent, *"Le livre que Marie a lu"*, précisera que le groupe nominal *"Le livre"* a comme type $\Diamond_0\Box_0^{\downarrow}np$ pour dire qu'il a été extrait et le participe passé *"lu"*, quant à lui, aura un marqueur signifiant que son argument peut-être extrait.

Grammaires Catégorielles Combinatoires Elles ont été introduites par Steedman [2001] et utilisent des nouveaux types de règles [8] :

– Le *lifting*, noté T :

$$\frac{A}{B(A\backslash B)}\ [>_T] \qquad \frac{A}{(B/A)\backslash B}\ [<_T]$$

– La *composition*, notée B :

$$\frac{A/B \quad B/C}{A/C}\ [>_B] \qquad \frac{C\backslash B \quad B\backslash A}{C\backslash A}\ [<_B]$$

– La *mixed composition*, notée B_x :

$$\frac{A/B \quad C\backslash B}{C\backslash A}\ [>_{B_x}] \qquad \frac{B/C \quad B\backslash A}{A/C}\ [<_{B_x}]$$

Les règles de *lifting* et de *composition*, permettent de faire des dérivations valides en calcul de Lambek. Lorsque l'on a des dérivations utilisant la *mixed composition*, le langage généré devient faiblement contextuel et d'une puissance expressive plus importante que ce que peut générer une grammaire AB ou une grammaire de Lambek (des langages algébriques).

La figure 2.4 montre la dérivation du groupe nominal *"le livre que Marie a lu"* en CCG.

De plus, on note que les CCG modernes sont aussi multimodales, ce qui leur donnent plus de souplesse [Baldridge, 2002].

Grammaires Catégorielles Abstraites Ces grammaires (voir [de Groote, 2001]) ont deux alphabets liés en entrée : un alphabet abstrait et un alphabet d'objets. Les deux seront liés par un lexique et l'alphabet abstrait pourra être transformé en λ-terme et représente la structure syntaxique abstraite, alors que l'alphabet d'objet représente la forme de surface (la phrase). Il faut noter cependant que ces grammaires n'ont pas de traitement pour la coordination aussi esthétique que celui effectué par les grammaires AB ou les grammaires de Lambek [Muskens, 2001; Kubota et Levine, 2013] ; à ce niveau là, elles semblent donc moins adaptées pour la sémantique.

7. Comme les *"islands constraints"* de Kurtonina et Moortgat.

8. Ce ne sont pas les seuls types de combinateurs. Il y a aussi la substitution, notée S, et des versions plus généralisées de la composition.

2.2 Apprentissage

C'est en 1967 que Gold [1967] définit une notion clef : celle de l'identification à la limite. Fondée sur l'exemple d'un enfant apprenant sa langue maternelle, cette méthode part du principe que l'apprenant reçoit une suite infinie d'exemples positifs et construit petit à petit la grammaire correspondante. La suite d'exemple doit avoir les caractéristiques suivantes (voir [Béchet *et al.*, 2007]) :

- Ne contenir que des exemples positifs : des phrases correctement formées et ayant un sens.
- Ne contenir aucun contre-exemple. Gold est parti du principe que les enfants n'apprenaient pas par contre-exemple (correction d'une faute récurrente) mais par mimétisme par rapport aux adultes qui pratiquent correctement la langue.
- Contenir tout ce qui peut apparaître dans le langage.
- Les exemples peuvent apparaître dans n'importe quel ordre sans que cela influe sur l'apprentissage de la langue.
- Les exemples peuvent apparaître plusieurs fois (cela permet de modéliser une suite infinie avec un ensemble d'exemples fini).

L'idée est que, même si la suite d'exemples reçus est infinie, la langue apprise va, à partir d'un temps fini, être capable de reconnaître comme lui appartenant toutes les nouvelles phrases. A chaque exemple positif reçu, il y a deux possibilités :

- La grammaire apprise permet de reconnaître la phrase, rien n'est fait.
- La grammaire ne permet pas de reconnaître la phrase, de nouvelles règles permettant de reconnaître celles-ci sont ajoutées.

Il y a eu plusieurs études du modèle de Gold (voir [Bonato et Retoré, 2001; Retoré et Bonato, 2013; Moreau, 2006; Clark et Lappin, 2010]) et Moreau [2007] précise que cette identification à la limite peut s'appliquer à des classes de langage, de grammaires, mais non à une seule grammaire précise. Tout

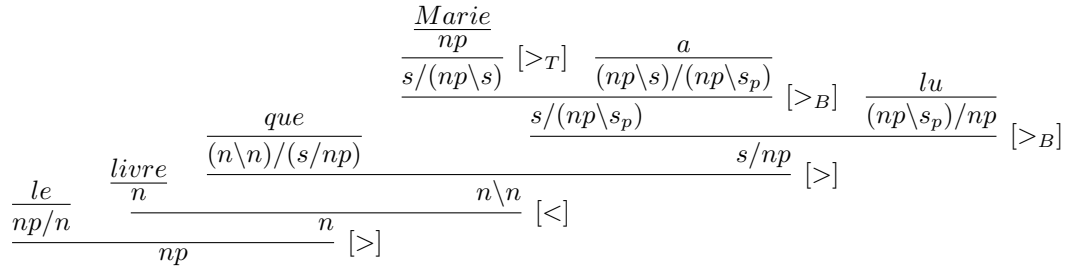


FIGURE 2.4 – Dérivation en CCG de *"le livre que Marie a lu"*. On remarque que, pour indiquer si c'est une élimination à droite ou à gauche, la notation sera $<$ ou $>$.

d'abord, il définit un triplet $\langle \mathcal{U}, \mathcal{G}, \mathcal{M} \rangle$, appelé système de grammaires, où $\mathcal{U}$ et l'ensemble d'objets (pour Univers) $\mathcal{G}$ un ensemble de grammaires et $\mathcal{M}$ une fonction qui pour chaque grammaire lui associe un sous-ensemble dans $\mathcal{U}$.

Définition 1. *(Moreau [2007]) Soit $\langle \mathcal{U}, \mathcal{G}, \mathcal{M} \rangle$ un système de grammaires, ψ une fonction d'apprentissage et $L \subseteq \mathcal{U}$ un langage. Soit $\langle a_0, a_1, \ldots \rangle$ une séquence infinie d'objets de $\mathcal{U}$, telle que $a \in \{a_i \mid i \in \mathbb{N}\}$ si et seulement si $a \in L$.*

ψ converge vers G s'il existe $n \in \mathbb{N}$ tel que pour tout $i \geq n$ $G_i = \psi(\langle a_0, a_1, \ldots \rangle)$ est définie et $G_1 = G$.

ψ apprend un langage L si, pour toute énumeration de L, ψ converge vers une grammaire G telle que $\mathcal{M}(G) = L$.

Une classe de langages $\mathcal{L} \subseteq P(\mathcal{U})$ est dite apprenable s'il existe une fonction d'apprentissage ψ telle que ψ apprend L pour tout langage $L \in \mathcal{L}$.

Si initialement ce modèle n'était pas très efficace, Angluin [1980] a donné un nouvelle essor à celui-ci et plusieurs méthodes d'apprentissage sont à présent des références dans le domaine de l'apprentissage de grammaires.

Une longue étude du modèle de Gold et des différents algorithmes d'apprentissage a été effectuée dans la thèse de Moreau [2006]. De même, Costa-Florêncio [2003] effectue une étude détaillée des différentes classes de grammaires de Lambek et leur apprenabilité. Clark et Lappin [2010] quant à eux considèrent le problème de l'apprenabilité des grammaires de Lambek comme un problème d'apprentissage automatique.

Les méthodes d'apprentissage sont diverses, comme nous avons pu le voir dans l'introduction. Nous allons ici nous focaliser sur les algorithmes d'apprentissage de grammaires AB. Il existe de nombreux algorithmes d'apprentissage, et nous nous focaliserons ici sur deux piliers d'apprentissage de grammaires AB, les algorithmes de Buszkowski et Penn [1990] (voir aussi [Buszkowski, 1987]) et Kanazawa [1998].

Le premier apprend une grammaire AB rigide, c'est à dire dont le nombre de type par mots est limité à un, et le second une grammaire k-valuée, c'est à dire où chaque mot du lexique a le droit d'avoir jusqu'à k catégories associées. Le point commun entre ces deux algorithmes, outre le fait qu'ils apprennent des grammaires AB, est le format de données pris en entrée : des FA-structures, avec FA pour *Foncteur-Argument*. Grâce à Bechet et Foret [2003b], nous savons qu'aussi bien les grammaires de Lambek rigides que les grammaires k-valuées sont apprenables à partir de FA-structures [9], par conséquent les grammaires AB le sont aussi. De plus, une étude détaillée de l'apprenabilité des grammaires rigides a été effectuée par Bonato [2006].

9. On ne peut cependant pas apprendre de grammaires de Lambek à partir de chaînes [Bechet et Foret, 2003a; Foret et Le Nir, 2002].

La figure 2.5 montre la FA-structure correspondant à la phrase *"CSF a créé un journal"*. Dans ce cas, les seules informations que nous avons sur l'arbre sont que la racine est étiquetée s et quelle règle nous allons utiliser en suivant.

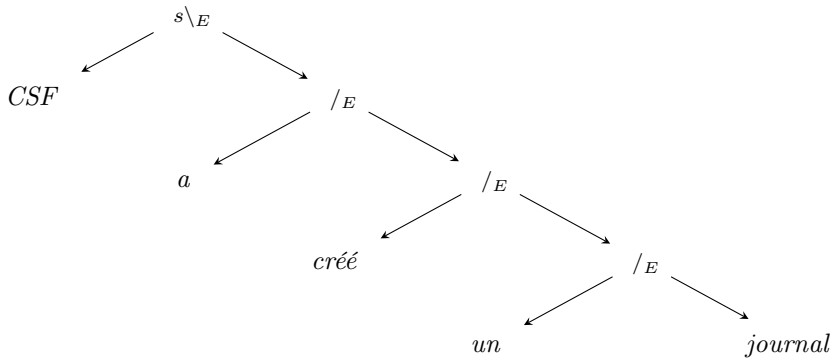


FIGURE 2.5 – FA-structure de *"CSF a créé un journal"*. Seules sont précisées les règles utilisées, C'est à dire l'élimination à droite ou à gauche dans le cas présent.

Ces structures suffisent à déduire les types de tous les mots, étant donné que pour chaque niveau, en fonction de la catégorie x du père, le fils foncteur obtiendra la catégorie x/y ou $y\backslash x$ et le fils argument y, en fonction de si c'est une règle d'élimination à droite ou à gauche.

Dans la suite du document, nous utiliserons des arbres où les variables sont précisées à chaque niveau, de manière à ce que ce soit plus lisible, ce qui donne beaucoup d'informations superflues dans la présentation mais qui permet de faciliter la lecture des arbres.

2.2.1 Apprentissage d'une grammaire rigide

L'algorithme de Buszkowski et Penn permet d'apprendre une grammaire rigide à partir d'un ensemble d'arbres. On peut résumer la méthode par l'algorithme 1.

Cependant, les grammaires rigides ne permettent pas de représenter correctement une langue naturelle. Pour illustrer le problème de l'unification, il suffit de prendre deux phrases du corpus de Paris VII : *"Nous avons de plus en plus de monde dans les DOM-TOM"* et *"Le gouvernement n'avait ni écrit ni choisi cet accord dont nous avons hérité"*, dont les arbres (des FA-structures enrichies des variables sur chaque nœud) sont représentés figure 2.6. Le lexique avant unification est décrit dans la table 2.1.

En appliquant l'algorithme de Buszkowski et Penn, le verbe *avons* aura deux types qui se ressemblent : $(e\backslash b)/f$ et $(u\backslash t)/v$. En effet, à chaque fois *avons*

Données : arbres dont la racine est étiquetée s, les nœuds internes / ou \, et les feuilles avec des variables
Résultat : une grammaire rigide
création d'un lexique de mots contenant toutes les variables qui leur sont liées ;
tant que *chaque mot a plusieurs types qui lui sont liés* **faire**
  rechercher l'unificateur le plus généraliste, de manière à réduire globalement le nombre de variables liées aux mots;
  si *l'unification n'est pas possible* **alors** l'algorithme échoue;
fin
Algorithme 1: Algorithme d'apprentissage d'une grammaire rigide.

prend deux arguments, l'un à sa gauche et l'autre à sa droite. Cependant, nous ne pouvons pas unifier ces deux types, parce que dans le premier cas l'argument de droite est un groupe nominal et dans le second cas un participe passé ; en outre nous ne souhaitons pas que les deux aient le même type, pour éviter des phrases agrammaticales. Pour ces deux phrases, nous avons donc besoin au minimum d'une grammaire 2-valuée, et *avons* doit avoir deux types différents : $(np\backslash s)/np$ et $(np\backslash s)/(np\backslash s_p)$.

nous	e, u	avons	$(e\ \backslash b)/f$ $(u\ \backslash t)/v$	de_plus_en_plus	f/g
de	g/h	monde	h	dans	$(b\backslash a)/c$
les	c/d	DOM-TOM	d	le	j/k
gouvernement	k	n'	$((j\ \backslash i)/l)/m$	avait	n/o
ni	o/p, $n\backslash m$	écrit	p	choisi	l/q
cet	q/r	accord	s	dont	$(s\backslash r)/t$
hérité	v	.	$a\backslash s$, $i\backslash s$		

TABLE 2.1 – Lexique avant unification

De même, on voit que si l'on essaye d'avoir une grammaire rigide qui couvre *"Jean et Marie"* ainsi que *"manger et dormir"*, soit les verbes auront le même type que les noms propres, soit l'unification est impossible.

Les grammaires rigides présentent l'avantage de donner un résultat à coup sûr, alors que, comme nous verrons dans la section suivante, les grammaires k-valuées : soit les types du lexique peuvent être unifiés jusqu'à ce qu'il n'y en ait plus qu'un par mot, soit l'algorithme échoue. Il faut cependant vérifier que le lexique obtenu ne présente pas d'incohérence, comme unifier deux variables qui, en réalité, n'ont rien à voir.

Il est à noter aussi que les grammaires rigides ne sont pas apprenables à partir de chaînes [Foret et Le Nir, 2002]. Il faut donc nécessairement une entrée sous forme d'arbre.

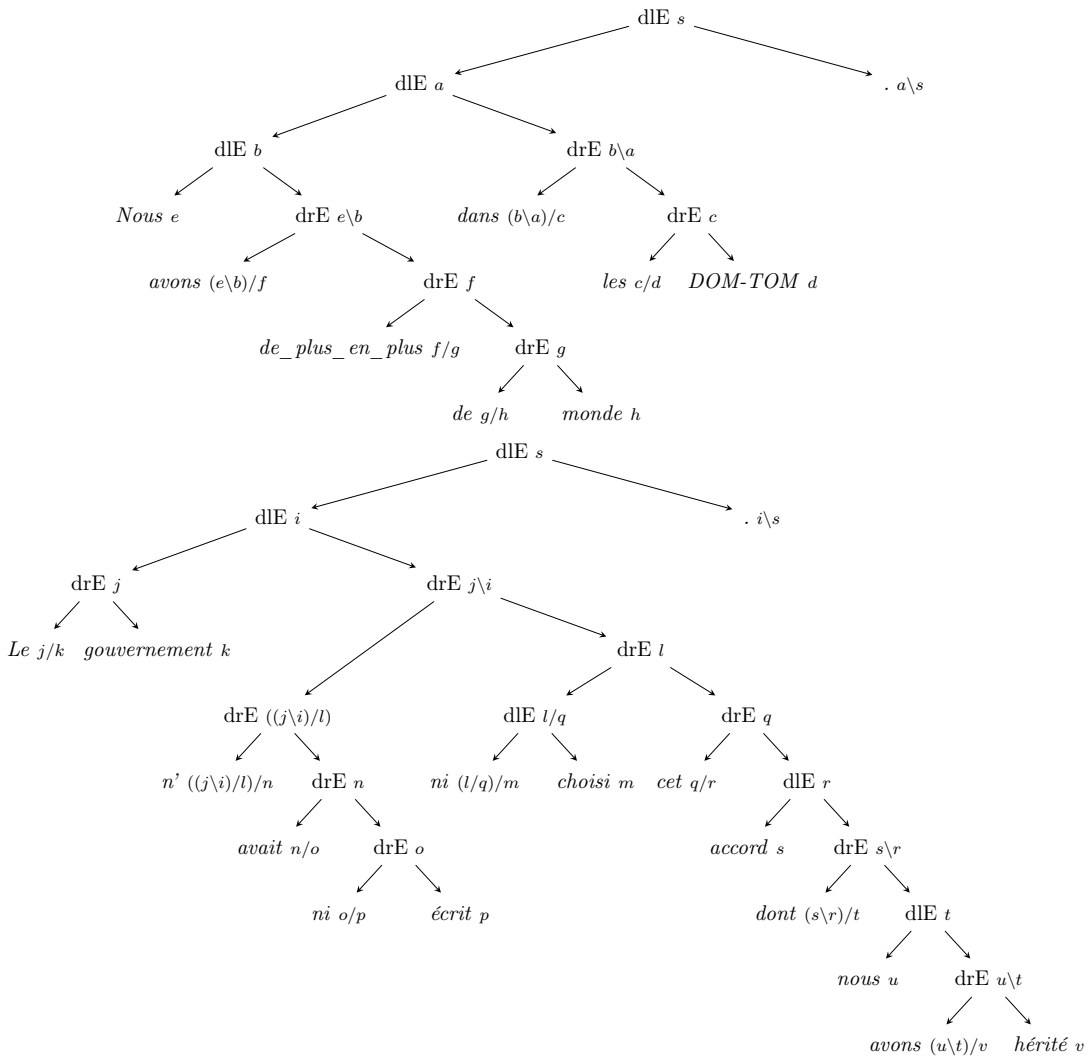


FIGURE 2.6 – Entrées pour notre exemple d'unification, équivalentes à des FA-structures, mais les variables ont été rajoutées sur chaque nœud.

2.2.2 Apprentissage d'une grammaire k-valuée

Pour apprendre une grammaire k-valuée [Kanazawa, 1998] il y a besoin du même format d'entrée, comme montre la figure 2.6. Le cœur du problème avec les grammaires k-valuées est de décider quels types doivent être unifiés, car la meilleure unification possible ne peut être décidée que d'un point de vue global. C'est pour cela que ce problème d'inférence grammaticale est NP-dur [Costa-Florêncio, 2001] dès que $k \geq 2$. Il est également important de noter que k n'est généralement pas connu d'avance, ce qui complique la résolution de l'algorithme.

La solution usuelle, pour trouver k, est d'utiliser une recherche dichotomique de celui-ci : en partant d'un k_1 qui paraît satisfaisant [10] on essaye de réaliser une première unification. Ensuite, deux cas de figure s'offrent à nous :

- L'unification réussit avec k_1. Celui-ci devient une "borne supérieure" et on tente avec $k_1/2 + 1$.
- L'unification échoue avec k_1. Il devient une "borne inférieure" et on tente avec $k_1 \times 2$.

Une fois que l'on a trouvé une borne inférieure et une borne supérieure, on va s'approcher du bon résultat en testant à chaque fois le k_n moyen entre les deux bornes. Lorsque l'on essaye de passer à k_i types alors qu'un mot en a $k_i + j$, cela nécessite d'essayer d'unifier j types parmi $k_i + j$. Cela demande donc de faire : $\frac{(k_i+j)!}{j!\times k_i!}$ tentatives, et ce pour chaque mot dont le nombre de types dépasse k_i.

Il faut cependant préciser que certains mots très utilisés ont très peu de catégories, comme *"ce"*, alors que *"et"* est obligé d'avoir de nombreuses formules. Il est donc dommageable de s'arrêter de factoriser les types lorsque *"et"* n'est plus factorisable alors que *"ce"* peut encore réduire la taille de son lexique.

Plusieurs autres chercheurs se sont penchés, comme nous, sur l'apprentissage de grammaires catégorielles. Retoré et Bonato [2013] ont mis au point un algorithme qui apprend une grammaire de Lambek rigide, utilisant la règle du produit. Les structures de départ sont des réseaux de preuves, qui sont une évolution des FA-structures auxquelles on a adjoint les informations nécessaires pour les règles d'introduction et le produit. Ce travail peut être vu comme une extension totalement compatible avec leurs travaux précédents, [Bonato et Retoré, 2001], où la grammaire apprise ne prenait pas en compte le produit. L'apprenabilité du calcul de Lambek associatif a été étudiée par Foret [2001].

Nous avons voulu nous approcher de ces méthodes d'unification tout en améliorant les quelques inconvénients, surtout au niveau de la complexité des algorithmes. La méthode utilise pour cela une étape préliminaire de clustering hiérarchique, plus de détails sont donnés dans le chapitre 4.

2.3 Corpus

Tout notre travail repose sur l'utilisation de corpus. Nous avons principalement utilisé le corpus de Paris VII [Abeillé *et al.*, 2003], et plus récemment le corpus Séquoia [Candito et Seddah, 2012]. Les deux corpus sont disponibles sous deux formes :

- Un format parenthésé, lisible avec Stanford Tregex [Levy et Andrew, 2006a], où les informations sont simplifiées. On a dans ce cas uniquement

10. Pour une langue naturelle, il a été montré qu'un k de 180 était satisfaisant Moot [2010a]; Hockenmaier [2006] pour des grammaires.

le type lexical du mot et les types ainsi que le rôle dans la phrase des syntagmes représentés par les sous-arbres ;
– un format `XML` qui contient en plus des informations de nombre et genre sur les mots, ainsi que le type temps et la personne des verbes.

Nous avons pris le parti d'utiliser les premiers. En effet, pour extraire une grammaire AB les informations contenues dans les versions parenthésées des corpus sont largement suffisantes, car c'est surtout l'attachement des syntagmes dans la phrase qui va jouer. Un exemple de phrase est montré figure 2.7.

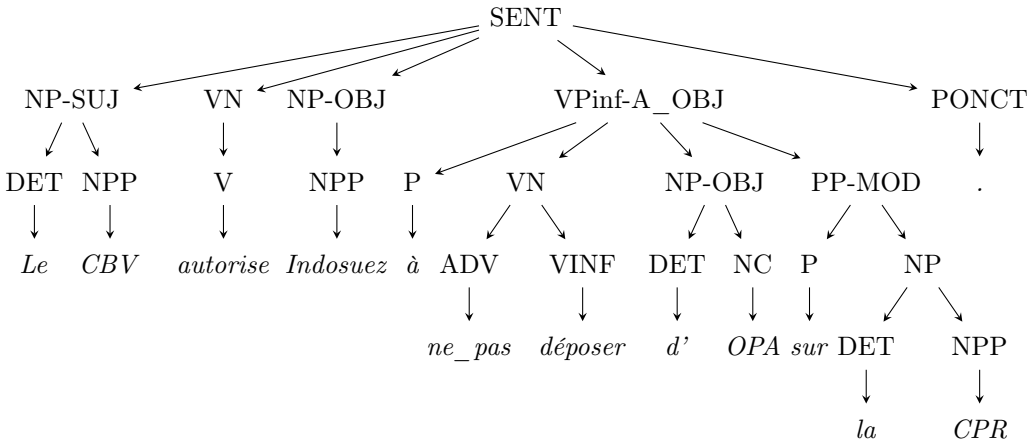


FIGURE 2.7 – Exemple d'arbre syntaxique du corpus de Paris VII.

2.3.1 Annotations

Les annotations des corpus sont de deux types : les POS-tags [11] qui donnent le type lexical du mot (ce sont uniquement les nœuds pré-terminaux des arbres) ainsi que les annotations morpho-syntaxiques qui donnent le type du syntagme du sous-arbre ainsi que, parfois, son rôle dans la phrase. Le sens de chaque étiquette est expliqué en suivant, et pour plus de détails on peut se référer à [Abeillé et Clément, 2003].

11. *Part-Of-Speech*, autrement appelés *Parties du discours*.

POS-tags

Les POS-tags sont utilisés uniquement sur les nœuds pré-terminaux des arbres.

Etiquette	**signification**	**exemple**	**remarque**
ADJ	Adjectifs	premier	
ADJWH	Adjectifs interrogatif	quel	
ADV	Adverbes	longtemps	
ADVWH	Adverbes interrogatifs	pourquoi	Itilisé en début de phrase interrogative.
CLO	Clitique objet	en	
CLR	Clitique réflexif	s'	
CLS	Clitique Sujet	-il	
CC	Conjonction de coordination	et	
CS	Conjonction de subordination	que	
DET	Déterminants	le	
DETWH	Déterminants interrogatifs	Quelle	
ET	Mot étranger	investment_banks	
I	interjection	hélas	
NC	Noms communs	ministre	
NPP	Noms propres	Charasse	
P	Prépositions	par	
P+D	Prépositions et déterminants	au	
PRO	Pronoms	celui-ci	
PROREL	Pronoms relatifs	laquelle	
PROWH	Pronoms interrogatifs	qui	Utilisé pour des phrases interrogatives telles que *"Qui [...] ?"*
PONCT	Ponctuations	:	
V	Verbes conjugués	est	
VIMP	Verbes à l'impératif	organisons	

Etiquette	signification	exemple	remarque
VINF	Verbes à l'infinitif	étendre	
VPP	Participes passés	liée	
VPR	Participes présents	resserrant	

Les clitiques peuvent, en plus, avoir une précision sur leur rôle dans la phrase : sujet, objet ou modificateur.

Annotations morpho-syntaxiques

Elles donnent le rôle dans la phrase du syntagme correspondant à leur sous-arbre.

Etiquette	Signification	Exemple
*AdP**	Syntagme adverbial	*"très peu"*
*AP**	Syntagme adjectival	*"acquis ou potentiel"*
COORD	Coordination	*"et la commission financière"*
*NP**	Syntagme nominal	*"la péninsule ibérique"*
*PP**	Syntagme prépositionnel	*"par la bonne odeur du pain"*
SENT	Phrase	*"A cette époque, on avait dénombré 140 candidats."*
*Sint**	Proposition conjuguée interne	*"rien ne l'y obligerait"*
*Srel**	Proposition relative	*"dont trente-deux sont des femmes"*
*Ssub**	Proposition subordonnée	*"qu'il touche l'ensemble du secteur"*
VN	Noyau verbal	*"se dessinait"*
*VPinf**	Proposition infinitive	*"sans oublier les compagnies pétrolières"*
*VPpart**	Proposition participiale	*"posée par la crise de la cinq"*

Aux étiquettes ayant une astérisque en exposant, on peut ajouter les informations suivantes :

Etiquette	signification	Etiquette	signification
A_OBJ	Objet précédé de la préposition *à*	*ATO*	Attribut d'objet
DE_OBJ	Objet précédé de la préposition *de*	*ATS*	Attribut du sujet
P_OBJ	Objet précédé d'une préposition	*OBJ*	Objet
SUJ	Sujet	*MOD*	Modificateur

2.3.2 Les différents corpus

Corpus de Paris VII : La version parenthésée que nous utilisons date de 2004, et contient 12 351 phrases tirées d'articles du journal *Le Monde*, entre décembre 1989 et janvier 1994 [Abeillé *et al.*, 2003]. Nous avons retiré, pour notre ensemble-test, trois phrases qui ne nous semblaient pas pertinentes (telles que *"Mais PO"*) pour notre travail. Il existe une version en `XML` qui contient 504 phrases supplémentaires. Nous avons effectué une extraction et transformation manuelle de ces phrases supplémentaires pour créer un "corpus d'évaluation", c'est à dire un corpus qui vient du même endroit que le corpus principal que nous utilisons pour nos travaux d'apprentissage (voir chapitre 3) mais que nous n'avons pas utilisé pour élaborer nos méthodes. Il a donc représenté, jusqu'à l'apparition du corpus Séquoia, notre seul moyen de tester nos algorithmes sur un ensemble de données non utilisé pour les créer.

Séquoia : Plus récent (2012), ce corpus contient actuellement 3 200 phrases venant de diverses origines : Wikipédia, l'Est Républicain et des notices de médicaments [Candito et Seddah, 2012]. Les phrases offrent une variété plus grande dans le style rédactionnel mais sont généralement moins complexes que le corpus de Paris VII. Elles sont cependant annotées et présentées de la même manière que le corpus précédent, c'est à dire sous forme d'arbres syntaxiques, disponibles en format parenthésé ou `XML`.

Est Républicain : dans un registre totalement différent, nous avons utilisé le corpus de l'Est Républicain [Gaiffe et Nehbi, 2009] comme réservoir de phrases accessibles et utilisables. Ce corpus n'est pas annoté sous forme d'arbres syntaxiques mais découpé en articles et en phrases, présenté sous forme `XML`. Nous en avons extrait 520 phrases dans le but de les analyser (voir chapitre 5). Notre choix s'est porté sur les articles politiques qui nous semblaient dans le même style que ceux de Paris VII, qui n'apparaissaient cependant pas dans Séquoia.

Chapitre 3

Transducteur pour inférence grammaticale

Les transducteurs sont des outils mathématiques décrits en détail dans [Comon *et al.*, 2007], dont les propriétés formelles sont bien connues.

Les applications possibles pour les transducteurs sont nombreuses et bien connues : en informatique, on les utilise pour travailler les fichiers `XML` ou `XSSLT`, et leur utilité en compilation a été longuement expliquée par Fülöp et Vogler [1998].

Une des motivations du développement de l'utilisation des transducteurs dans le domaine du traitement automatique des langues est de modéliser une partie importante de la théorie transformationnelle de Chomsky [1965], comme on peut voir dans l'article de Rounds [1970]. Les utilisations des transducteurs se sont ensuite multipliées : Knight et Graehl [2005] présentent un survol orienté vers le traitement statistique des langues naturelles avec des transducteurs ; l'article de Maletti *et al.* [2009], se focalise sur les transducteurs d'arbres descendants étendus, donnant en exemple d'utilisation de la transformation des arbres syntaxiques anglais en arbres syntaxiques arabes. Kaplan et Kay [1994], quant à eux, s'en servent pour expliciter les règles phonologiques d'une langue naturelle.

La définition élégante d'un transducteur ainsi que sa puissance expressive en faisait, pour nous, l'outil idéal pour transformer les arbres syntaxiques des corpus en arbres de dérivation. Plus que binariser les arbres, ce qu'un algorithme de mise en forme normale de Chomsky ou de Greibach [Autebert *et al.*, 1984] peut faire sans problème, nous avons voulu écrire des règles de manière à apprendre en même temps une grammaire AB sans passer par un algorithme d'apprentissage tel que ceux de Buszkowski et Penn [1990] ou Kanazawa [1998].

En effet, d'un point de vue pratique, force nous a été de constater que :

– convertir simplement les arbres des corpus en formats d'entrée demandés pour les algorithmes d'apprentissage présentés en section 2.2 est aussi

complexe que de générer directement des arbres de dérivation [1]. Les annotations linguistiques présentes dans les corpus donnent généralement assez d'informations pour savoir, entre deux nœuds, qui sera foncteur et qui sera argument ainsi que l'instantiation des types [2].
– quand le lexique représentatif de la grammaire permet d'associer plus d'un type à chaque mot, la complexité de l'apprentissage devient NP-dure [Costa-Florêncio, 2001].

Il était donc plus intéressant de générer une grammaire AB tout en restant compatible avec les algorithmes d'apprentissages vus dans le chapitre 2.

Les travaux appliqués à des corpus qui apprennent eux aussi des grammaires catégorielles, telles que [Hockenmaier, 2003, 2006; Moot, 2010a,b], utilisent des algorithmes spécifiques aux corpus sur lesquels ils ont été bâtis, avec peu d'espoir de pouvoir les adapter aux corpus français : la complexité de la tâche équivaudrait à créer un nouvel algorithme. De plus, il est toujours délicat de vérifier qu'un tel algorithme satisfait les propriétés de la grammaire que nous recherchons avec une autre méthode que la vérification systématique de la sortie.

C'est aussi en cela que nous avons voulu différencier notre transducteur des autres méthodes. Cela sera expliqué plus en détail dans la section 3.3 : le cœur du transducteur a été implémenté en respectant la définition formelle que nous donnons en suivant, mais les règles de transduction en elles-même sont à part. Chacun peut donc, en théorie, implémenter facilement son propre ensemble de règles et l'ajuster à son ensemble de données, quelque soit la langue du corpus qu'il veut transformer.

Ces travaux ont fait l'objet de deux publications [Sandillon-Rezer et Moot, 2011; Sandillon-Rezer, 2011].

Dans un premier temps, nous donnerons une définition formelle de notre transducteur, bien que celle-ci soit, à part au niveau de la description des règles, identique à celle d'un transducteur d'arbre descendant. Nous nous focaliserons ensuite sur la description des règles de transduction ainsi que l'implémentation que nous avons faite du G-transducteur. Nous terminerons par une évaluation de notre outil.

1. Kanazawa [1998] propose des algorithmes pour apprendre des grammaire catégorielles à partir de phrases sans structure. Cependant, leur complexité particulièrement élevée empêche de les utiliser dans un cas pratique.

2. Un travail en ce sens avait été entamé par Foret *et al.* [2006], mais est resté à l'étape d'ébauche.

3.1 Définition formelle

Notre transducteur est une extension conservative d'un transducteur descendant usuel comme vu dans [Comon *et al.*, 2007]. Nous avons facilité l'expression des règles en gardant, cependant, les propriétés clefs des transducteurs d'arbres standards. Il est donc bon de rappeler et de noter les différences ainsi que les points communs entre les deux.

Définition 2. *Un transducteur généralisé est un tuple* $\langle Q, q_i, Q_f, X, \delta \rangle$ *où :*

- Q *est l'ensemble des états.*
- q_i *est l'état initial, avec* $q_i \in Q$.
- Q_f *est l'ensemble des états terminaux, avec* $Q_f \subseteq Q$.
- X *est l'alphabet de lecture et d'écriture.* $X = \{A \cup M \cup T\}$ *avec* A *l'aphabet d'annotation du corpus (SENT, NP, VN, etc.),* M *les mots français et* T *pour les types (np/n, etc.).*
- δ *est l'ensemble des règles de transduction, de la forme :*
  $q(f(x_1, \ldots, x_n)) \mapsto u[q_1(t_1), \ldots q_p(t_p)]$ *où pour chaque* $i \in [1, p]$, *soit* $t_i \in \{x_1, \ldots, x_n\}$ *soit* $t_i = f_i(x_{i,1}, \ldots x_{i,m})$ *avec* $x_{i,j} \in \{x_1, \ldots, x_n\}$ *;* $m \leq n-1$. *La fonction* $f \in A$ *est d'arité* n *;* $q, q_1, \ldots, q_n \in Q$ *et* $u \in T(r, X_n)$ *(l'ensemble des sous arbres ayant pour racine* $r \in A$ *et avec comme feuilles* X_n, *où pour chaque* $x_i \in X_n$, *on a* $x \in A$*).*
  Dans notre cas, nous utilisons une sous-classe de ces règles :

  $$q(f(x_1, \ldots, x_n)) \mapsto u[q_1(t_1), \ldots q_p(t_p)] \text{ où pour tout } i \in [1, p],$$

  $$t_i \in \{x_1, \ldots x_n\} \text{ ou } \begin{cases} q_i = q \text{ et} \\ t_i = f(x_{i,1}, \ldots x_{i,m}) \end{cases}$$

  avec $(x_{i,1}, \ldots x_{i,m})$ *une sous-séquence quelconque de* $(x_1, \ldots, x_n)$. *Nous appelons ces règles des "règles récursives", étant donné que* $q(f(\ldots))$ *appelle* $q(f(\ldots))$.

3.1.1 Propriétés d'un transducteur

Pour garder l'intégrité des phrases après transduction, nous nous sommes assurés que le G-transducteur garde certaines propriétés des transducteurs usuels.

Temps réel (ε-free) : il n'y a pas d'ε-transduction.

Linéaire : il n'y a pas de variable qui apparaisse deux fois dans la partie droite des règles.

Non-effaçant : au moins un symbole de A apparaît dans la partie droite des règles.

Complet : pour chaque règle de type $f(q_1(x_1), \ldots, q_n(x_n)) \mapsto q(u)$ (pour les transducteurs ascendants) ou $q(f(x_1, \ldots, x_n)) \mapsto u[q_1(x_{i_1}), \ldots q_p(x_{i_p})]$ (pour les transducteurs descendants) toutes les variables x_i apparaissent au moins une fois dans u.

Déterministe s'il n'y a pas d'ε-règle et qu'il n'existe pas deux règles avec la même partie gauche.

***Look-ahead* fini :** nous autorisons chaque règle de transduction à avoir un arbre complexe dans sa partie gauche. C'est à dire qu'aussi bien f et les f_i peuvent avoir un arbre complexe avec la frontière indiquée. Cela correspond au fait d'avoir un *look-ahead* fini. [3]

3.1.2 Exemple d'utilisation d'un transducteur

Nous pouvons utiliser un transducteur, par exemple, pour passer une phrase de l'actif au passif. Ainsi, la phrase *"le gouvernement souligne l'évolution"* doit devenir *"l'évolution est soulignée par le gouvernement"*. Nous avons besoin du *look-ahead* fini pour déplacer les sous-arbres tout en sachant qu'il faudra les modifier ensuite. Tout d'abord, le sujet et l'objet doivent être inversés. Ensuite, le verbe doit devenir passif : on introduit l'auxiliaire "être" et le verbe devient un participe passé. Enfin, l'ancien sujet est intégré dans un syntagme prépositionnel objet commençant par "par". La figure 3.1 montre l'arbre syntaxique avant et après la transduction.

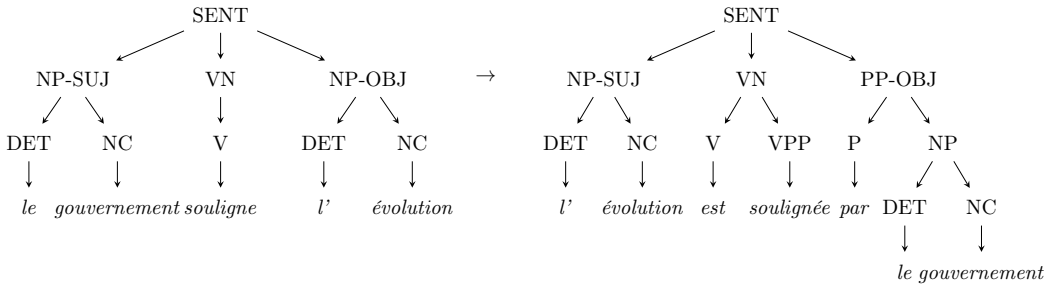


FIGURE 3.1 – Passage de l'actif au passif : l'objet est devenu sujet sans modification, le verbe devient passif et le sujet est intégré à un syntagme prépositionnel.

La règle de transduction correspondante, écrite comme vu dans TATA, sera :

3. Dans la terminologie utilisée par [Knight et Graehl, 2005], nos transducteurs sont des extensions des transducteurs *xRLND*, où le préfixe x signifie que nous autorisons nos règles à avoir un arbre au lieu d'un nœud dans leur partie gauche, R indique *Root to frontier*, c'est-à-dire descendant, L est pour linéaire, N pour Non-effaçant et D pour déterministe.

```
SENT(NP-SUJ(X) VN(V(Y)) NP-OBJ(Z))↦
SENT(NP-SUJ(Z), VN(V(est), VPP(Y+é)) PP-OBJ(P(par) NP(X)))
```

où $Y+$ est l'inflection qui transforme Y en son participe passé. Il faut cependant noter que toutes les phrases ne peuvent pas utiliser cette technique pour être passées au passif. Il faut évidemment que le verbe soit transitif et prenne en argument un complément d'objet direct. On peut noter aussi :

- Dans le cas du pronom personnel indéfini *"on"*, celui-ci disparait complètement. Ainsi, la phrase *"On avait dénombré cent-quarante candidats"* devient *"Cent-quarante candidats avaient été dénombrés"* et l'ancien sujet disparait complètement.
- Dans le cas des verbes exprimant un sentiment, on préfèrera utiliser la préposition *"de"* plutôt que *"par"*.
- Certaines phrases sans agentivité ne peuvent pas être passées au passif, telles que *"Ils ont pesé 3kg"* ou *"Il a eu sa punition"* [Chomsky, 1955].

3.1.3 Particularités du G-transducteur

Ces particularités différencient le G-transducteur d'un simple transducteur descendant, ce qui permet une spécification plus compacte des règles.

Récursivité : Une règle récursive s'applique à un nœud avec une étiquette donnée mais un nombre arbitraire de fils. La figure 3.2 montre un exemple de règle récursive, où le motif reconnu correspond au fils le plus à droite uniquement (le motif reconnu peut être plus complexe et contenir plusieurs fois un nombre arbitraire de nœuds, comme nous expliquons section 3.2). Le nœud père P a $n+k$ fils. Les descendants de 1 à n peuvent correspondre à n'importe quelle séquence de nœuds, mais les nœuds de $n+1$ à $n+k$ doivent correspondre au motif demandé. En d'autres termes, le motif reconnu correspond à k termes avec au moins un nœud à sa gauche. On remarque que les figures 3.3 et 3.4 sont aussi des instances de ce schéma, avec respectivement $n = 1$ et $n = 2$. Comme montré dans la figure 3.2, le transducteur passe au nouveau nœud P d'arité n et "restructure" les nœuds $n + 1$ à $n + k$ en un sous-arbre T. Ce sous-arbre sera traité dans une étape de transduction postérieure.

Etant donné que ces règles récursives généralisent la définition standard d'une règle de transduction en autorisant les nœuds à avoir une arité arbitraire on peut, en connaissant l'arité maximale des arbres d'entrée, convertir ces règles en plusieurs règles "ordinaires" de transduction.

Paramétrisation : Nous autorisons les règles avec une quantification restreinte sur les étiquettes des nœuds. Un exemple est montré dans la figure 3.3. La variable X peut correspondre à trois étiquettes différentes. Cette propriété est équivalente à écrire trois fois la règle dans le cas cité

en exemple, mais cela permet lors de la création des règles d'être certain de gérer les cas équivalents de la même manière.

Priorité : Pour éviter le non-déterminisme de notre transducteur, nous avons pris le parti d'appliquer les règles toujours dans le même ordre, comme montré par la figure 3.4. Le seul inconvénient avec cette méthode est qu'elle donne toujours une visibilité plus importante aux mêmes nœuds. Il faut noter cependant que même si cela biaise la forme des arbres donnés en résultat, cela ne fait aucune différence au niveau des types donnés aux mots du lexique. De plus, il semble complexe d'améliorer le transducteur de manière à ce qu'il puisse déduire automatiquement et correctement quelle règle appliquer en premier car les informations de portée des nœuds ne sont pas contenues dans les corpus.

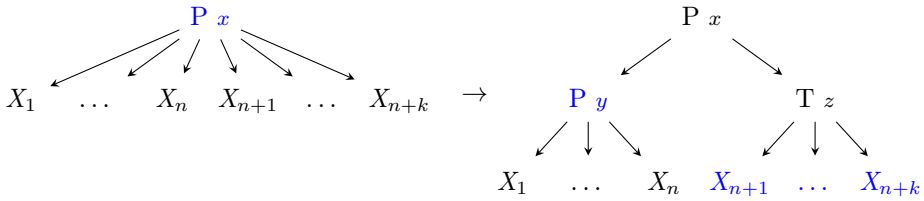


FIGURE 3.2 – Le nouveau nœud P a moins de fils qu'avant l'étape de transduction et le transducteur restera dans le même état pour le traiter avec la même étiquette de sortie que le nœud parent. Une telle règle sera instanciée de manière à ce que T soit binaire. Les types y et z respecteront bien sûr les règles d'une grammaire AB, c'est-à-dire que combinés ensemble le résultat sera x.

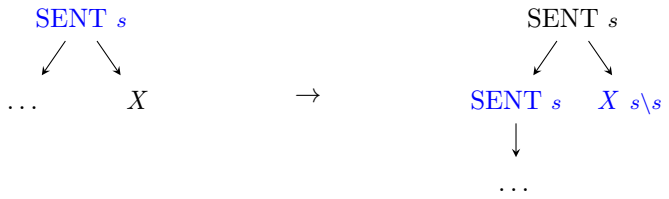


FIGURE 3.3 – La même règle sera appliquée pour $X \in \{ADV,\ PP\text{-}MOD,\ AdP\text{-}MOD, ...\}$.

Dans les figures 3.3 et 3.4, on remarque que seuls certains nœuds sont spécifiés, alors que le reste n'est pas important. Dans la suite, nous utiliserons le mot clef *tree*, qui représentera une suite quelconque de nœuds.

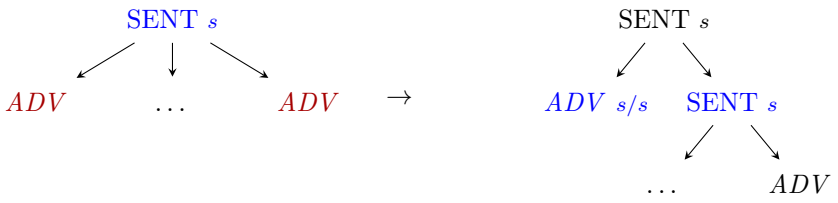
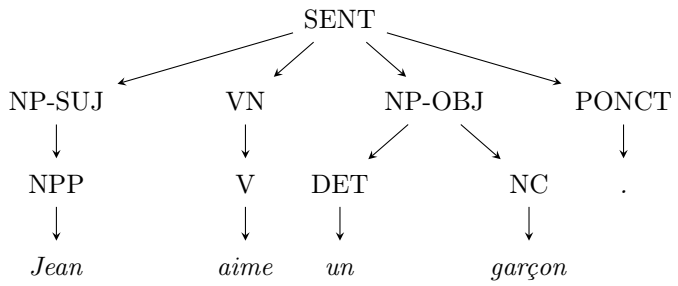


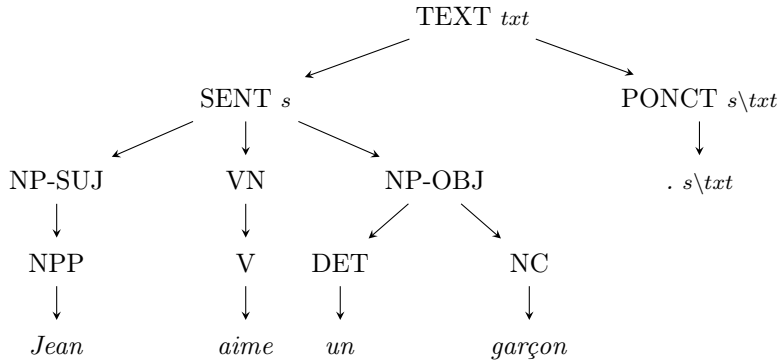
FIGURE 3.4 – Quand plus d'une règle peut être appliquée, nous suivons toujours le même ordre d'application.

3.1.4 Exemple d'utilisation du G-transducteur

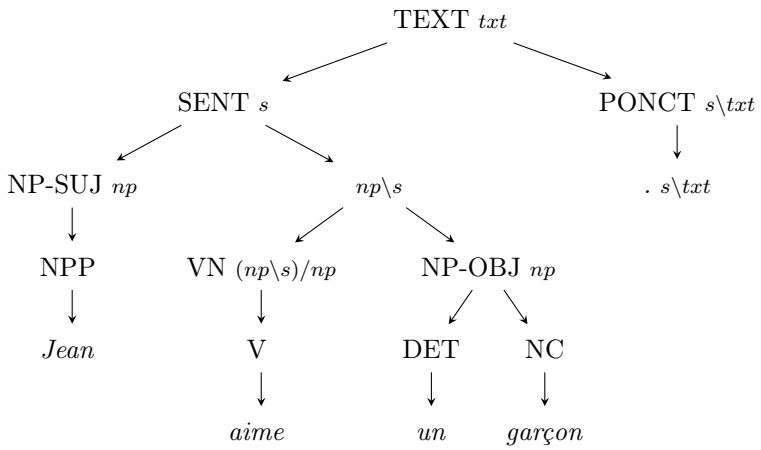
Avant de présenter en détail les règles de transduction, nous allons faire tourner le G-transducteur manuellement sur un exemple simple : *"Jean aime un garçon."*.



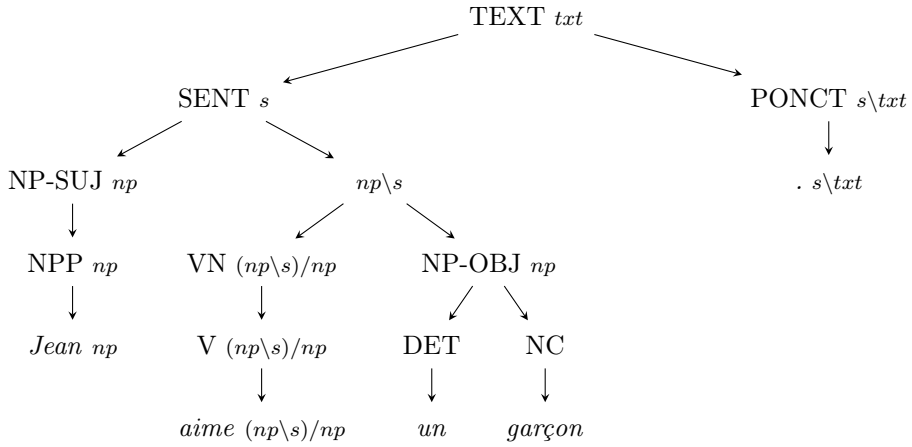
Le but sera donc d'avoir un arbre binaire où chaque nœud aura un type correspondant à une dérivation d'une grammaire AB. Dans un premier temps, nous gérons la ponctuation finale.



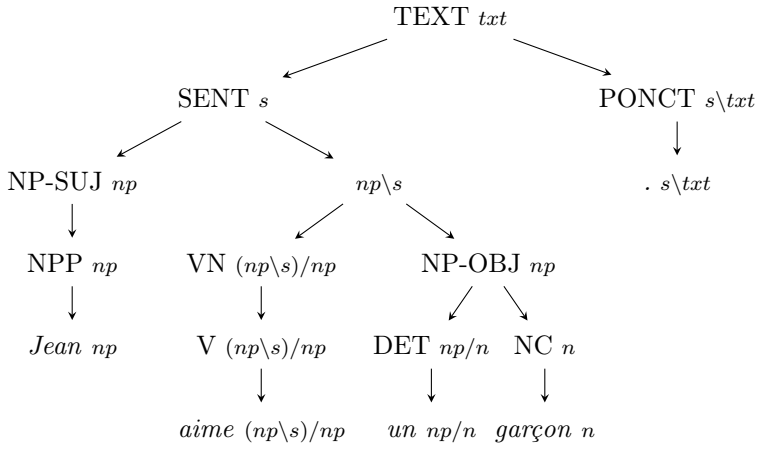
Ensuite le motif *NP-SUJ VN NP-OBJ* est reconnu d'un coup.



Les types sont hérités pour les nœuds *NP-SUJ* et *VN*, étant donné que ce sont des nœuds unaires.



Pour finir on détermine qui, parmi les deux fils du nœud *NP-OBJ*, sera foncteur et qui sera argument. Après cette étape, la transduction est terminée.



La transduction est ainsi terminée et l'arbre en résultat est binaire et représentatif d'une dérivation d'une grammaire AB.

3.2 Règles de transduction

Bien que le G-transducteur nous permette de spécifier de manière compacte des schémas de règles, nous avons quand même besoin d'écrire un grand nombre de règles de transduction dans l'idée d'obtenir une couverture satisfaisante des corpus (voir section 3.4). Dans sa version actuelle, le transducteur utilise 1 675 règles pour couvrir les corpus de Paris VII et Séquoia, listées par fréquence d'utilisation en annexe (voir le chapitre 6).

Nous décrirons schématiquement les idées et les principes derrière les règles de transduction pour permettre de mieux appréhender le fonctionnement de celui-ci.

Pour les règles de transduction, nous avons voulu rester aussi près que possible des méthodes usuelles pour analyser des phrases ainsi que des expressions complexes des grammaires catégorielles. Nous nous sommes fondés sur des exemples tirés de [Lambek, 1958; Morrill, 1994; Moortgat, 1997; Steedman, 2001] pour les fondations syntaxiques, mais aussi sur d'autres méthodes d'extraction, telles que [Hockenmaier, 2003, 2006; Moot, 2010a,b; Moortgat et Moot, 2001]. On peut aussi noter qu'on est dans le même esprit que Carpenter [1993], même si on ne se sert pas de règles lexicales. Ainsi, par exemple, nous pouvons citer les quelques règles principales qui nous ont servi de bases :

- Les noms communs (*NC* dans les corpus) seront autant que possible typés n.
- Les syntagmes nominaux (*NP*) seront le plus souvent typés np.
- Les adjectifs ont généralement le type soit n/n soit $n\backslash n$ en fonction de leur position par rapport au nom qu'ils qualifient.
- Lorsqu'ils sont arguments, les participes passés (*VPpart*) ont pour type $np\backslash s_p$ et les syntagmes infinitivaux (*VPinf*) $np\backslash s_i$. La forme parenthésée des corpus que nous utilisons ne différencie pas les participes passés utilisés dans une forme passive d'un passé composé. La structure des phrases *"Marie était partie"* et *"Marie était aimée"* sera donc exactement la même. Nous ne pouvons pas, hélas, faire de différence dans l'état actuel des choses sans tagger à nouveau manuellement une partie du corpus.
- Les subordonnées (*Ssub*) ont le type cs.
- Etc.

Nous utilisons donc plus que les trois ou quatre catégories de bases présentées dans la section 2.1 et en voici le résumé :

txt **:** correspond à une phrase terminée par une ponctuation.

s **:** correspond à une phrase ou un sous-arbre considéré comme une phrase.

np **:** correspond aux groupes nominaux et aux noms propres.

n **:** correspond aux groupes nominaux dont on a retiré le déterminant ou aux noms communs.

$np\backslash s_p$ **:** correspond aux participes passés ou présent et aux groupes participiaux.

$np\backslash s_i$ **:** correspond aux verbes à l'infinitif et aux groupes infinitivaux.

cs **:** correspond aux propositions subordonnées.

pf **:** correspond aux parenthèses fermantes.

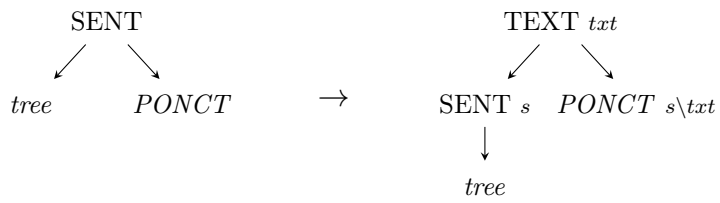
gf **:** correspond aux guillemets fermants.

Nous allons à présent nous focaliser sur le traitement des principales catégories d'arbres en fonction de leurs racines. Pour les divers syntagmes, qui peuvent avoir sur leur étiquette un rôle grammatical ajouté, nous ne précisons pas ceux-ci, sauf lorsque les règles à appliquer sont particulières à un rôle grammatical. Cela signifie que nous appliquons le même ensemble de règle pour traiter un nœud *NP*, *NP-OBJ* ou *NP-MOD*, etc. Bien que l'ordre d'application des règles soit très important pour la forme finale des arbres, nous avons préféré les présenter en fonction de leur racine et en traitant les plus complexes en fin de chaque section.

On note aussi que, par la suite, le type t sera utilisé pour définir un type quelconque hérité des transductions précédentes.

3.2.1 Les nœuds *SENT*

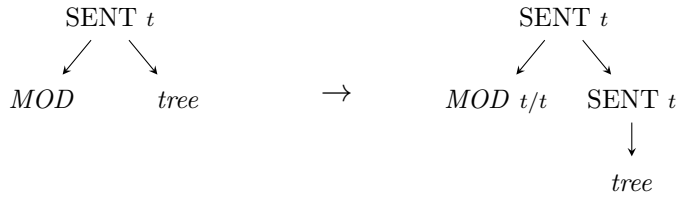
La gestion des nœuds étiquetés *SENT* donne la forme générale de la phrase. La première règle que nous appliquons est celle de la ponctuation finale, qui transforme d'ailleurs la phrase en *TEXT*. Ce traitement permet d'utiliser ensuite un même ensemble de règles pour la phrase jusqu'à ce qu'il ne reste plus que le noyau verbal.



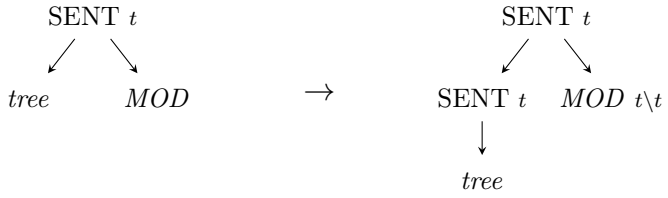
Les modificateurs de phrase

En début et fin de phrase apparaissent généralement des modificateurs, c'est-à-dire des adverbes étiquetés *ADV* ou bien d'autres sous-arbres correspondant à une étiquette $*$-*MOD*. Un modificateur ne change pas le sous-arbre

dont il est foncteur. Il aura donc presque systématiquement le type t/t ou $t\backslash t$, sauf cas particulier. Si le modificateur est en début de phrase, il sera traité comme suit :

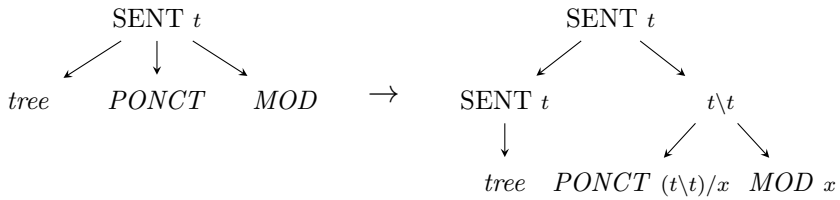


Et s'il est en fin de phrase :



Lorsqu'il y a plusieurs modificateurs, ils sont traités récursivement. Dans les deux règles ci-dessus, le nœud *MOD* peut être remplacé par : *NP-MOD*, *PP-MOD*, *ADV*, *AdP-MOD*, *Sint-MOD*, *Ssub-MOD*, *Srel-MOD*, *VPpart-MOD* ou *VPinf-MOD*. Il faut noter que le même traitement est effectué pour les interjections étiquetées *I*.

Une gestion particulière des modificateurs en fin de phrase est effectuée lorsque ceux-ci sont précédés d'une ponctuation. Dans ce cas, le nœud modificateur est traité comme s'il n'avait pas l'étiquette *-MOD* à la fin et la ponctuation récupère le type complexe.



Le type x sera remplacé par celui donné en majorité aux nœuds lorsqu'ils ne sont pas utilisés en tant que modificateurs. Dans le cas des adverbes, le type restera $t\backslash t$. Ce traitement a été effectué dans l'idée d'homogénéiser les types donnés à l'intérieur des sous-arbres des modificateurs.

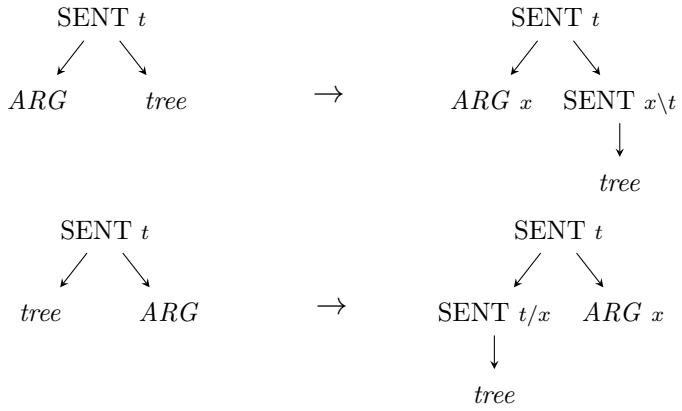
Les arguments du noyau verbal

Après avoir traité les modificateurs, nous nous focalisons sur les arguments du noyau verbal. Dans la généralité, ceux-ci se traitent simplement : l'argument

Etiquette	**type**	**Etiquette**	**type**	**Etiquette**	**type**
NP	np	*ET*	np	*PP*	pp
PP_A	pp_a	*PP_DE*	pp_{de}	*AP*	$n\backslash n$
VPinf	$np\backslash s_i$	*VPpart*	$np\backslash s_p$	*Ssub*	cs
Sint	s	*Srel*	s		

TABLE 3.1 – Récapitulatif des types donnés de préférence aux différents nœuds en fonction de leur étiquette. Ces types sont aussi donnés lorsque l'étiquette est adjointe d'un rôle grammatical comme *-SUJ*, *-OBJ*, *-ATS* ou *-ATO*.

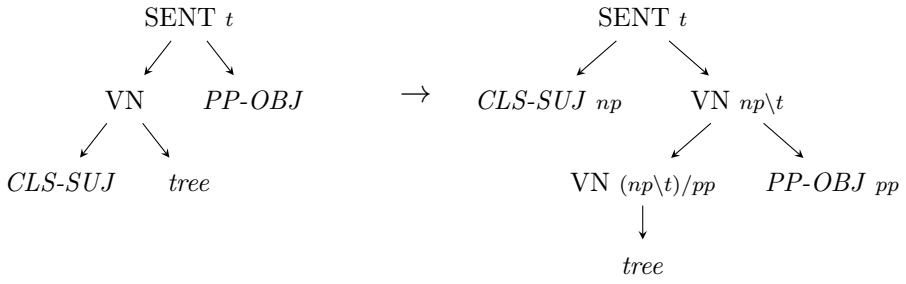
obtient un type simple sélectionné dans la table 3.1 et l'ensemble de la phrase devient foncteur.



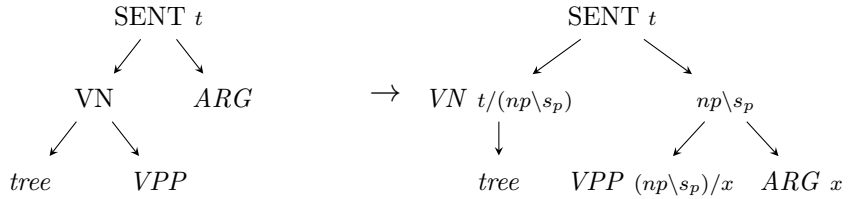
L'étiquette *ARG* peut être remplacé par n'importe quel nœud présenté dans la table 3.1, cependant on note que dans la majorité des cas le nœud en début de phrase est un sujet, alors qu'en fin de phrase, il représente majoritairement des objets du verbe ou des syntagmes utilisés en tant qu'attribut du sujet ou de l'objet. De même, la variable x est remplacée par le type idoine.

Cependant, certains cas doivent être gérés avant, de manière à créer des arbres de dérivation en adéquation avec le sens de la phrase. Nous présentons ici les mécanismes généraux. Il faut penser qu'ensuite certains cas du corpus nécessitent de fusionner des règles qui sont peu utilisées.

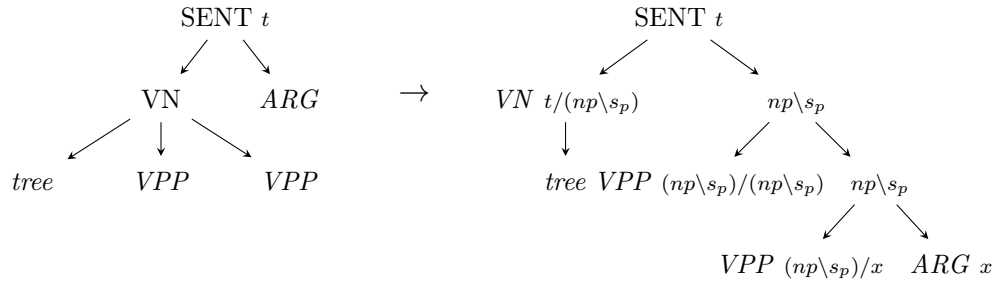
1. Tout d'abord, nous regardons si le sujet fait partie du noyau verbal ou non : c'est le cas lorsque le sujet est un clitique. Cependant, étant donné que nous souhaitons que le sujet soit l'argument le plus à l'extérieur de l'arbre de dérivation, nous traitons les clitiques sujets comme s'ils étaient des nœuds *NP-SUJ*, en renvoyant le clitique hors du sous arbre du noyau verbal.



2. Lorsque le noyau verbal se termine par un participe passé (ou par un verbe à l'infinitif) et qu'il est suivi directement par un argument du verbe, le noyau verbal prendra en argument un nouveau sous-arbre, de type $np\backslash s_p$ ou $np\backslash s_i$ et le nœud *VPP* ou *VINF* devient foncteur.

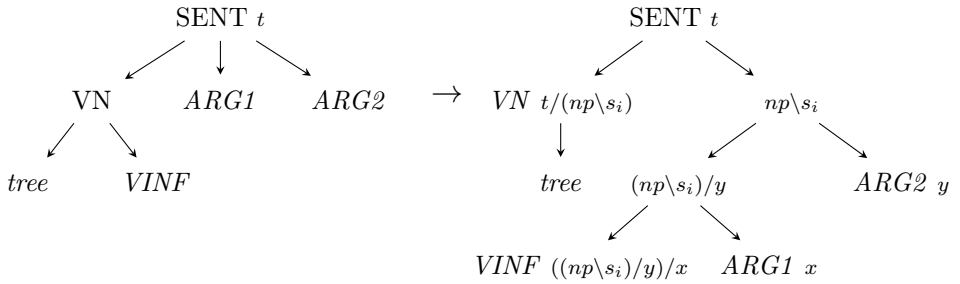


Cette manipulation est effectuée aussi bien dans le cas d'un *VPP* que d'un *VINF*, le type $np\backslash s_p$ devient juste $np\backslash s_i$. Dans le cas où le noyau verbal contient deux participes passés ou un participe passé suivi d'un verbe à l'infinitif (comme dans les phrases *"[...] François Staedelin a été élu président [...]"* et *"[...] a fait savoir que [...]"*), le premier participe passé prend en argument l'objet et le participe passé ou l'infinitif suivant. Seul le second participe passé peut être remplacé par un nœud *VINF*.

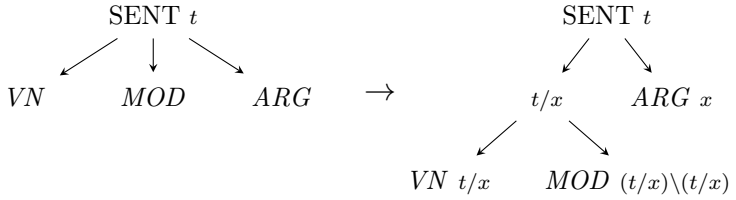


Il va sans dire qu'étant donné l'ordre de priorité imposé par le transducteur, on traite le cas ayant deux participes passés (ou du participe passé suivi de l'infinitif) avant de traiter celui où il n'y en a qu'un seul.

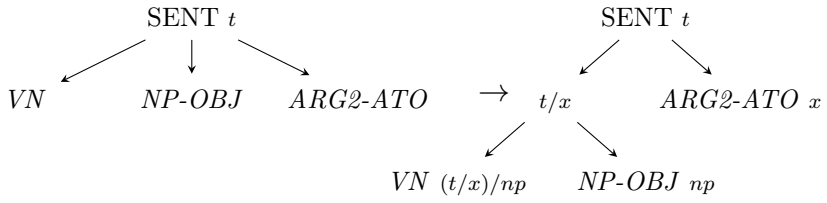
3. Lorsque deux objets suivent le noyau verbal et que celui-ci se termine par un nœud *VPP* ou *VINF* (par exemple des phrases construites sur le schéma : *"empêcher A de B"*), les deux objets sont pris en argument l'un après l'autre. On remplace simplement $np\backslash s_i$ par $np\backslash s_p$ lorsque le noyau verbal se termine par un nœud *VPP*.



4. Il arrive qu'un modificateur, parfois même deux, soient entre le noyau verbal et ses arguments. Dans ce cas, les modificateurs agissent sur le noyau verbal et celui-ci gère ses arguments comme s'il n'y avait pas d'adverbe entre lui et eux.



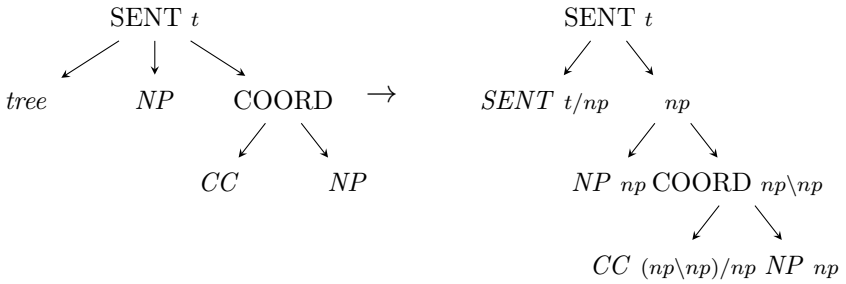
5. Le cas où l'objet est suivi d'un attribut d'objet ne nécessite pas d'utiliser cet attribut comme foncteur de l'objet : ce sont le plus souvent des cas comme *"voir notre économie croître"*, où aussi bien l'attribut d'objet que l'objet doivent être pris en argument par le verbe.



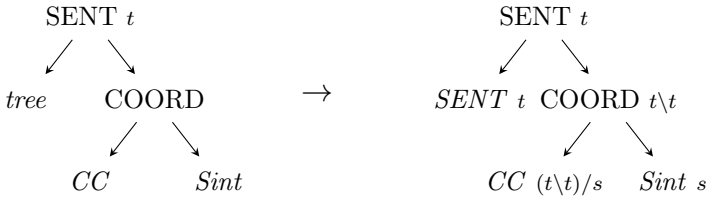
Divers

Nous allons à présent nous focaliser sur quelques cas dont le traitement nous a semblé être intéressant.

Coordination d'arguments : Dans ce cas de figure, un argument est suivi directement d'une coordination contenant ce même argument. L'argument et la coordination sont placés dans un sous-arbre qui prend le type correspondant à l'argument, comme on peut voir ici avec une coordination de nœuds *NP*.



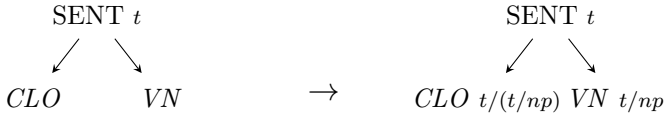
Coordination de phrases : Lorsque le sous-arbre du nœud *SENT* se termine par une coordination contenant un *Sint*, nous sommes dans le cas d'une coordination de phrases. Cette règle est placée très tôt dans le fichier de règles, de manière à ce que les deux phrases soient à peu près à la même hauteur dans l'arbre de dérivation. Ici, nous pouvons voir un cas idéal de coordination de phrases.



Lorsque le nœud *COORD* a plus de deux fils, il faut vérifier qu'il y a bien un noyau verbal parmi eux pour considérer qu'il s'agit bien d'une coordination de phrases.

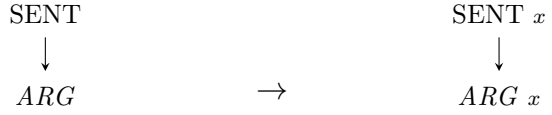
Noyau verbal remplacé par un participe : Lorsque la phrase ne contient pas de noyau verbal mais un nœud *VPP* ou *VPR*, on considère que celui-ci remplace le noyau verbal et les mêmes règles s'appliquent.

Gestion des clitiques : Lorsque le clitique est à l'extérieur du noyau verbal, il sera considéré comme argument de ce dernier, sauf si c'est un clitique objet antéposé. Dans ce cas, le noyau verbal hérite de la catégorie qu'il aurait dû avoir si son argument avait été à sa droite. Le clitique, quant à lui, devient foncteur et hérite du type complexe. En fonction de ce que remplace le clitique, les types changent aussi bien au niveau de celui-ci qu'au niveau du noyau verbal.



Lorsque le clitique objet est simplement étiqueté *CLO*, il remplace un groupe nominal. Il peut sinon remplacer un syntagme prépositionnel (*CLO-OBJ*, *CLO-DE_OBJ* ou *CLO-A_OBJ*) auquel cas *np* sera remplacé par les types idoines (pp, pp_{de} ou pp_a). Dans le cadre des grammaires AB, c'est le plus efficace qui soit faisable.

Phrases non verbales : Certaines "phrases", correspondant généralement aux titres d'articles ou à la signature des journalistes ne comportent pas de noyau verbal (exemple : *"Par Jean-Michel Normand", "Inflation et dettes publiques"*, etc.). Les phrases constituées d'un unique nœud représentent 1,5% du corpus, et nous ne voulions pas les laisser de côté. Le nœud *SENT* obtient alors le type que devrait avoir son fils et ce dernier est traité normalement. Les nœuds concernés sont *NP*, *VPpart*, *VPinf*, *AP* ou *PP*. Ainsi le type qui remplacera x sera respectivement np, $np\backslash s_p$, $np\backslash s_i$, $n\backslash n$ ou pp.



Une minorité de phrases est constituée de deux ou trois nœuds, généralement un groupe nominal et une ou plusieurs prépositions qui gravitent autour. Dans ces cas là, la phrase aura le type np et les syntagmes prépositionnels seront considérés comme des foncteurs du nœud *NP*.

Phrases sans ponctuation finales : Certaines phrases ne se terminent pas par une ponctuation finale. Dans ce cas, elles sont traitées normalement, sauf qu'elles n'ont pas l'étiquette *TEXT*, qui est réservée aux phrases ponctuées.

3.2.2 Les syntagmes nominaux

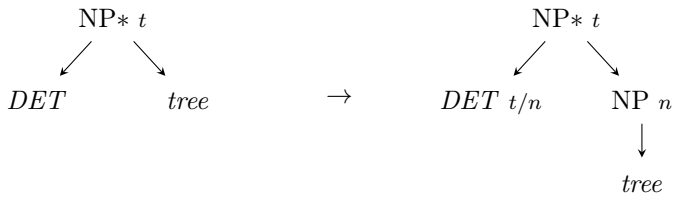
Les syntagmes nominaux sont étiquetés *NP* avec parfois une précision sur le rôle dans la phrase : *NP-SUJ* pour un sujet, *NP-OBJ* pour un objet, *NP-ATS* ou *NP-ATO* pour un attribut du sujet ou de l'objet et *NP-MOD* pour un modificateur, comme "cette année" ou bien "Mardi 31 décembre". Au niveau du traitement du sous-arbre, quelle que soit l'étiquette de la racine parmi celles citées plus haut, le traitement sera le même. La seule différence est qu'un nœud *NP-MOD* sera toujours considéré comme un modificateur alors que les autres seront considérés comme des arguments.

Le type donné aux modificateurs sera toujours x/x ou $x\backslash x$ en fonction du type du sous-arbre qu'il modifie. Un syntagme nominal *NP-MOD* en début de phrase, par exemple, aura donc comme type s/s. Les autres syntagmes nominaux auront pour type np dans tous les cas. Une fois le type de la racine donné, nous nous sommes focalisés sur la composition du sous-arbre pour donner un traitement homogène qui réduise le nombre de types associés aux mots du lexique.

Pour la suite des explications, on considèrera que t est le type de la racine du syntagme nominal et *NP*$*$ son étiquette.

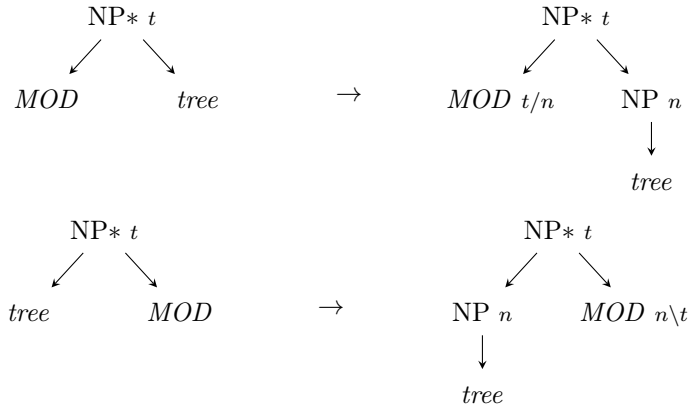
Les déterminants

Le premier schéma que l'on va traiter sera celui du déterminant en début de syntagme nominal. Le déterminant va récupérer le type complexe tandis que le reste du syntagme gagne le type n. On délaisse aussi la spécification de la racine en ne gardant que l'étiquette *NP*, car l'information quant à sa fonction dans la phrase n'a pas besoin d'être redondante.



Les modificateurs

Ensuite, on effeuille le sous-arbre en retirant les modificateurs petit à petit, comme montré ci-dessous.



Cette méthode a pour but de toujours laisser au noyau le type n. Les modificateurs auront le plus généralement le type n/n ou $n\backslash n$ sauf s'il n'y a pas eu de déterminant avant pour absorber le type t de la racine. Les modificateurs peuvent être :

- un adjectif ou un syntagme adjectival : *ADJ* ou *AP* ;
- une préposition étiquetée *PP* ;
- un adverbe ou une proposition adverbiale : *ADV* ou *AdP* ;
- une proposition subordonnée, proposition relative ou proposition conjuguée interne, respectivement *Ssub*, *Srel* et *Sint* ;
- un participe passé ou une proposition participiale : *VPP* ou *VPpart* ;
- un infinitif ou une proposition infinitive : *VINF* ou *VPinf*.

Cas particuliers

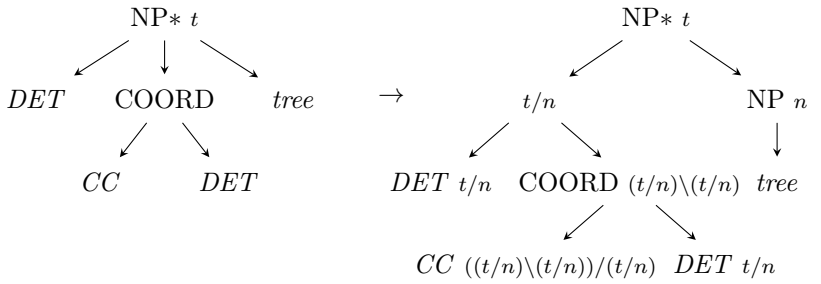
Les noms propres : Les noms propres prennent le type *np* et le reste de l'arbre devient foncteur. En effet, un nom propre peut servir de syntagme nominal à lui tout seul, comme vu dans la section 2.1.

Deux noms communs à la suite : Ce sont des cas tels que "l'année 1990" ou "juin 1985", mais aussi certains rares cas mal étiquetés tels que "M. Parretti" où "Parretti" n'apparaît pas comme un nom propre. Nous avons pris le parti de considérer le deuxième nom commun comme un modificateur du premier.

Nom commun suivi d'un nom propre : C'est le cas pour "M. Vianet", par exemple. On veut alors que le nom propre ait le type *np* et le nom commun qui le précède obtient donc le type t/np.

Les coordinations

L'idée globale pour traiter une coordination est toujours la même : on va créer un nœud intermédiaire dont le type correspond à celui attendu par chacun des mots ou groupes de mots liés par la coordination. Voici par exemple une coordination de déterminants comme *"entre 200 et 300 francs"*.



Les trois types de coordinations que l'on peut trouver dans les syntagmes nominaux sont les suivants :

Coordination de déterminants : C'est le cas le plus simple à traiter, comme montré par la figure ci-dessus. Les déterminants sont placés en début de syntagmes nominaux et la coordination ne contient que deux nœuds.

Coordination de modificateurs : Les coordinations de modificateurs fonctionnent comme les déterminants, mais celles-ci peuvent être placées n'importe où dans le syntagme nominal. Il faut cependant noter que les corpus présentent parfois des coordinations qui devraient couvrir plus que simplement le dernier modificateur avant la coordination, comme montré dans la figure 3.5.

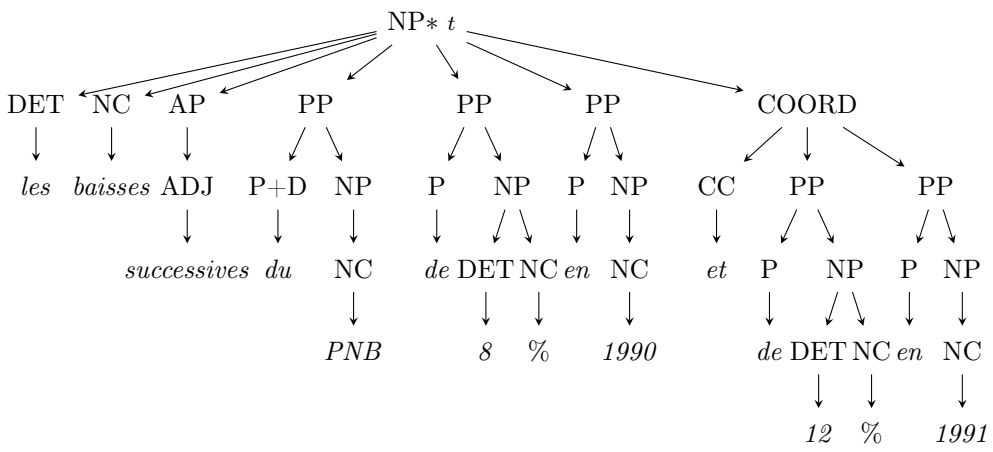
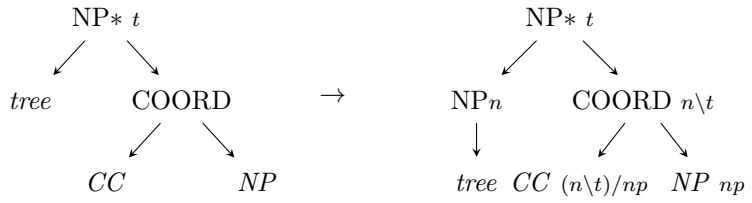


FIGURE 3.5 – Dans ce cas précis, on souhaite que les deux *PP* dans la coordination soient coordonnés avec les deux propositions prépositionnelles situées avant la coordination, et non simplement la dernière.

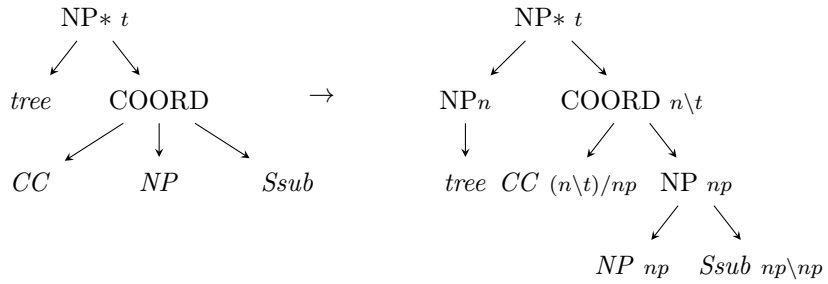
On remarque que si l'on utilisait le calcul de Lambek dans son intégralité, ce ne serait pas nécessaire. On pourrait effectuer la coordination juste avec le premier *PP*, et ensuite appliquer le principe d'associativité :

$$n\backslash n \quad n\backslash n \vdash n\backslash n$$

Coordination de groupes nominaux : Généralement placés à la fin du syntagme nominal, il existe deux catégories de coordinations. La première catégorie est simple à traiter, car le transducteur est certain du fait que ce soit une coordination de syntagmes nominaux, étant donné que la coordination contient une conjonction et un groupe nominal uniquement.

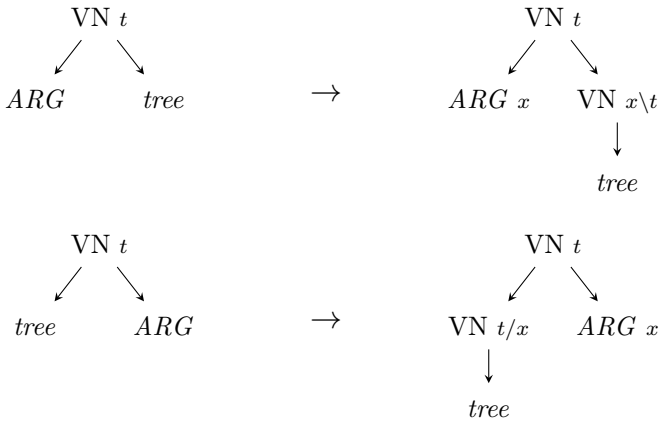


La seconde catégorie englobe les cas où la coordination contient plusieurs nœuds, c'est-à-dire un nœud *NP* et ses modificateurs. Il faut donc créer un sur-nœud *NP* qui englobe la totalité du syntagme nominal compris dans la coordination.



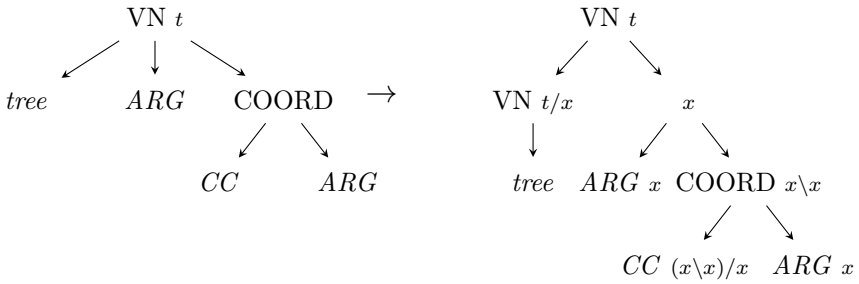
3.2.3 Le noyau verbal

Le noyau verbal contient, dans la majorité, des clitiques (pour une explication détaillée du rôle des clitiques, voir [Kraak, 1998]), des modificateurs, des participes présents ou passés, et des proposition infinitives. On peut aussi y trouver des groupes nominaux, des adjectifs, des syntagmes prépositionnels, des propositions subordonnées ou des propositions internes.



On note une forte similitude avec les règles utilisées pour les nœuds *SENT* : ces derniers seront systématiquement considérés comme des arguments du reste du noyau verbal. Lorsque l'on a une coordination d'arguments, celle-ci est traitée de manière à ce que le noyau verbal ne la différencie pas d'un simple argument. On note qu'il n'y a pas de coordination de clitiques [4].

4. Un cas isolé de coordination de clitique sujet, est disserté dans l'article [Miller et Monachesi, 2003].

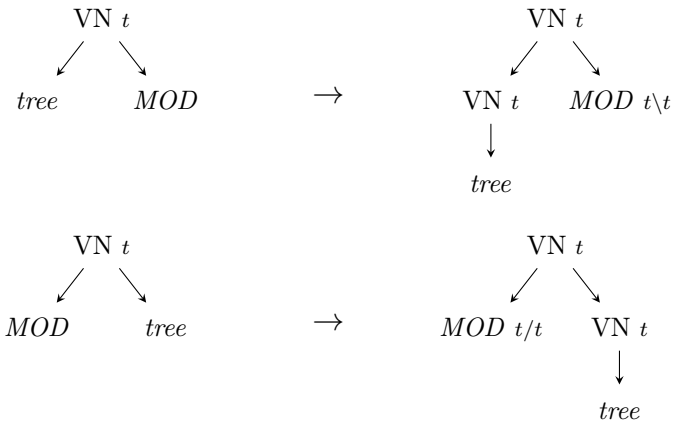


Les types donnés dans ce cas aux arguments sont référencés dans la table 3.1 (36), auquel nous pouvons ajouter, étant donné que nous pouvons aussi trouver des POS-tags dans les noyaux verbaux, les types référencés dans la table 3.2. Un cas particulier est celui de deux participes passés à la suite.

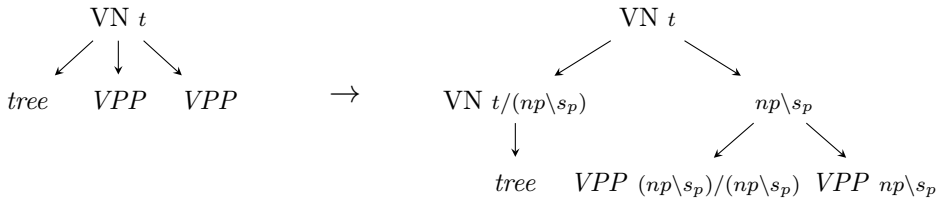
Etiquette	**type**	**Etiquette**	**type**	**Etiquette**	**type**
NC	n	*ADJ*	$n\backslash n$	*P*	pp
VINF	$np\backslash s_i$	*VPP*	$np\backslash s_p$	*VPR*	$np\backslash s_p$
CLR	cl_r	*CLS*	np	*CLO-OBJ*	np
CLO-P_ OBJ	pp	*CLO-A_ OBJ*	pp_a	*CLO-DE_ OBJ*	pp_{de}

TABLE 3.2 – Récapitulatif des types donnés de préférence aux différents nœuds en fonction de leur POS-tag. On note que pour les clitiques *CLR* et *CLS*, on peut avoir des précisions quant à leur rôle grammatical. Les clitiques objets, eux, seront considérés comme un argument du verbe uniquement s'ils sont à sa droite. A la gauche du verbe, nous sommes dans le cas d'une inversion, et le clitique objet devient foncteur du verbe.

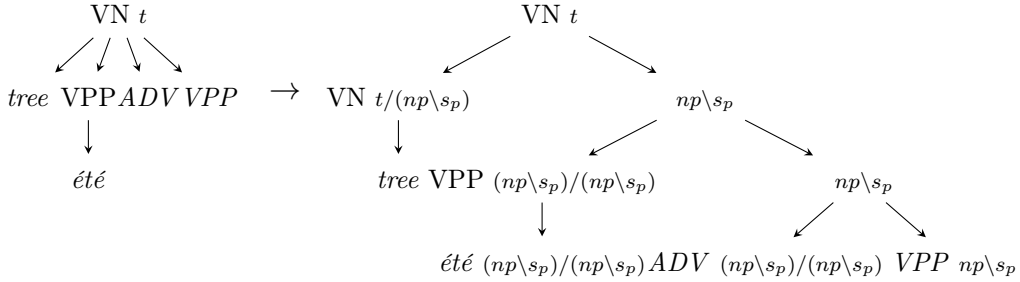
Les modificateurs qui apparaissent généralement dans les noyaux verbaux sont des adverbes, bien que l'on trouve des nœuds *NP-MOD*, *PP-MOD* ou *AdP-MOD*. Dans tous les cas, nous utilisons les deux règles suivantes :



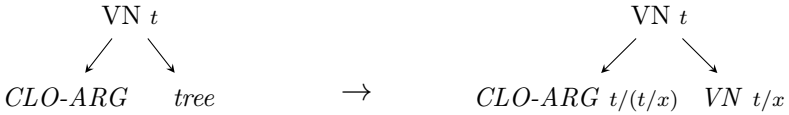
Lorsqu'il y a deux participes passés à la suite, nous préférons considérer que le second est argument du premier, comme vu pour la gestion globale des phrases.



Dans tous les cas, nous souhaitons que le premier *VPP* prenne le second en argument. Dans la majorité des cas (533 sur 537), cette construction correspond à *"été + participe passé"*, mais le corpus présente quelques exceptions où le second participe passé est en fait utilisé comme un adjectif (*"[...] se sont vues dépossédées [...]"* par exemple). Un cas particulier des deux participes passés à la suite est l'insertion d'un adverbe entre les deux. Dans ce cas, et uniquement si le premier est "été", l'adverbe portera sur le second participe passé (exemple : *"[...] été clairement demandé [...]"*).



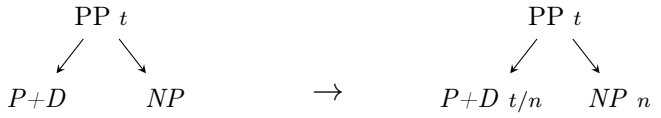
Les clitiques objets, lorsqu'ils sont placés à gauche du noyau verbal, ont d'office un rôle de foncteur, comme vu dans la section 3.2.1. Ainsi, en fonction de ce qu'ils remplacent (un groupe nominal ou une préposition) la catégorie du verbe ainsi que celle du clitique changeront.

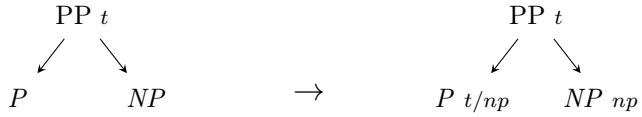


Si *ARG* est remplacé par *OBJ* ou rien, x sera égal à np, s'il est remplacé par *P_ OBJ*, *A_ OBJ* ou *DE_ OBJ*, x sera respectivement remplacé par pp, pp_a ou pp_{de}.

3.2.4 Les syntagmes prépositionnels

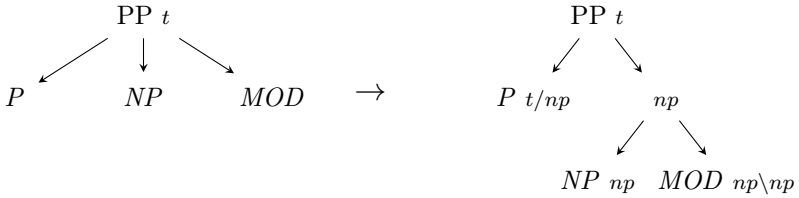
Les syntagmes prépositionnels ont tous la même formation théorique : une préposition (étiquetée P ou $P{+}D$) suivie d'un groupe nominal. Les cas les plus fréquents seront donc :





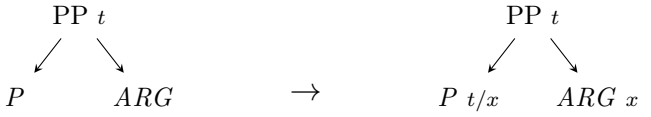
L'étiquette NP peut être remplacée par PRO, $PROWH$ ou ADJ sans qu'il y ait de modification de type ; cependant lorsqu'elle est remplacée par NC, le type devient systématiquement n. On note que, lorsque le déterminant est hors du groupe nominal, celui-ci a le type n au lieu de np.

Il peut arriver qu'un modificateur qui s'applique forcément sur le groupe nominal ne fasse pas partie du sous-arbre de celui-ci.



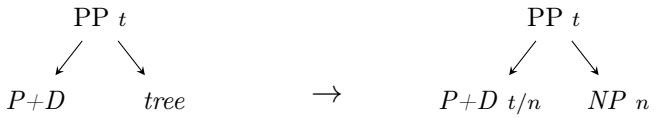
Le modificateur peut être un autre syntagme prépositionnel, un syntagme adjectival ou, dans de rares cas, un second groupe nominal.

Certaines prépositions sont suivies par autre chose qu'un groupe nominal, auquel cas la préposition est forcément étiquetée P et à part le type donné à l'argument, on utilise la même règle que dans le cas basique.



Le nœud ARG et son type associé peuvent être remplacés par : $VPinf$: $np\backslash s_i$, $Ssub$: cs, $ADVWH$: s/s, $Srel$: s, $Sint$: s, VN : $np\backslash s_p$, AP : $n\backslash n$, AdP : $n\backslash n$, ADV : $n\backslash n$ ou PP : $n\backslash n$.

Il arrive aussi que, à cause d'erreurs d'annotations, le contenu du groupe nominal soit directement dans le sous-arbre du syntagme prépositionnel. Dans ce cas, tous les nœuds après la préposition sont regroupés sous un nœud NP.



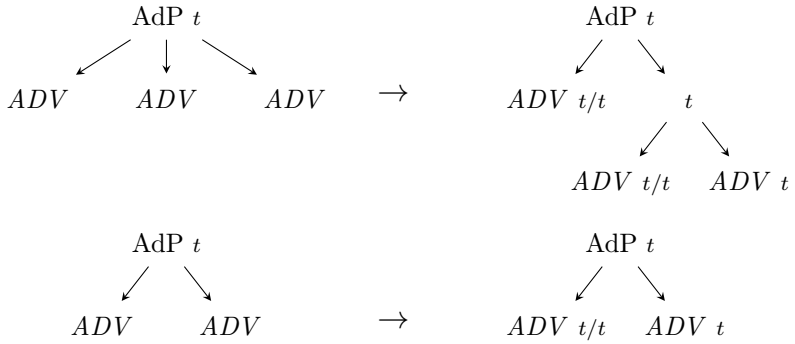
Cette règle est utilisée parmi les dernières. S'il y a une préposition *P*, le type de l'arbre est bien sûr *np*.

3.2.5 Autre

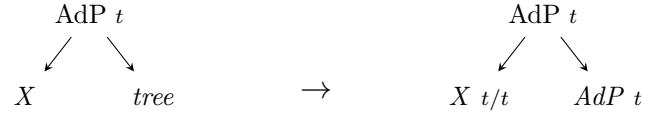
Pour les autres composants syntaxiques, nous avons fait un récapitulatif rapide des règles utilisées, pour donner un aperçu du fonctionnement de celles-ci. En effet, elles sont assez simples.

Les syntagmes adverbiaux

Ils sont étiquetés *AdP*, *AdP-ATS*, *AdP-OBJ* ou *AdP-MOD* et peuvent contenir en plus des adverbes : des prépositions, des subordonnées et parfois des groupes nominaux, des syntagmes adjectivaux ou des subordonnées relatives. Dans la majeure partie des cas, les sous-arbres se composent de deux ou trois adverbes.



Lorsque le premier nœud du sous-arbre est une préposition (étiquette *P*), un déterminant ou une conjonction de coordination, celui-ci est foncteur du reste du sous-arbre, comme on peut le voir ci-dessous.



Il faut préciser que *X* peut être remplacé par *DET*, *P* ou *CC*.

Comme vu précédemment, lorsque le sous-arbre contient un *NP*, *PP*, *AP*, *Ssub* ou *Srel*, ceux-ci seront automatiquement considérés comme des arguments du reste du sous-arbre. De même, les coordinations d'adverbes ou de syntagmes adverbiaux sont gérés comme noté ci-dessus, à l'exception près qu'aucun type n'est précisé, comme montré figure 3.6.

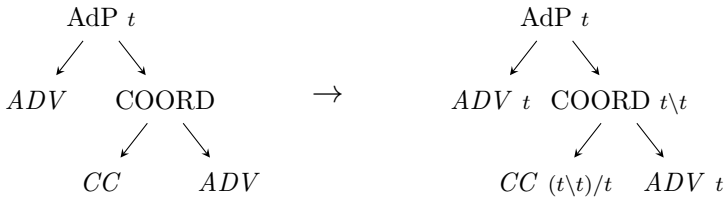
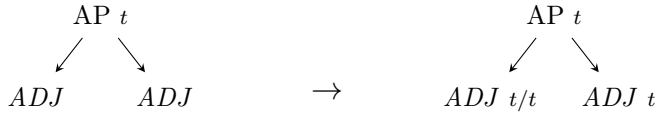


FIGURE 3.6 – Coordination d'adverbes. On note que le type de l'étiquette *AdP* est hérité par les deux adverbes.

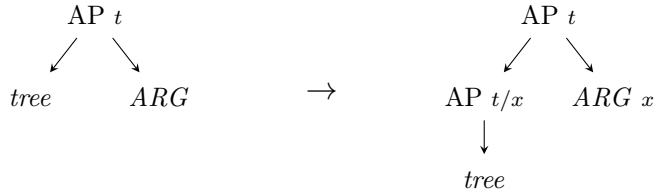
Les syntagmes adjectivaux

Ces sous-arbres voient leurs racines étiquetées *AP*, *AP-ATS*, *AP-ATO*, *AP-SUJ* (cas exceptionnels) ou *AP-MOD*. Cependant, quelle que soit cette étiquette, les règles utilisées seront les mêmes.

Un cas particulièrement fréquent est celui de deux adjectifs à la suite (exemple : *"[...] deux derniers [...]"*), et nous appliquons la règle suivante :



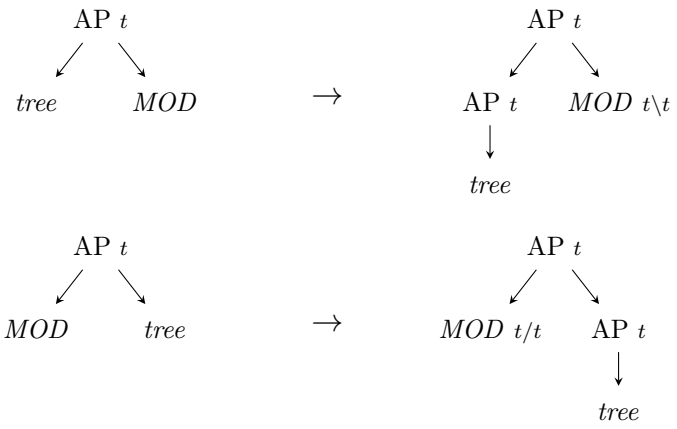
Il y a ensuite de nombreux nœuds qui seront pris en argument du reste de l'arbre, systématiquement en fin de syntagme. La liste des types donnés aux arguments est la même que dans la table 3.2 (page 45), en modifiant le type donné aux participes passés *VPP* par $n\backslash n$ [5].



Les arguments que l'on peut trouver dans un syntagme nominal sont : *PP*, *Ssub*, *NC*, *NP*, *NPP*, *Srel*, *VPP*, *ADJ*, *Sint*, *VPpart*, *PP-A_ OBJ* ou *ADJ*.

Les autres nœuds que l'on peut trouver dans les syntagmes adjectivaux sont utilisés comme foncteurs.

5. Les participes passés peuvent être utilisés comme des adjectifs, comme montré par [Carpenter, 1993; Morrill, 2001; Hockenmaier et Steedman, 2007; Moot, 2013]. Nous avons donc, dans ce cas de figure, privilégié le type donné à un adjectif.

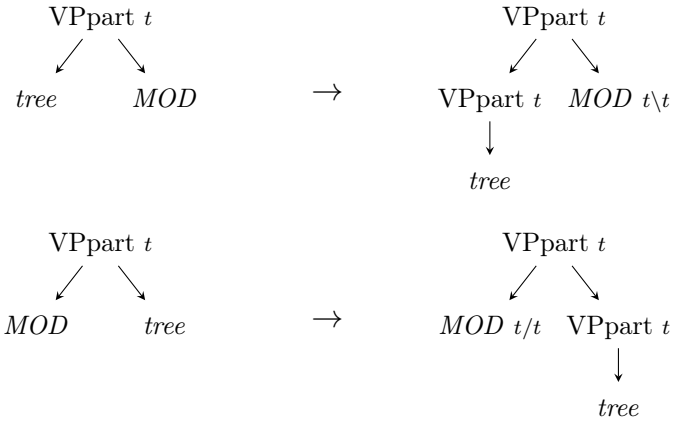


Les foncteurs peuvent être : *ADV*, *PREF*, *AdP*, *PP-MOD*, *CC*, *VPinf*, *VPinf-MOD*, *PP-MOD* ou *Ssub-MOD*.

Les coordinations de syntagmes adjectivaux ou d'adjectifs se gèrent exactement comme pour les syntagmes adverbiaux (voir figure 3.6).

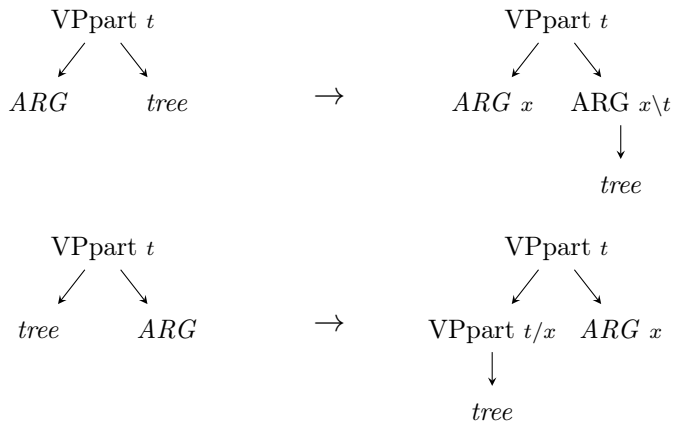
Propositions participiales et infinitives

Les règles de transductions pour les propositions participiales sont proches de celles utilisées pour les noyaux verbaux. On retrouve en effet les modificateurs qui ne changent pas le type du nœud.



Les modificateurs sont : *ADV*, *PP-MOD*, *Sint-MOD*, *NP-MOD*, *Ssub-MOD*, *VPint-MOD*, *VPpart-MOD*, *AP-MOD* ou *AdP*. Le mot clef *tree* et sa transformation en *VPpart* peut être remplacé par un simple *VPP* ou, dans de plus rares cas, par un *VN* qui contiendra lui-même le participe.

Les autres nœuds sont pris en argument par le reste de la proposition.

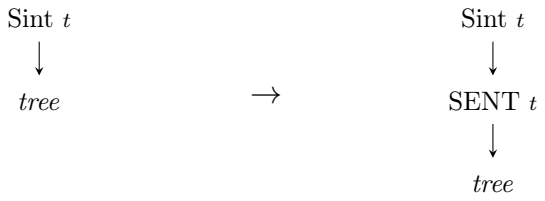


Le nœud *ARG* peut être remplacé par : *PP*, *PP-P_OBJ*, *PP-DE_OBJ*, *ADJ*, *NP* ou *NP-SUJ*. Encore une fois on a des occurrences de règles où *tree* est remplacé par *VN* ou *VPP*.

Pour les propositions infinitives, les règles sont construites sur le même schéma, les occurences de *VPpart* étant à remplacer par *VPinf* et celles de *VPP* par *VINF*.

Propositions conjuguées internes

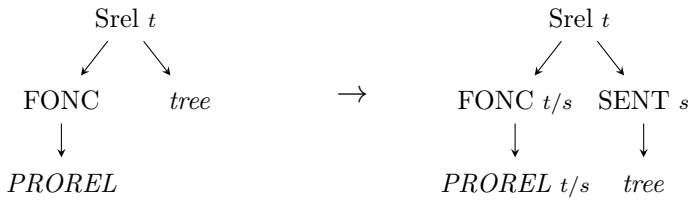
Les propositions conjuguées internes, notées *Sint*, sont construites comme des phrases. Il nous a donc semblé normal de ne pas réécrire les règles. Le sous-arbre généré est traité comme un nœud *SENT*, avec le type s sauf si le nœud a servi de modificateur.



Propositions relatives

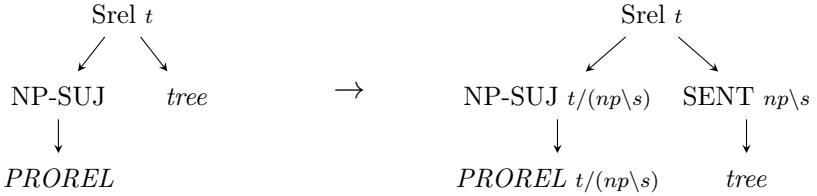
Les propositions relatives commencent en majorité (94% du temps) par un pronom relatif, suivi d'une phrase verbale. Elles sont difficiles à traiter avec les grammaires AB, et l'analyse que nous en avons fait, si elle fonctionne parfaitement avec les propositions qui commencent par le pronom "qui", n'a pas pu être plus optimisée pour les autres pronoms. Un exemple d'analyse que nous voudrions faire pour les pronoms relatifs autres que "qui" est donné dans le chapitre 2, avec la phrase *"Le livre que Marie a lu"*. Avec notre méthode d'analyse, nous ne pouvons pas donner à *"que"* le type $(n\backslash n)/(s/np)$, ce qui

correspondrait pourtant à l'inversion faite. On va donc appliquer une règle qui considère que le nœud *PROREL* (dont, qui, que, etc.) ou *PROWH* (où) est foncteur de tout le reste de la phrase, qui est regroupée sous un nœud *SENT*.

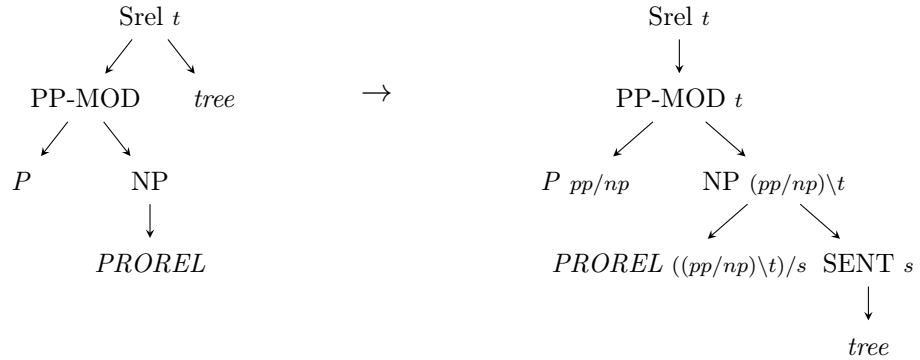


Le nœud *FONC* est remplacé par : *NP-OBJ*, *NP-ATS*, *NP-MOD*, *PP-DE_OBJ* ou *PP-P_OBJ* ; à la place d'un *PROREL* on peut trouver un *PROWH*.

Un cas particulier est à noter : lorsque le nœud *PROREL* est considéré comme un sujet, le type de la phrase s'en ressent et devient $np\backslash s$.



On vérifie ensuite qu'il n'y a pas de pronom relatif encastré dans un syntagme prépositionnel. Si c'est le cas, c'est le pronom relatif qui récupère le type complexe, alors que la préposition garde la catégorie usuelle (pp/np) et que la fin du sous-arbre est placée sous un nœud *SENT*.

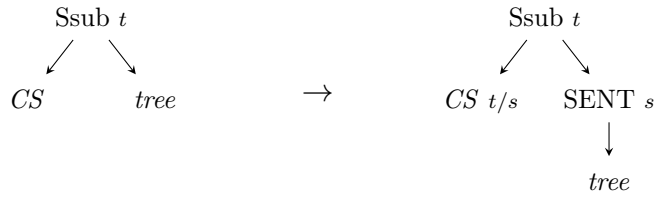


Le nœud *PP-MOD* peut être remplacé par *PP-P_OBJ*, *PP-A_OBJ* ou *PP-DE_OBJ*.

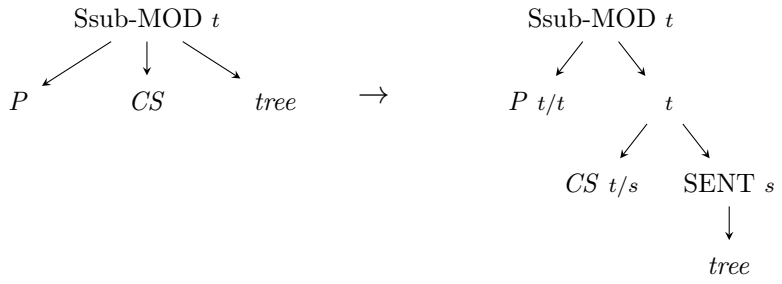
Pour les quelques propositions relatives qui ne sont pas traitées par ces règles, comme pour les nœud *Sint*, la totalité du sous-arbre est placé sous un nœud *SENT*.

Propositions subordonnées

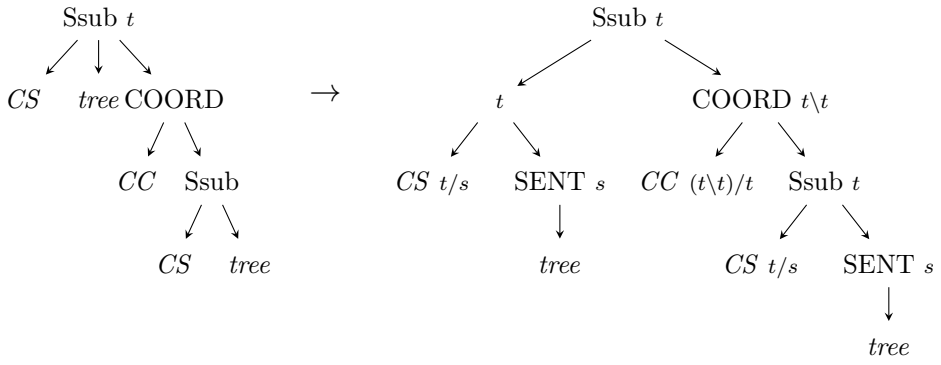
Les propositions subordonnées sont caractérisées par le fait qu'elles commencent par une conjonction de subordination, bien que celle-ci puisse être précédée par un adverbe ou une préposition. Une fois que la conjonction est mise de côté, on veut que les règles des *SENT* s'appliquent. Aussi la majorité des subordonnées sont traitées comme s'en suit :



Pour les cas où la conjonction est précédée par un modificateur, on utilise la règle suivante :



On peut avoir, en modificateur, soit un adverbe soit une préposition.
Il faut penser avant à gérer le cas des coordinations de subordonnées.



Pour les dernières propositions subordonnées que nous n'avons pas pu traiter avec ces règles, nous utilisons la même astuce que pour les nœuds *Srel* et *Sint*, et considérons que la totalité de la proposition est une phrase.

3.3 Implémentation

L'implémentation du transducteur, SynTAB [6], a été faite en `C++`, et testée sous Linux et Mac OS.X.

Nous avons d'abord envisagé d'utiliser le langage `XSLT`, car il permet de manipuler facilement des arbres sous forme XML, et que le corpus complet est disponible sous ce format. Cependant ce langage aurait nécessité d'implémenter chaque règle directement en `XSLT`, ce qui aurait été fastidieux et propice aux erreurs. Qui plus est, les ponctuations situées en milieu de phrase nécessitent un traitement particulier qu'il aurait été difficile d'implémenter élégamment en `XSLT`.

Une autre possibilité aurait été d'exprimer les transductions avec le language d'expressions rationnelles disponible dans le logiciel Standford Tregex, Tsurgeon (voir article [Levy et Andrew, 2006b]). Cependant la syntaxe de Tsurgeon est certes très puissante mais aussi très complexe, et étant donné le nombre de règles que nous avions à définir cela aurait été excessivement long pour un bénéfice négligeable.

Pour illustrer nos propos, voici un exemple de règle simple que l'on applique lorsqu'un groupe nominal est composé d'un déterminant, d'un adjectif et d'un nom commun :

```
NP < DET $+ (ADJ=a $+ NC=n)
adjoin (NC = new NC*) n
move a >0 new
```

Ce qui se traduit par : "Lorsque l'on recontre un nœud *NP* avec pour fils *DET*, *ADJ*, *NC*, on nomme *ADJ* en *a* et *NC* en *n*. On rajoute un nœud nommé *new* avec l'étiquette *NC* au dessus du nœud nommé *n*. Enfin, on déplace le nœud nommé *a* le plus à gauche possible du nœud *new*". Cet exemple ne gère pas les types, même si le logiciel le peut : nous avons préféré éviter de les ajouter pour ne pas le surcharger.

Nous avons donc décidé de créer un langage de description de règles spécifique, décrit en section 3.3.2, qui permet de spécifier les règles de transduction simplement [7] tout en rajoutant de nombreux contrôles permettant de détecter les erreurs humaines lors de la saisie des règles. Pour implémenter cela, nous avions besoin d'un langage plus généraliste que le `XSLT`, nous avons donc opté pour le `C++` qui offre une plus grande souplesse que le `XSLT`.

Comme montré sur le schéma 3.7, les arbres sont donc dans un premier temps passés au programme `reparer` (qui les uniformise), puis utilisés, conjointement à un fichier de règles, comme entrée du transducteur. A partir des arbres

6. Syntactic Tree to AB grammar.

7. Tout du moins de manière simplifiée par rapport aux expressions rationnelles de Tsurgeon.

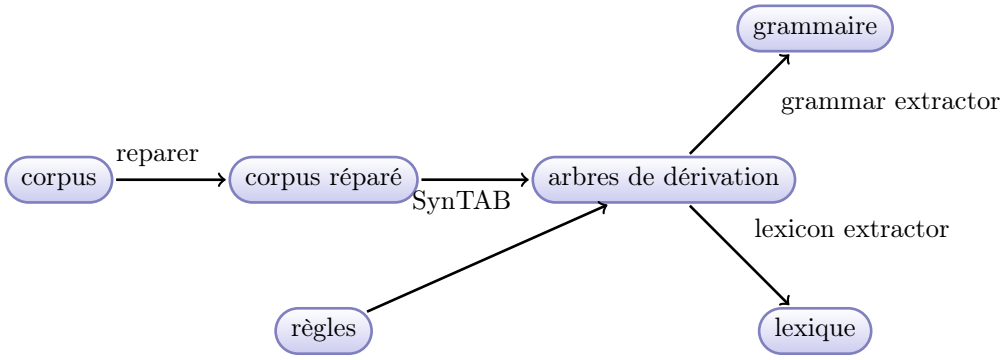


FIGURE 3.7 – Schéma global de notre implémentation.

de dérivation, on peut extraire soit une grammaire AB, soit un lexique, qui est la forme la plus usuelle pour représenter une grammaire catégorielle.

Il est important de noter que nous avons voulu construire un outil qui soit utilisable pour d'autres langues et d'autres corpus : ainsi l'implémentation est totalement séparée du fichier de règles et les utilisateurs qui voudraient créer leurs propres ensembles de données n'auraient pas à toucher au cœur du code pour utiliser le transducteur. De même, n'importe quel corpus sous forme parenthésée peut être pris en paramètre.

3.3.1 SynTAB

Le cœur de notre implémentation est la bibliothèque `TransducTree`. Elle contient l'implémentation de toutes les structures de données que nous utilisons dans les différents programmes. Elle se compose de plusieurs classes :

La classe `Type` qui représente les types au sens de Lambek, et implémente la vérification de la cohérence des types dans les arbres binaires.

La classe `Tree` qui implémente les arbres (binaires et planaires). Chaque nœud de l'arbre contient une étiquette et un type ainsi que la liste de ses fils. Pour faciliter le parcours des arbres binaires, cette classe contient une classe fille `DepthIterator` qui implémente un parcours en profondeur.

La classe `TreeParser` se charge de lire les fichiers en entrée (arbres bien parenthésés) et de produire des arbres. Elle implémente de nombreux tests sur la syntaxe des fichiers et essaye de produire des messages d'erreur les plus clairs possibles en cas de problème.

La classe `Transducer` s'occupe de charger les règles puis de les appliquer aux arbres. Le fonctionnement de cette classe est expliqué en section 3.3.1.

La classe `Lexicalizer` permet de générer un lexique à partir d'une forêt d'arbres binaires typés et compte les occurrences d'un mot qui apparaît

et de ses types.

Algorithme de transduction

Une règle de transduction est composée d'une part d'un motif qui représente le nœud (et son sous-arbre) à traduire et d'autre part de la traduction binaire typée de celui-ci, comme vu en section 3.1. L'implémentation actuelle se contente de parcourir en profondeur l'arbre auquel on souhaite appliquer une transduction et, pour chaque nœud, de chercher dans la liste des règles [8] celle qui correspond au nœud que l'on est en train de traiter et à son sous-arbre. On applique ensuite la même méthodologie pour les descendants ; deux cas s'offrent alors à nous :

- La règle reconnaît un motif précis. La transduction va alors rajouter des nœuds intermédiaires qui ne seront pas traités par la suite : on passera directement aux fils d'origine.
- La règle prend en compte des suites de nœuds quelconques et un ou plusieurs nœuds particuliers. Les nouveaux parents des suites de nœuds seront alors traités.

Actuellement les motifs des règles sont testés un par un jusqu'à trouver celui qui correspond à l'arbre que l'on souhaite traiter. La recherche d'une règle est donc en $O(n)$, n étant le nombre de règles. Cela implique de sévères restrictions au niveau des performances. Il serait souhaitable d'avoir une recherche des règles en temps constant par rapport au nombre de règles possédées par le transducteur. Pour améliorer cet état de fait, on pourrait utiliser un système équivalent à ceux des logiciels tels que StandFord Tregex ou Tgrep2, qui permettent de rechercher rapidement un motif grâce aux expressions rationnelles.

3.3.2 Langage de description de règles

Nous avons eu à créer un langage de description de règles qui permette à la fois de saisir celles-ci facilement et d'offrir une certaine souplesse qui concorde avec la création théorique de notre transducteur. Toutes les règles ont la même structure globale :

```
(rule [options]
     (racine motif)
     (remplacement))
```

Mot-clef `rule` : annonce le début d'une règle.

8. Notre implémentation se contente de stocker la liste des règles dans l'ordre de leur lecture.

Options : la seule option possible actuellement est "use-ponct", pour préciser qu'il faut gérer la ponctuation. Dans le cas où cette option n'est pas activée, la règle ignore les ponctuations.

Racine : représente le nœud que l'on souhaite traiter.

Motif : sous-arbre de la racine. Lorsque celui-ci contient le mot-clef *tree*, on considère une suite de nœuds quelconque.

Remplacement : arbre qui va prendre la place de la racine et son sous-arbre.

Un certain nombre de contrôles sont appliqués sur les règles à chaque invocation du programme pour vérifier leur bon fonctionnement :

Vérification des types : à chaque niveau, on vérifie que l'argument et le foncteur s'associent ensemble.

Vérification de l'arité de l'arbre de remplacement : lorsque le motif de remplacement n'est pas binaire, la règle n'est pas acceptée.

Vérification de la racine : le motif de remplacement doit avoir la même racine (étiquette et type si ladite racine présente un type) que le motif d'origine.

Vérification des feuilles : le motif de remplacement doit présenter les mêmes feuilles que le motif d'origine, sauf utilisation du mot-clef *tree*.

Ainsi, on est certain, lorsqu'une règle est acceptée par notre programme, qu'elle est fonctionnelle et que le résultat de son application sera un arbre binaire avec des types cohérents entre eux.

On peut donner un exemple de règle figure 3.8. Cette règle est la première utilisée, dans le cas d'une phrase bien construite (les \ doivent être doublés lors de l'écriture des règles pour signifier au transducteur qu'on utilise le caractère spécial "\").

```
(rule use-ponct
     (SENT tree PONCT)
     ("TEXT:s"
          "SENT:s" "PONCT:s\\s"))
```

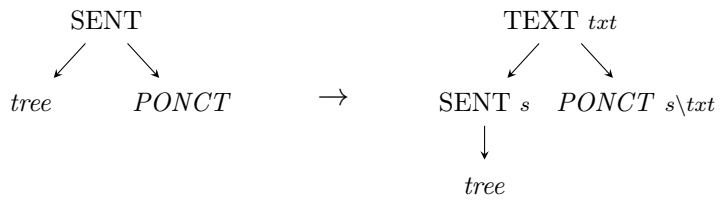


FIGURE 3.8 – Une règle et son schéma.

La transformation des règles papier en règles lisibles par le transducteur a demandé beaucoup de travail et de régularité.

3.3.3 Correcteur de corpus

Comme n'importe quels grands corpus, Séquoia et Paris VII présentent certaines erreurs et incohérences d'annotation. Plutôt que de multiplier le nombre de règles pour gérer ces erreurs, nous avons préféré implémenter un correcteur séparé, qui prend en charge les problèmes rencontrés plusieurs fois dans le corpus.

Correction des erreurs : lorsqu'un mot peut avoir deux fonctions dans une phrase, il arrive que l'annotation choisisse la mauvaise. Cela arrive parfois, par exemple, quand le participe passé d'un verbe est équivalent à une de ses formes conjuguées. Dans ce cas, l'étiquette donnée lors de l'annotation est *V* au lieu de *VPP* et le transducteur pense avoir affaire à une forme idiomatique et non à un verbe au passé composé (voir figure 3.9).

Une autre erreur d'annotation se retrouve sur la ponctuation finale : en théorie, le nœud *PONCT* est directement le fils du nœud *SENT*. En pratique, il arrive qu'il soit parfois plus profondément inscrit dans l'arbre. Dans ce cas, nous changeons simplement la ponctuation de place, de manière à ce qu'elle soit la plus extérieure possible.

Corrections des incohérences : étant donné que de nombreux annotateurs ont travaillé sur le corpus, il arrive que la même construction grammaticale ait une annotation différente d'une phrase à l'autre. Comme montré en figure 3.10, les coordinations sont les constructions qui présentent facilement des erreurs. Dans le corpus, bien que la majorité des coordinations aient la forme désirée (partie droite du schéma), des variantes de la construction de départ se trouvent en grand nombre. La phase de correction transforme ces structures en arbre montré à droite du schéma, et produit une annotation plus cohérente.

Transformation d'étiquettes : à cause de l'implémentation en `C++`, les guillemets et les deux points ne sont pas lus par le programme : ils sont considérés comme des caractères spéciaux. Nous les avons remplacés, respectivement, par \″ et `-COL-`, de manière à ce que le transducteur puisse les lire.

3.3.4 Extracteur de lexique

L'extracteur de lexique prend en paramètre un corpus arboré et restitue une liste de mots ou de POS-tags auxquels sont associés leurs différents types

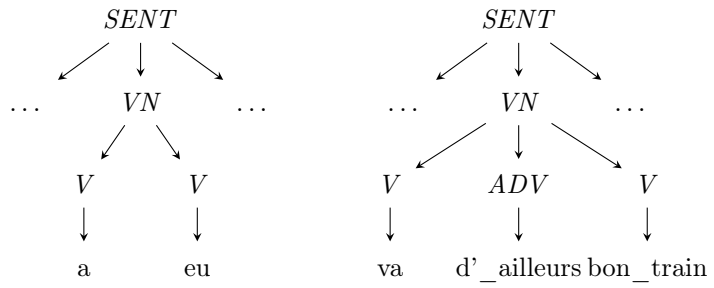


FIGURE 3.9 – Erreur d'annotation concernant un participe passé *"eu"*, facilement confondu avec l'expression idiomatique *"aller bon train"*.

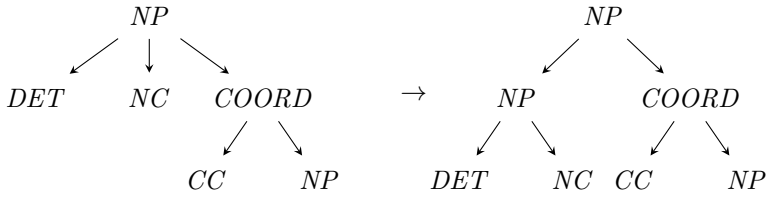


FIGURE 3.10 – L'arbre de gauche laisse entendre que la coordination s'applique sur un nom commun et un groupe nominal. Une analyse qui nous semble plus naturelle est celle montrée par l'arbre de droite : on crée un nouveau nœud *NP* au dessus du déterminant et du nom commun, pour tomber dans le cas standard d'une coordination de *NP*. Cette correction permet de diminuer le nombre de règles nécessaires au transducteur pour les coordinations.

s'il y a lieu. Ainsi, un lexique extrait directement d'un corpus permettra juste de connaître les occurrences de chaque mot présent, alors qu'un lexique venant des arbres de dérivation représentera la grammaire AB du langage. Par défaut, le lexique listera simplement les mots et leurs types, sans faire de différence entre les différentes fonctions du mot, ce qui peut poser problème pour les homographes (le déterminant "la" et le nom commun "la" par exemple).

Cependant plusieurs options sont possibles :

`-n :` Cette option permet de créer un lexique d'étiquettes qui liste tous les types associés à celles-ci.

`-l :` Cette option permet de préciser le POS-tag des mots.

`-g :` Cette option n'affiche que les mots avec leur POS-tag.

On peut voir ci-dessous un extrait du lexique de Paris VII, généré avec l'option `-n`.

```
8620:la:det: - [...] 5:(s\s)/n, 15:np, 21:(s/s)/n,
                     34:s/n, 48:(n\np)/n, 59:(n\n)/n,
                     378:n/n, 7963:np/n
```

Le déterminant “la” et ses 8 types les plus utilisés. Les informations sont les suivantes : en premier le nombre d'occurrences, puis le mot. En fonction de l'origine et des options, on peut ensuite avoir son POS-tag, et enfin les types, par ordre croissant d'utilisation.

3.3.5 Extracteur de grammaire

Pour pouvoir ensuite analyser des phrases (voir section 5), nous avons eu besoin d'extraire des arbres de dérivation une grammaire hors contexte dans une forme plus usuelle qu'un lexique. Les arbres de dérivation représentent les applications successives des règles de dérivation, aussi n'avons nous eu qu'à parcourir les arbres et stocker les règles au fur et à mesure.

Toutes les règles sont formées sur le même principe :

```
(analyse "racine" (sons "fils1" "fils2") occurrence pourcentage)
```

analyse : indique que la structure parenthésée va être une règle. Etant donné que nous utilisons le même parseur pour les règles de transduction, les arbres et la grammaire, nous devons indiquer dans quel cas de figure nous sommes.

sons : indique que les informations suivantes seront les fils.

“racine” : racine et son type.

“fils1” : premier fils ainsi que le type qui lui est associé.

“fils2” : second fils et son type. En fonction de la forme des arbres qui sont donnés en entrée au programme, il n'existe pas systématiquement.

occurrence : l'occurrence de la règle.

pourcentage : le pourcentage lié à une règle est calculé en fonction de toutes les règles partageant la même racine et non la totalité des règles.

Lorsque les arbres juste après transduction sont donnés en entrée, toutes les règles qui lient le POS-tag au mot sont unaires. Pour l'analyse de phrases, nous préférons utiliser des arbres nettoyés de toute étiquette ou mot, de manière à ce qu'il n'y ait plus que les types, c'est-à-dire des squelettes d'arbres, comme montré figure 3.11. Cela nous permet de réduire le nombre de règles (nous passons de 68 287 règles à 4 457) et d'analyser des phrases dont les mots n'apparaissent pas dans les différents corpus. Un exemple de règle extraite de ces squelettes est montré figure 3.12.

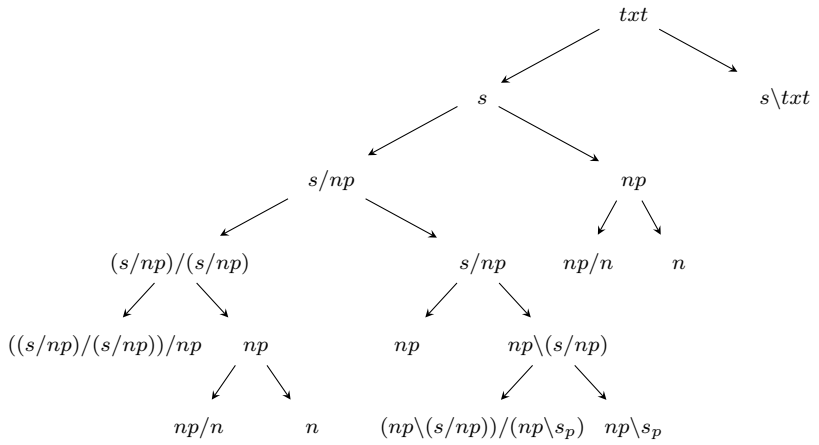


FIGURE 3.11 – Exemple de squelette d'arbre correspondant à la phrase *"A cette époque, on avait dénombré 140 candidats"*.

```
(analyse "NP:n" (sons "NC:n" "VPpart:n\\n" ) 19 0.000264352)

(analyse ":n" (sons ":n" ":n\\n" ) 37109 0.754325)
```

FIGURE 3.12 – Règle gérant un modificateur de nom et son argument, extrait d'une part directement après transduction (en haut) et d'autre part d'un squelette d'arbre (en bas).

3.4 Evaluation

Notre transducteur transforme, en l'état, entre 87% et 94% des corpus que nous lui avons donnés en entrée (voir tableau 3.3), avec un ensemble de 1 675 règles en entrée. Le graphique 3.14 montre l'évolution de l'efficacité du transducteur en fonction du nombre de règles implémentées, sur le corpus de Paris VII. En suivant l'évolution de la courbe, on remarque que pour analyser de plus en plus de phrases du corpus de Paris VII, il faut de plus en plus de règles, presque le double si l'on suit l'évolution de la courbe. En fin de transduction, le transducteur nous donne comme information le nombre de règles qu'il lui a manqué pour traiter l'intégralité du corpus. L'information ne tient pas compte de l'occurrence des règles manquantes. Il faudrait actuellement 997 règles supplémentaires pour arriver à traiter l'intégralité du corpus, et un rapide survol des sous-arbres qui ne sont pas encore traités ne nous a pas fait découvrir de cas factorisables. Un exemple de phrase que nous n'avons pas pu traiter est donné figure 3.13.

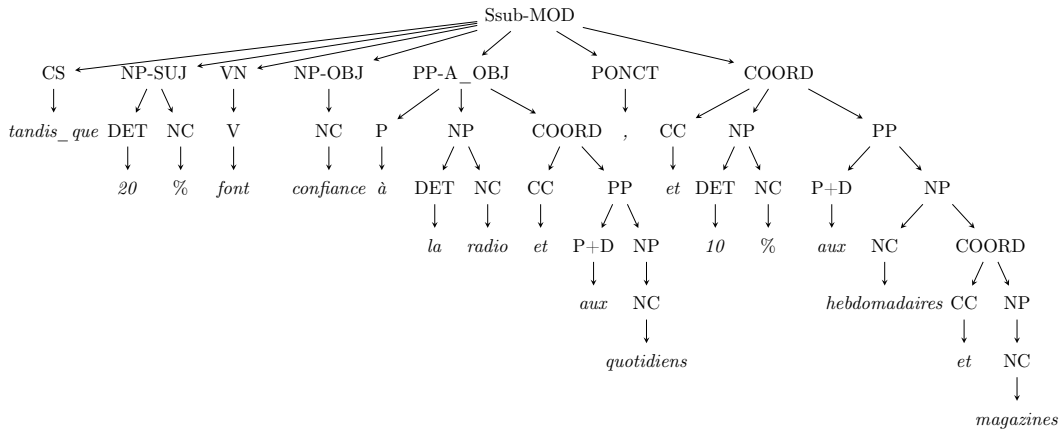


FIGURE 3.13 – Le motif de la coordination, c'est à dire un syntagme prépositionnel et groupe nominal est unique dans le cas d'une ellipse de verbe et de complément d'objet. Nous avons voulu éviter au maximum de faire des règles qui ne s'appliquaient que dans un unique cas.

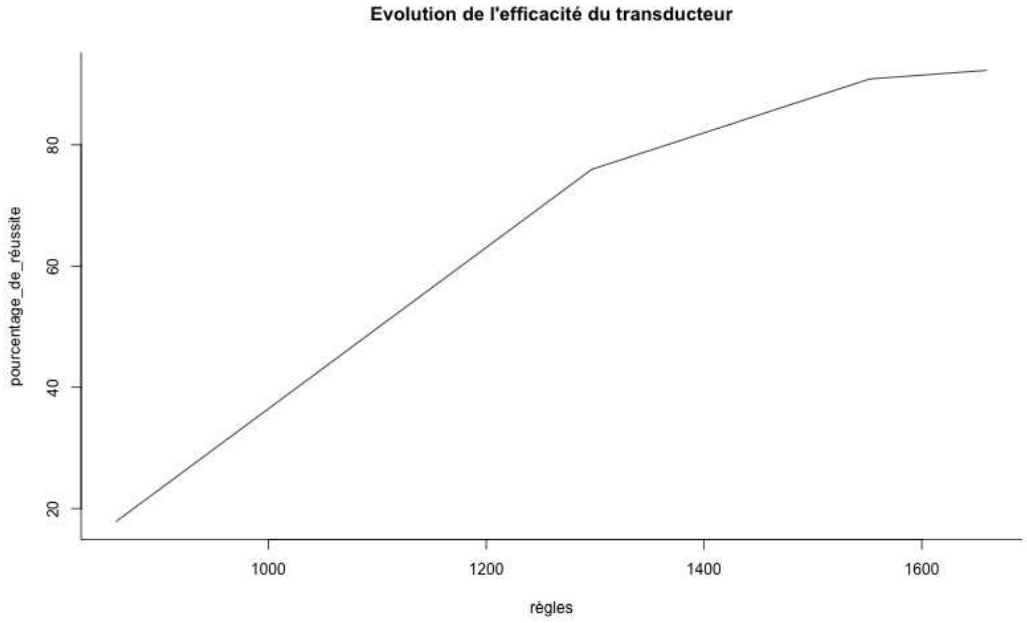


FIGURE 3.14 – Couverture du corpus de Paris VII en fonction du nombre de règles, exprimée en pourcentage.

Nous avons lancé le transducteur sur trois corpus : Séquoia, Paris VII et les 500 phrases de Paris VII qui n'étaient pas représentées sous forme parenthésée

et que nous avons converties nous-même. Le tableau 3.3 récapitule les résultats sur les trois corpus. Nous considérons le temps (proportionnel au nombre de nœuds à traiter dans le corpus) pris pour transformer les arbres, le pourcentage de réussite en nombre de phrases et le ratio du lexique couvert par les arbres de dérivation.

Le lexique généré est ambigu : de nombreux mots fréquents ont plus d'une centaine de formules assignées. Par exemple, dans le lexique de Paris VII, la conjonction de coordination "et" apparaît 4 720 fois et a 219 types différents. Le pronom "lui" a, pour sa part, plusieurs entrées en fonction de son POS-tag (*CLO-A_OBJ*, *CLO*, *PRO*). Avec le rôle de *CLO-A_OBJ*, il apparaît 125 fois et a 33 types différents, avec *PRO* il apparaît 66 fois et a 6 types différents et avec *CLO* il n'apparaît qu'une seule fois.

Les phrases qui ne sont pas analysées contiennent des structures trop rares pour qu'il faille faire une règle particulière pour chaque cas de figure. Nous nous sommes focalisés sur la création de règles de transduction qui soient les plus généralistes possibles. Le tableau 3.4 résume l'utilisation des règles.

Un exemple de sortie du transducteur est montré figure 3.15.

	Séquoia	Paris VII	Paris VII sup
Phrases transformées	3 007 (94 %)	11 397 (92,3 %)	504 (87,9 %)
Lexique couvert	122 254 (88,3 %)	565 582 (83,3%)	20 212 (77,3%)
Temps d'exécution en secondes par phrase	0,069	0,065	0,049
Temps total en minutes	2"25	12"20	0"25

TABLE 3.3 – Récapitulatif des résultats du transducteur. Le temps d'exécution est en minutes et a été calculé avec la fonction `time` de linux. Les lexiques ont été calculés en tenant compte des POS-tags des mots.

Occurrence	nombre de règles	utilisations totales
≤ 5	752	1 620
> 5 et ≤ 100	667	17 742
> 100 et ≤ 1000	182	58 358
> 1000	46	188 108

TABLE 3.4 – Récapitulatif de l'utilisation des règles.

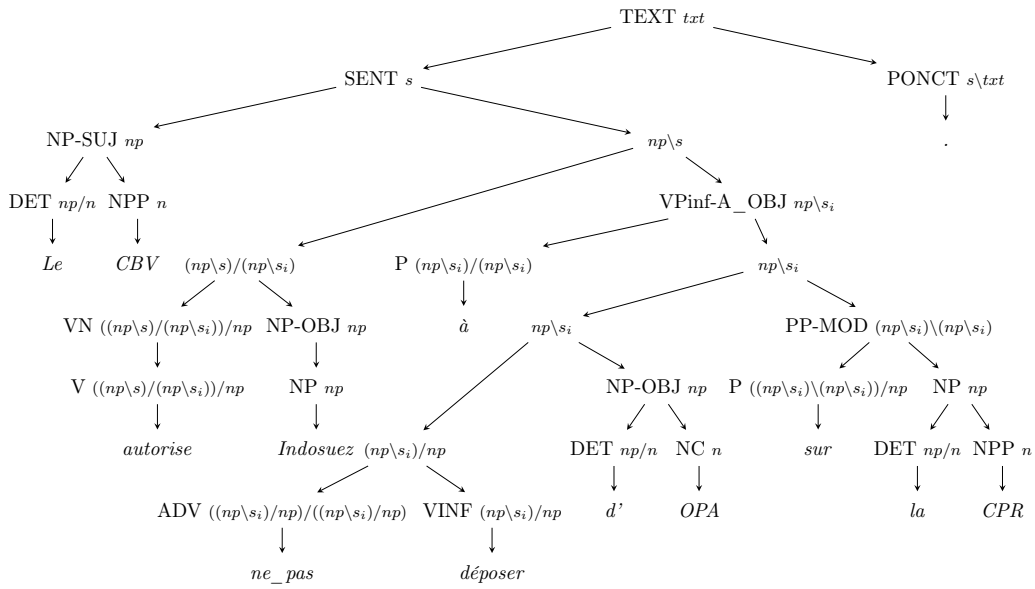


FIGURE 3.15 – Sortie du transducteur. Les informations provenant de l'arbre syntaxique sont toujours présentes. Il est à noter que sur la sortie réelle, les types sont aussi présents dans les feuilles ; ils sont hérités des nœuds préterminaux.

Lorsque nous avons implémenté le transducteur, le gain de temps n'était clairement pas notre objectif principal. Une amélioration possible pour cet état de fait serait de paralléliser le programme, chaque *thread* s'occupant d'un arbre au lieu de traiter les arbres les uns après les autres.

3.5 Perspectives

La généralisation des transducteurs descendants que nous avons introduite montre que cet outil peut être utilisé pour l'extraction automatique d'une grammaire.

Bien qu'il soit difficile d'améliorer le transducteur actuellement, étant donné le nombre de règles que cela nous demanderait de générer pour analyser un petit nombre de phrases, nous avons plusieurs pistes de travail. Il serait intéressant d'utiliser des types plus détaillés, en utilisant cette fois non pas les annotations simplifiées du corpus en forme parenthésée mais les annotations XML, qui différencient les formes des verbes, ainsi que le genre et le nombre des mots. Une autre évolution serait d'utiliser des transducteurs "en cascade" pour simplifier modifier les arbres syntaxiques et avoir ensuite moins de cas à gérer,

donc utiliser moins de règles de transduction. Par exemple, une transduction possible serait d'extraire les clitiques sujets du noyau verbal et les placer sous un nœud *NP-SUJ*. Ainsi, le cas où le sujet est un clitique et celui où le sujet est un groupe nominal sont traités exactement de la même manière, sans avoir à écrire des règles supplémentaires.

Une possibilité d'évolution de ce travail serait de faire évoluer la grammaire vers des incarnations plus modernes des grammaires catégorielles. Tout d'abord, il faudrait passer au calcul de grammaires de Lambek non associatives [Lambek, 1961]. On sait d'après Aarts et Trautwein [1995] que la forme d'arbre de dérivation est adaptée à ces grammaires. A partir des grammaires non associatives, il est possible de passer aux grammaires multimodales [Moortgat, 1997; Moot, 2010b, 2013], qui présentent l'avantage, entre autre, de gérer élégamment les phénomènes d'extraction.

Il faudrait cependant faire un long travail pour passer des grammaires AB aux grammaires de Lambek, car nous n'utilisons pas de règles d'introduction actuellement. Les informations nécessaires, telles que les traces, ne sont pas présentes dans le corpus et nécessiteraient d'être ajoutées d'une manière ou d'une autre, certainement avec une étape manuelle. De plus, les transducteurs d'arbres-vers-arbres ne conviennent plus dans ces cas là, car nous avons besoin de structures plus riches permettant la coindexation entre hypothèses et règles d'introduction. Les démonstrations peuvent alors être vues comme des graphes (voir [Roorda, 1991; Lamarche et Retoré, 1996; Moot et Puite, 2002]) et il nous faudrait utiliser des transducteurs d'arbres-vers-graphes, tels que définis par Engelfriet et Vogler [1994].

Chapitre 4

Inférence grammaticale sur corpus via clustering avec convergence à la Gold

La méthode d'inférence grammaticale présentée dans ce chapitre fait partie des méthodes prenant en entrée des structures partiellement définies, c'est à dire des arbres. Nous avons souhaité rester dans la lignée de Buszkowski et Penn [1990] ou Kanazawa [1998], tout en améliorant la complexité de l'étape d'unification des variables en appliquant un algorithme de clustering. Les inconvénients des méthodes précitées ont été résumés dans le chapitre 2, et notre méthode permet une baisse de la complexité, qui passe d'exponentielle à polynomiale pour l'unification. Il y a cependant une différence importante entre notre algorithme et ceux de Buszkowski et Penn et Kanazawa : alors que les grammaires k-valuées se focalisent sur le nombre de types associés à un mot, nous préférons minimiser le nombre de types du lexique, en unifiant les types des mots présents dans un même contexte syntaxique.

Nous avons souhaité que notre méthode reste dans l'esprit d'un apprentissage à la limite de Gold [1967]. Si les mots ou leur futur type sont déjà dans la grammaire et permettent l'analyse de la phrase, le lexique final n'aura pas de types nouvellement ajoutés. Sinon ils sont inclus dans le cluster et une unification est possible. Nous sommes donc dans un contexte semblable à celui défini par Gold et, dans la section 4.4, nous démontrons la convergence des clusters correspondant aux arbres.

Le résultat de l'unification est une grammaire AB. Ce choix a été motivé par l'envie de comparer les résultats avec ceux du transducteur, tant au niveau des lexiques que de l'efficacité des grammaires (voir chapitre 5). De plus, initialement nous avions créé le transducteur pour binariser intelligemment les arbres du corpus de Paris VII. Il nous semblait donc être dans la suite logique d'utiliser non plus la grammaire extraite directement après transduction mais

les arbres de dérivation pour un algorithme semblable à celui de Buszkowski et Penn ou Kanazawa.

Le passage aux grammaires de Lambek ou aux CCG a été exclu car les structures de données ne comprenaient pas assez d'information pour savoir, par exemple, quand utiliser une règle d'introduction.

Ces travaux ont fait l'objet de trois publications [Sandillon-Rezer, 2013b,c,a].

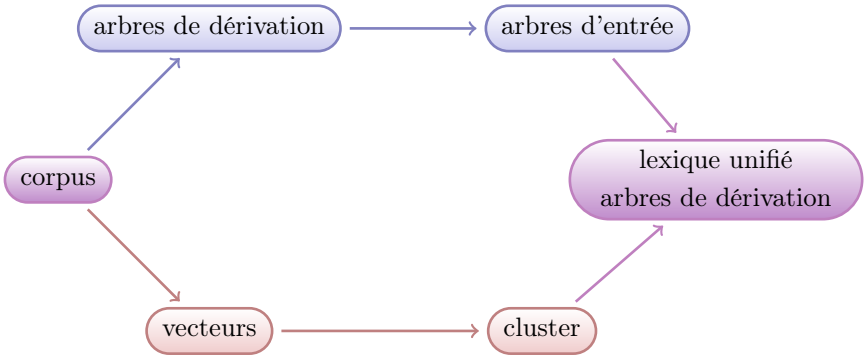


FIGURE 4.1 – Mise en place et utilisation des différents éléments permettant l'unification. Le chemin bleu représente les étapes effectuées pour obtenir les arbres d'entrée de notre algorithme, celui en rouge celles nécessaires à l'obtention du cluster et la réunion des deux, en violet, correspond à l'algorithme d'unification en lui-même.

La figure 4.1 schématise le fonctionnement de notre méthode et montre le découpage de notre chapitre. Dans un premier temps, nous nous focaliserons sur la manière dont nous obtenons les arbres d'entrée (section 4.1), puis dans un second temps sur l'extraction des vecteurs et le clustering (section 4.2). Enfin, nous expliquerons notre algorithme d'unification (section 4.3). La preuve de convergence sera donnée section 4.4. Quelques détails sur l'implémentation seront donnés en section 4.5 avant l'évaluation de notre méthode section 4.6.

4.1 Arbres d'entrée

Les algorithmes de Buszkowski et Penn et Kanazawa demandent en entrée des arbres binaires dont la racine est étiquetée s et les nœuds internes / ou \, de manière à savoir s'il faut appliquer une règle d'élimination à droite ou à gauche. Nous générions depuis longtemps, via un simple programme de transformation, de tels arbres mais avons décidé de ne pas les prendre en entrée de notre algorithme : nous avons privilégié des arbres contenant plus d'informations.

Certains types sont fixés, ce qui nous donne en entrée des arbres de dérivation avec partiellement des variables et partiellement des types.

Comme montré sur le schéma 4.1, la binarisation des arbres est effectuée par le transducteur, puis les arbres de dérivation sont passés à un petit programme qui va automatiquement mettre les types à jour et effacer les étiquettes qui demeurent sur certains nœuds. L'algorithme de transformation des arbres est très simple : en premier lieu, on fixe la racine, soit à *txt* soit à s. Ensuite on parcourt l'arbre en profondeur et à chaque nœud le traitement portera sur ses deux fils :

1. On vérifie qui est l'argument et qui est le foncteur.
2. On regarde si l'étiquette de l'argument est dans le tableau 4.1 :
   – si l'étiquette y est, le type est donné à l'argument,
   – sinon l'argument est typé par une variable.
3. le foncteur obtient la catégorie *type_du_père* / *type_de_l'argument* ou *type_de_l'argument* \ *type_du_père*, en fonction de la règle d'élimination utilisée.

Nous n'avons pas décidé de remplir le tableau 4.1 au hasard : nous nous sommes fondés sur le lexique d'étiquettes créé à partir des arbres de dérivation du corpus de Paris VII et avons gardé le type le plus utilisé si son occurrence est supérieure à 70%.

étiquette	**type**	**étiquette**	**type**	**étiquette**	**type**
TEXT	*txt*	*SENT*	s	*NP*	spécial
NP-SUJ	np	*NP-OBJ*	np	*NP-A_OBJ*	np
NP-DE_OBJ	np	*NP-ATS*	np	*NP-ATO*	np
PP	pp	*PP-A_OBJ*	pp_a	*PP-DE_OBJ*	pp_de
PP-P_OBJ	pp	*AP-OBJ*	$n\backslash n$	*AP-ATS*	$n\backslash n$
VPinf-ATS	$np\backslash s_i$	*VPinf-ATO*	$np\backslash s_i$	*VPinf-OBJ*	$np\backslash s_i$
VPinf-DE_OBJ	$np\backslash s_i$	*VPinf*	$np\backslash s_i$	*VPinf-A_OBJ*	$np\backslash s_i$
VPinf-P_OBJ	$np\backslash s_i$	*VPpart-ATS*	$np\backslash s_p$	*VPpart-OBJ*	$np\backslash s_p$
VPpart-DE_OBJ	$np\backslash s_p$	*VPpart*	$np\backslash s_p$	*VPpart-A_OBJ*	$np\backslash s_p$
CLS	np	*CLS-SUJ*	np	*CLS-A_OBJ*	np
VPP	$np\backslash s_p$	*VINF*	$np\backslash s_i$	*NC*	n
CLR	cl_r	*NPP*	np		

Table 4.1 – Liste des types assignés aux nœuds lorsque ceux-ci ont la bonne étiquette, si et seulement s'ils sont arguments et non foncteurs. Les nœuds *NP* ont un traitement spécial : en effet, le type le plus courant est n, mais lorsque le déterminant est encore présent dans le groupe nominal, nous avons souhaité que le type donné soit np. Notre programme vérifie donc le type du nœud lorsque celui-ci est étiqueté *NP* et, si le type associé est np, le laisse.

Les arbres d'entrée contiendront donc aussi bien des variables que des types déjà calculés, et c'est ce sur quoi nous comptons pour réussir à faire une unification automatique qui ne laisse pas ou peu de variables dans l'arbre résultant.

Un exemple d'arbre d'entrée est montré figure 4.2. On note que des types composés, comme $((np\backslash s)/(np\backslash s_i))/np$, sont déjà présents dans l'arbre d'entrée.

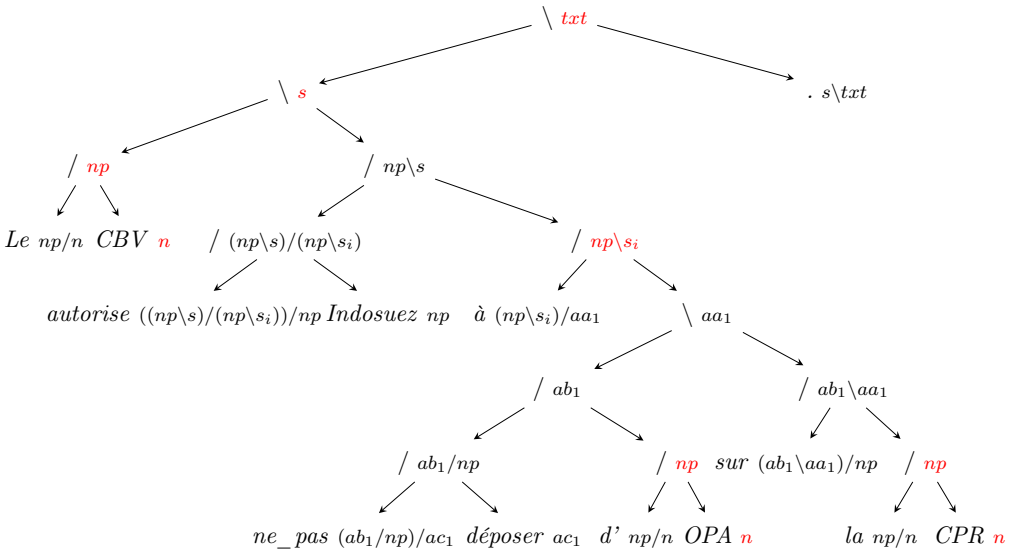


FIGURE 4.2 – Arbre d'entrée pour notre algorithme d'inférence grammaticale. Certains types, en rouge, sont déjà présents et les autres sont remplacés par des variables.

4.2 Etape de clustering

Nous effectuons un clustering sur les mots des arbres syntaxiques, de manière à avoir des informations sur le contexte du mot.

Les clusters que nous utilisons sont des clusters hiérarchiques. Le terme cluster est très général, et utilisé aussi bien en musique pour parler d'un ensemble de notes espacées du même temps, qu'en réseau au sujet de "grappes" d'ordinateurs, mais aussi en astronomie pour décrire un amas d'étoiles, etc. En analyse de données, les clusters décrivent la manière dont on partitionne des éléments en fonction de leurs caractéristiques. C'est cette définition de cluster que nous allons utiliser. L'aspect hiérarchique du clustering que nous utilisons se traduit par le fait que les éléments les plus semblables sont rangés de manière à être les plus proches les uns des autres.

Plus formellement, le clustering hiérarchique peut être défini comme il suit :

Définition 3. *Un clustering hiérarchique est composé d'un tuple* $< E, D_m, C >$, *avec :*

E **:** *Un ensemble de vecteurs représentant les caractéristiques des données.*

D_m **:** *Une fonction qui permettra de calculer la distance métrique entre les vecteurs.*

C **:** *Une fonction de calcul de cluster qui, en fonction du résultat de la distance métrique, ordonnera les éléments du cluster.*

L'étape d'extraction des vecteurs explique justement quelles informations nous avons jugé bon de garder.

4.2.1 Extraction de vecteurs

L'extraction des vecteurs s'effectue en deux étapes : dans un premier temps on récupère les informations du corpus, puis celles-ci sont transformées en vecteurs numériques de manière à être utilisées pour le clustering. Ce qui a motivé le passage vers des vecteurs numériques était la nécessité de comparer de manière efficace les étiquettes : comment dire qu'un *NC* est plus ou moins éloigné d'un *ADJ* ?

La première phase, cependant, a demandé de choisir le nombre de dimensions que nous voulions dans nos vecteurs, c'est à dire le nombre d'informations qui nous semblaient importantes à garder. Ainsi, nous avons voulu garder des informations liées aux frères à droite et à gauche du nœud duquel on extrait le vecteur, car en français ceux-ci ont généralement une influence forte sur les mots. Pour un nom commun, par exemple, il y a à sa gauche son déterminant dans 82% des cas (sur le corpus de Paris VII), et un verbe sera suivi, dans 20% des cas environ, d'un participe passé. Dans ces deux exemples, le voisin fait partie du même sous-arbre et influe directement sur le nœud en question, que ce soit en étant son foncteur ou son argument. Nous avons voulu pondérer l'influence de ces voisins en notant la profondeur de liaison de ceux-ci avec notre mot. Ceci est représenté par la distance jusqu'au plus proche ancêtre commun. La distance minimale est deux (c'est à dire qu'à une distance de un, cela signifierait que les mots ont le même nœud en guise de POS-tag, ce qui est impossible) et peut grimper jusqu'à la hauteur maximale de l'arbre [1], si les deux mots n'ont en commun que la racine, étiquetée *SENT*.

Dans la version actuelle, nous avons des vecteurs à six dimensions :

1 POS-tag du mot (père),

2 information morpho-syntaxique (grand-père) [Johnshon, 1998],

3-4 POS-tag des frères à gauche et à droite,

1. La hauteur maximale est la distance séparant les deux vecteurs les plus éloignés.

5-6 distance jusqu'au plus proche ancêtre commun avec le voisin de gauche et de droite.

Initialement, nos vecteurs ne comprenaient que les quatre premières dimensions, mais nous nous sommes rendus compte que dans des cas assez fréquents, les voisins de droite et gauche n'étaient pas dans le même sous-arbre. Il nous semblait important que ceux-ci aient, par conséquent, une influence différente sur le mot que s'ils faisaient partie du même sous-arbre. Le sous-arbre montré figure 4.3 résume les informations que nous prenons pour créer les vecteurs.

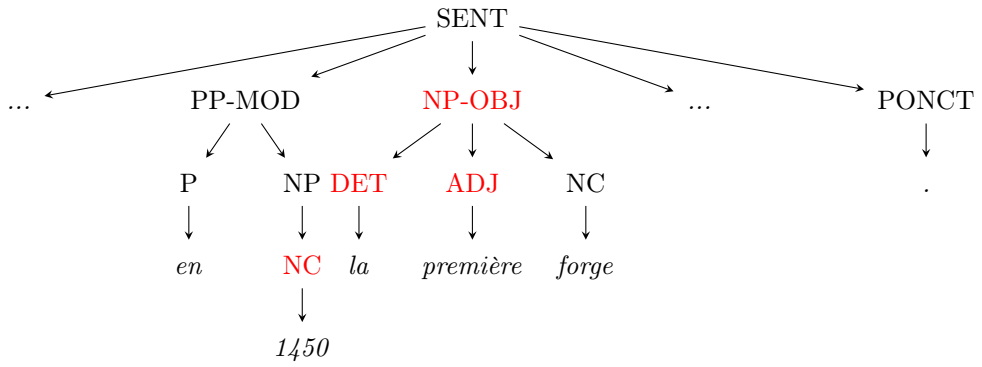


FIGURE 4.3 – Les informations qui seront comptées dans le lexique sont notées en rouge.

S'il n'y a pas de voisin de droite ou de gauche (dernier ou premier mot d'une phrase), la valeur correspondant à la coordonnée de ce vecteur sera instanciée à *NIL* ou -5, suivant si c'est une étiquette ou un nombre. Le vecteur correspondant au déterminant *"la"* présent dans l'arbre 4.3 est donné figure 4.4.

$$la < \text{DET, NP-OBJ, NC, ADJ}, 4, 2 >$$

FIGURE 4.4 – Vecteur correspondant au déterminant "la" de la figure 4.3.

Pour comparer les vecteurs, nous avons eu besoin de les transformer en vecteurs dans $\mathbb{Z}^n, n \in \mathbb{N}^*$. Nous avons pris le parti de transformer chaque étiquette en vecteur où seulement une ou deux dimensions possèdent la valeur 1 et le reste des coordonnées a pour valeur 0. Les POS-tags et les informations syntaxiques sont transformés de cette manière. Les distances numériques restent telles quelles, comme montré figure 4.4. La transformation est illustrée par la table 4.2 avec seulement une portion des données.

Il y a une "dimension" pour presque chacun des POS-tags (avec cependant quelques exceptions pour des cas que nous souhaitons voir unifiés, comme *ET*

pour les mots étrangers et *NPP* pour les noms propres) ; pour les informations morpho-syntaxiques, en plus d'une dimension pour chaque catégorie de base (*NP*, *PP*...) on fait seulement la différence entre les arguments (représentés par le *-SUJ, -OBJ, -ATS...* à la fin des étiquettes) et les modificateurs *-MOD*. Vu le nombre de POS-tags et d'annotations morpho-syntaxiques différentes, nous avons besoin de 14 "dimensions" par étiquette.

POS-tag	**NC**	**DET**	**P**	**...**
NC	1	0	0	0...0
DET	0	1	0	0...0
P+D	0	1	1	0...0
Other	**NP**	**...**	**-ARG**	**-MOD**
NP	1	0...0		
NP-SUJ	1	0...0	1	0
NP-MOD	1	0...0	0	1

TABLE 4.2 – Exemple de transformation de vecteurs.

Une fois les vecteurs composés uniquement de valeurs numériques, on peut passer à l'étape de clustering.

4.2.2 Clustering

Nous avons pris le parti de calculer des clusters hiérarchiques. Ainsi, les mots apparaissant dans des contextes similaires seront proches les uns des autres.

Un cluster peut aussi bien regrouper un ensemble de clusters que des mots. Chaque cluster est associé à une hauteur qui représente la similarité entre les données qu'il regroupe. Ainsi, les clusters de hauteur zéro regroupent les mots dont les vecteurs sont identiques, et les clusters de hauteur plus importante regroupent aussi bien des mots que d'autres clusters qui leur sont proches.

Etant donné que nous ne comptions pas implémenter notre propre algorithme de clustering, nous avons décidé d'utiliser la méthode de variance minimum de Ward [1963], déjà implémentée dans le logiciel R [Ihaka et Gentleman, 1993]. Pour le calcul de la distance métrique, nous avons choisi Manhattan. Avant de nous fixer sur Manhattan, nous avons testé les différentes métriques proposées par R : Euclidienne ($\sqrt{\sum |x_i - y_i|^2}$), Maximum ($max(|x_i - y_i|)$), Canberra ($\sum \frac{|x_i - y_i|}{|x_i + y_i|}$) et Minkowski ($\sqrt[p]{\sum |x_i - y_i|^p}$). Sur un échantillon d'une dizaine de phrases, les résultats des clusters étaient plus satisfaisants avec Manhattan, ce qui a motivé notre choix final.

Le cluster ressemble à un arbre, comme on peut en voir un extrait sur la figure 4.6. Cependant, vu la taille des données, cela ne permet pas de les visualiser en totalité. Une visualisation d'un cluster complet peut être obtenue

avec le logiciel Tulip [Auber et Mary, 2007], ce qui permet de créer des graphes comme ceux de la figure 4.5.

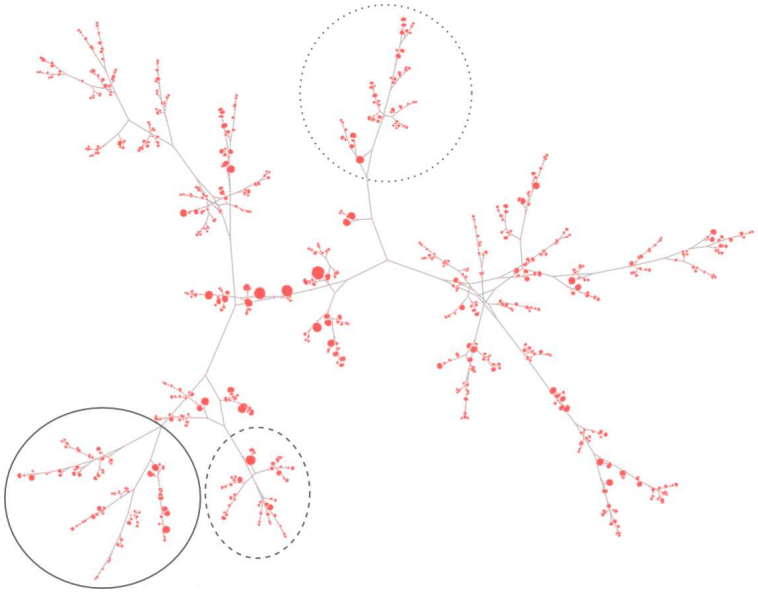


FIGURE 4.5 – Cluster correspondant à notre ensemble test de phrases. La partie entourée en pointillés correspond à l'endroit où il y a le plus de verbes ; celle entourée par des tirets aux déterminants et la partie en trait plein correspond aux adjectifs. On peut noter que les adjectifs et les déterminants sont proches les uns des autres. Cela s'explique parce qu'ils prennent généralement tous les deux un nom commun en argument et qu'ils sont présents dans des groupes nominaux.

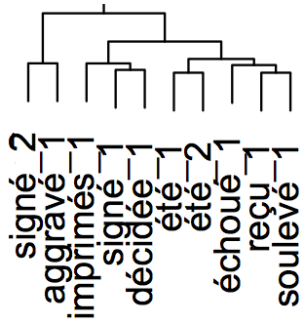


FIGURE 4.6 – Extrait d'un cluster de 5 phrases. Les participes passés sont regroupés d'abord par contexte puis tous ensembles dans de plus larges clusters jusqu'à n'en former plus qu'un.

Nous nous focaliserons sur des détails du cluster présenté figure 4.5 dans la section 4.3, de manière à montrer quand l'unification a lieu.

4.3 Unification

Le cluster aussi bien que les arbres binaires contenant à la fois des types et des variables sont donnés en entrée.

Nous unifions les clusters par hauteur croissante, ce qui nous permet d'unifier par ordre de similarité. Notre algorithme agit dans un premier temps en traitant les clusters paire par paire, soit lorsque la hauteur de deux clusters est identique, soit lorsqu'elle est très proche. La première étape consiste à fusionner les deux clusters. Pour ce faire, on parcourt tous les types du second cluster et il y a deux cas de figure possibles :

1. Le même type est déjà présent dans le premier, on se contente de recopier les mots associés au type du second cluster vers le premier.
2. Un type du second cluster n'est pas dans le premier cluster. On cherche alors s'il existe une possibilité d'assigner les variables entre elles et on garde le type de côté.

Lorsque l'on a fini de traiter tous les groupes de clusters de même hauteur, on se focalise sur la liste de types mis de côté. S'il n'y a qu'une possibilité, les variables sont unifiées. Sinon on peut soit unifier totalement au hasard, soit utiliser ces heuristiques par ordre de préférence :

1. unifier les types venant des plus petits clusters,
2. unifier avec le plus simple candidat (la complexité d'un candidat est calculée en fonction du nombre de \ et de / qu'il contient),
3. choisir le premier venu pour les autres variables, avec la possibilité de choisir aléatoirement l'unification.

Malheureusement, après la 3^{e} étape, une variable peut avoir plus d'une unification possible, et nous n'avons pas encore réussi à définir quelle était la meilleure. Dans ce cas, nous utilisons un système arbitraire : la méthode du premier venu a l'avantage de toujours donner le même résultat mais la méthode d'unification aléatoire donne des résultats légèrement meilleurs , bien qu'ils ne soient jamais les mêmes.

Il se peut que tous les mots ne soient pas représentés à un niveau donné (cela peut arriver lorsqu'un mot est isolé à hauteur zéro et ne rejoint un cluster que longtemps après), par conséquent il peut rester des variables dans les types après une étape de clustering. Dans ce cas, on passe à un nouveau niveau de clustering. Cette manière de procéder nous assure l'unification des variables qui apparaissent en premier lieu dans des contextes les plus similaires possibles.

Il est à noter que même avec des variables, les arbres de dérivation partiels restent des dérivations valides et représentatives d'une grammaire AB : les deux règles d'élimination sont les seules utilisées pour créer les arbres.

Pour illustrer l'unification, nous nous sommes focalisés sur deux clusters de niveau zéro (voir figure 4.7). Le cluster de gauche est un cluster d'adverbes,

ne contenant qu'une seule variable. Il n'y aura donc pas d'ambiguité et la variable ab_{331} sera unifiée avec np. Le cluster de droite, cependant, ne contient que des variables, et rassemble des adjectifs et participes passés utilisés en tant qu'adjectifs. Dans ce cas précis, soit un cluster de même niveau permet de désambiguïser les unifications possibles, soit nous avons besoin de choisir une variable (la première qui apparaît ou n'importe laquelle) qui deviendra le type de tous les mots du cluster. Dans tous les cas, les mots auront tous le même type.

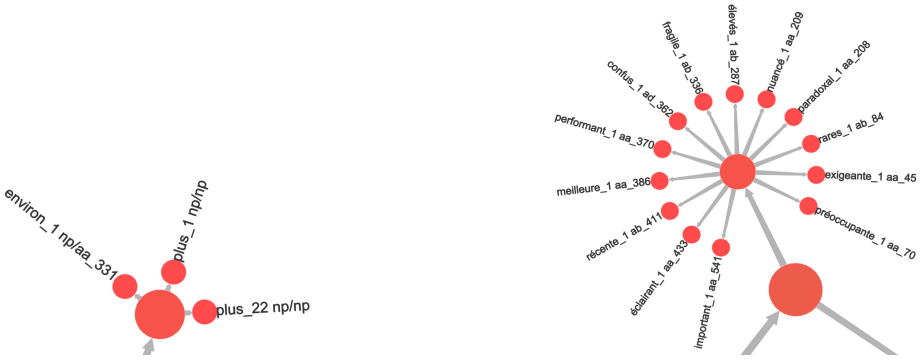


FIGURE 4.7 – Zoom sur deux clusters de niveau zéro.

4.4 Preuve de convergence

4.4.1 Définition de la grammaire de cluster

La preuve de convergence fonctionne non pas sur la grammaire qui donne le lexique final après unification, mais sur la grammaire de clusters. A chaque occurrence d'un mot correspond un cluster. Nous avons donc un algorithme qui, pour chaque mot du corpus, peut calculer son cluster. Les clusters calculés ainsi correspondent à des ensembles de formules qui, par construction (voir section 4.3) s'unifient et sont compatibles avec la structure de preuves d'une grammaire AB (par construction). La grammaire ainsi obtenue n'est pas rigide, étant donné qu'un mot peut avoir plusieurs types, mais la fonction qui associe une formule à un cluster de niveau zéro est rigide (la fonction qui à un mot, étant donné un contexte, associe un unique cluster cache ce point). De plus, certains types sont, dans l'arbre de dérivation, fixés par avance.

Les arbres de dérivation de notre grammaire de cluster sont donc, comme montré figure 4.8, des arbres de dérivation d'une grammaire AB dont les feuilles ont été remplacées par la référence à un cluster ainsi que le type associé à celui-ci.

Nous avons donc la définition d'une grammaire de clusters suivante :

Définition 4 (Grammaire de Clusters)**.** *Une grammaire de Clusters est une grammaire AB dont les terminaux font référence à un cluster. Nous avons donc :*

- *Deux règles d'élimination, / et \, comme pour une grammaire AB usuelle.*
- *Un ensemble de types construits à partir d'un ensemble* $\mathcal{P} = \{txt, s, np, n, np\backslash s_p, np\backslash s_i, pp, pp_a, pp_{de}, cl_r\} \cup O$*, nous avons ensuite l'usuel :*

$$\mathcal{L} := \mathcal{P} \mid \mathcal{L}/\mathcal{L} \mid \mathcal{L}\backslash\mathcal{L}$$

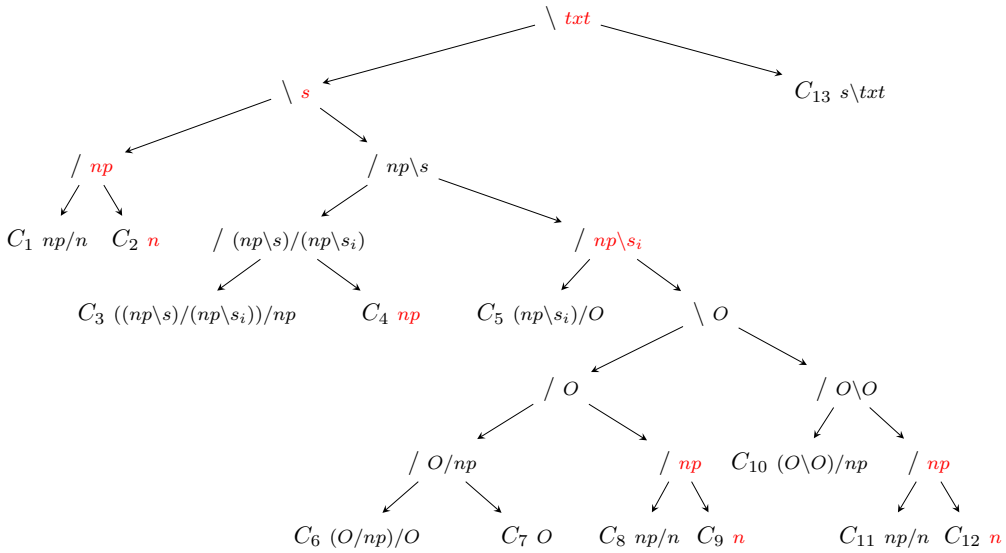


FIGURE 4.8 – Arbre de dérivation correspondant à notre grammaire de cluster, tel que pris en entrée par notre algorithme. Chaque mot a été remplacé par le cluster auquel il appartient. Les types en rouge sont fixés lors de la transformation en arbres de dérivation.

Nous souhaitons prouver que cette grammaire converge, c'est à dire qu'à un moment, le cluster global correspondant à la grammaire du français ne changera plus.

Après séparation des clusters "incohérents" au niveau des mots, c'est à dire séparation des clusters qui ne permettent pas une unification totale des types présents, par construction tous les éléments des clusters s'unifient. L'algorithme fonctionne de manière ascendante. Dans l'idéal, donc, les clusters représentent, surtout au niveau zéro, les mots qui sont dans le même contexte. On souhaiterait donc que les clusters ne contiennent que des variables, ou que des constantes identiques (des clusters de nom communs, par exemple) ou des variables et des constantes qui peuvent s'unifier sans problème, ce qui correspondrait au concept de grammaire rigide de cluster (on associe un seul type final par cluster).

Nous pouvons utiliser un oracle pour connaître exactement quel cluster est le bon pour un mot donné. Nous pouvons aussi séparer les clusters en utilisant notre algorithme d'unification, qui traite tous les cas de figures possible. Il est important de noter que, dans notre implémentation, la division des clusters en sous-clusters compatibles pour l'unification se fait déjà en fonction de la forme et de la compatibilité des types présents dedans. Même s'il n'y a pas de division de clusters en sous-clusters à proprement parler, nous unifions uniquement les types compatibles entre eux. Plus explicitement, voilà ce qui est effectué :

Nous différencierons les types composés d'une ou plusieurs constantes (tel que np, $np\backslash s_p$ ou $(np\backslash s)/np$), ceux composés uniquement de variables (aa_1, ab_1/ac_1, etc.) et ceux "mixtes", c'est à dire composés de constantes et de variables (aa_1/n par exemple). Nous appellerons :

Type-constant : un type composé uniquement de constantes.

Type-variable : un type composé uniquement de variables.

Type-mixte : un type composé de variables et de constantes.

Le cluster contient un type-constant et plusieurs types-variables : si le type-constant est compatible avec les types-variables, on laisse le cluster tel quel. Un type-constant, quel qu'il soit, est compatible avec des types-variables si :

- Les types-variables ne contiennent qu'une seule variable, comme aa_1, ab_2,... .
- Les types-variables ont la même forme que le type-constant (aa_1/ab_1 dans un cluster ou le seul type présent est n/n).
  Si certains types-variables ne sont pas compatibles avec le type-constant, on crée autant de nouveaux clusters que de formes de types-variables incompatibles (par exemple, $aa_1\backslash ab_1$ dans un cluster où le seul type présent est n/n).

Le cluster contient plusieurs types-constants et plusieurs types-variables : on crée un cluster pour chaque ensemble type-constant/type-variable compatibles. Bien que les types-variables non composés puissent aller avec n'importe quel type-constant, on choisit de les mettre avec le candidat le plus simple possible, pour réduire la taille finale de la grammaire.

Le cluster ne contient que des types-constants : on crée autant de clusters que de types différents.

Le cluster ne contient que des types-variables : si les variables sont unifiables entre elles (la même forme si ce sont des variables complexes ou des variables simples) on laisse le cluster tel quel, sinon on crée autant de clusters que de formes de variables présentes.

Les types-mixtes, eux, sont gérés comme les types-variables, en ajoutant la contrainte que s'ils sont dans un cluster comprenant un ou plusieurs type-

constant, ils doivent être partiellement compatibles (on considère comme compatible np/n et aa_1/n ou $(np\backslash s)/n$ et aa_1/n, et comme incompatible np/n et $(np\backslash s)/aa_1$).

Un soucis se crée lorsqu'un cluster est composé de deux types-constants et plusieurs types-variables, mais que nous n'avons pas de moyen de séparer les deux types-constants (par exemple n et np). Il est évident que l'on peut séparer les types-constants mais si les types-variables sont compatibles avec les deux, nous n'avons pas de "bon" moyen pour savoir comment les disperser. On note que c'est le même problème que l'on retrouve pour les grammaires k-valuées.

Les nouveaux clusters ainsi créés seront intégrés très proches de leur ancien cluster : la figure 4.9 montre la manière dont un cluster, noté C_2 est divisé en deux clusters C_2 et C_2', ainsi que la manière dont C_2' est attaché par rapport au reste du cluster.

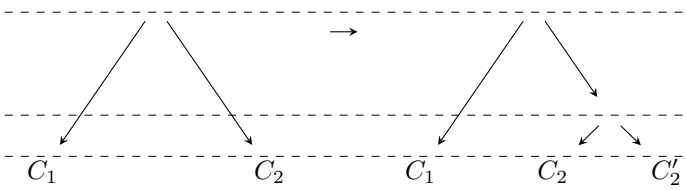


FIGURE 4.9 – Les deux nouveaux clusters, C_2 et C_2', sont proches au niveau hauteur, et tous les deux sont toujours liés à C_1 à la même hauteur qu'avant.

On peut se poser la question de l'influence de la duplication d'une phrase un grand nombre de fois. Cependant, comme un cluster peut aussi bien contenir un unique mot qu'un ensemble de mot, cela ne change rien à la forme générale du cluster, ni même à l'algorithme d'unification que nous utilisons.

4.4.2 Preuve de convergence

Nous suivons la preuve de Bonato [2006] (qui s'inspire de la preuve de Kanazawa [1998]) et réutilisée dans [Moot et Retoré, 2012; Retoré et Bonato, 2013] et nous allons nous pencher sur les grammaires rigides apprises avec notre algorithme.

Nous utiliserons les notations suivantes :

$G \subset G'$ cette relation réflexive entre G et G' se définit lorsque pour tout assignement lexical $a : T$ dans G, avec a un cluster et T un type ou un ensemble type/variable compatibles, on retrouve un assignement similaire, à une substitution près, dans G'. On remarque que c'est équivalent à la relation $\subset$ classique entre les lexiques : $lex_G(a) \subset lex_{G'}(a)$.

taille de la grammaire c'est la somme de toutes les occurrences des catégories simples dans le lexique. Ainsi, l'élément : $le : np/n$ comptera pour 2 occurrences. Les variables ne comptent pas dans la taille de la grammaire.

σ une substitution est une fonction allant d'un ensemble de catégories à un autre, qui est générée par une association σ_V de variables dans $\mathcal{V}$, V étant l'ensemble infini de variables et types [2]. Ainsi, en plus de fixer la substitution $\sigma(s) = s$, pour le type de la racine, tout ce qui n'est pas une variable dans les arbres d'origines ne pourra pas être modifié par σ.

$G \sqsubset G'$ cette relation réflexive entre G et G' se définit lorsqu'il existe une substitution σ telle que $\sigma(G) \subset G'$, ce qui ne différencie pas les différentes catégories d'un mot.

$CL(G)$ on dénote par ceci l'ensemble des structures (appelées par la suite CL-structures) produites par G.

$GF(D)$ étant donné un ensemble de D d'exemples structurés, la grammaire $GF(D)$ est définie en rassemblant les catégories de chaque mot dans les différents exemples de D. On différencie, au niveau des entrées du lexique, les mots en fonction de leur cluster de niveau 0.

$RG(D)$ étant donné un ensemble de CL-structures D, $RG(D)$ est le lexique obtenu en appliquant l'unificateur le plus général, lorsqu'il existe, à $GF(D)$. Le type O sera alors remplacé par des variables, comme montré section 4.1, ce qui donne le typage le plus général. Une grammaire est considérée comme rigide si, après unification des variables, chaque cluster n'a qu'un seul type.

Proposition 1 *Si :*
- *σ est une substitution,*
- *π une CL-structure générée par une grammaire G,*

alors
- *$\sigma(\pi)$ est généré par $\sigma(G)$,*
- *π et $\sigma(\pi)$ ont la même CL-structure associée.*

Preuve. Deux grammaires, G_1 et G_2, avec leur catégories dans un ensemble $\mathcal{V}$, sont dites égales lorsqu'il existe une fonction de renommage ν telle que $\nu(G_1) = G_2$. Une substitution σ unifie deux catégories A et B lorsque $\sigma(A) = \sigma(B)$. Une substitution σ unifie un ensemble de catégories T (une entrée lexicale) lorsque pour tout couple $\{A, B\} \in T$ on a $\sigma(A) = \sigma(B)$, c'est à dire qu'on peut trouver une substitution σ telle que toutes les catégories de T puissent être substituées par une unique.
Une substitution unifie une grammaire AB G si, pour tout cluster du lexique de G, σ unifie $lex_G(w)$, c'est à dire que tous les clusters du lexique n'ont qu'une

2. Ce sont les catégories que l'on donne aux nœuds avec notre étape préliminaire.

seule catégorie. La définition correspond à trouver l'unificateur qui permet d'avoir une grammaire rigide.
Une notion qui nous servira plus tard est la notion d'Unificateur le plus général (noté *mgu*, pour *most general unifier*) : il n'y a pas toujours une unification, mais lorsque celle-ci existe, on peut définir un *mgu*, σ_u tel que pour tout unificateur τ, il existe une substitution σ_τ telle que $\tau = \sigma_\tau \circ \sigma_u$. Ce *mgu* est unique à renommage près.

Proposition 2 *Si π_v est l'étiquetage le plus général pour une CL-structure π, alors il existe une substitution telle que $\pi = \sigma(\pi_v)$.*

Proposition 3 *Etant donné une grammaire G, le nombre de grammaires H telles que $H \sqsubset G$ est fini.*

Preuve. Etant donné que G est un ensemble fini d'assignations cluster-type, il y a un nombre fini de grammaires incluses dans G. Lorsque $\sigma(H) = K$ pour une substitution σ donnée, la taille de H est plus petite ou égale que celle de K (par définition de la substitution vu plus haut), et à un renommage près, il y a un nombre fini de grammaires plus petites qu'une grammaire fixée.
Par définition, si $H \sqsubset G$ alors il existe $K \subset G$ et une substitution σ telle que $\sigma(H) = K$. Etant donné qu'il n'y a qu'un nombre fini de K tel que $K \subset G$, et que pour tout K il y a un nombre fini de H pour lesquels il peut exister une substitution σ avec $\sigma(H) = K$ (les substitutions augmentent la taille des catégories) on conclut qu'à un renommage près il n'y a qu'un nombre fini de H tel que $H \sqsubset G$. □

Proposition 4 *Si $G \sqsubset G'$ alors $CL(G) \subset CL(G')$.*

Proposition 5 *Si $GF(D) \sqsubset G$ alors $D \subset CL(G)$.*

Preuve. Par construction de $GF(D)$, on a $D \subset CL(GF(D))$. De plus, d'après la proposition 4, on a $CL(GF(D)) \subset CL(G)$. □

Proposition 6 *Si $RG(D)$ existe alors $D \subset CL(RG(D))$.*

Preuve. Par définition, $RG(D) = \sigma_u(GF(D))$, où σ_u est la fonction d'unification utilisée par notre algorithme. Nous avons alors $GF(D) \sqsubset RG(D)$, et d'après la proposition 5, avec $G = RG(D)$, on obtient $D \subset CL(RG(D))$. □

Proposition 7 *Si $D \subset CL(G)$ alors $GF(D) \sqsubset G$.*

Preuve. Par construction, chaque type x, composé soit d'une variable, soit d'un ensemble de catégories et de variables, soit d'une catégorie, étiquette au plus une feuille d'au plus une CL-structure de D. En se fondant sur l'hypothèse que $D \subset CL(G)$, chaque CL-structure e_i dans D est la CL-structure associée à une CL-structure π_i de G. S'il en existe plusieurs dans G, on choisit une π_i quelconque mais fixée. Pour chaque variable x étiquetant un nœud interne à un cluster, on peut définir une substitution σ telle que $\sigma(x) = T$ où T est une catégorie qui étiquette le même nœud dans π_i : étant donné que x est unique, une telle substitution existe. Quand cette substitution est appliquée à $GF(D)$, cela produit une grammaire qui contient uniquement des assignations de types dans G (on doit appliquer la substitution à tout l'ensemble $CL(D)$). Lorsque l'on se focalise sur les feuilles, les lexiques des deux doivent coïncider. Nous avons bien une substitution telle que $GF(D) \sqsubset G$. □

Proposition 8 *Lorsque $D \subset CL(G)$ avec G une grammaire AB rigide, alors la grammaire $RG(D)$ existe et $RG(D) \sqsubset G$.*

Preuve. La proposition 7 nous donne $GF(D) \sqsubset G$. Il existe donc une substitution σ telle que $\sigma(GF(D)) \subset G$.
Etant donné que G est rigide, σ unifie toutes les catégories de chaque mot et $RG(D)$ existe.
$RG(D)$ est défini comme l'application du *mgu* σ_u sur $GF(D)$. Par définition de σ_u, il existe une substitution τ telle que $\sigma = \tau \circ \sigma_u$.
On a donc $\sigma(RG(D)) = \tau(\sigma_u(GF(D))) = \sigma(GF(D)) \subset G$.
Par conséquent $\tau(RG(D)) \subset G$, ce qui entraine $RG(D) \sqsubset G$. □

Proposition 9 *Si $D \subset D' \subset CL(G)$ avec G une grammaire AB, alors $RG(D) \sqsubset RG(D') \sqsubset RG(G)$.*

Preuve. D'après la proposition 8, $RG(D)$ et $RG(D')$ existent. On a $D \subset D'$ et $D' \subset CL(RG(D'))$, donc $D \subset CL(RG(D'))$. En appliquant la proposition 8 à D, avec $G = RG(D')$, on a bien $RG(D) \sqsubset RG(D')$. □

Théorème *Les grammaires de clusters convergent au sens de Gold.*

Preuve. On pose $(D_i)_{i\in\mathbb{N}}$ une séquence croissante d'ensemble d'exemples dans $CL(G)$, qui énumèrent $CL(G)$ (on a alors $\bigcup_{i\in w} D_i = CL(G)$) :

$$D_1 \subset D_2 \subset \ldots \subset D_i \subset D_{i+1} \subset \ldots \subset sPF(G)$$

Grâce à la proposition 8, pour chaque $i \in w$, la grammaire rigide $RG(D_i)$ existe et grâce à la proposition 9 les grammaires rigides $RG(D_i)$ croissent avec une relation d'$\sqsubset$-inclusion bornée par G (cela correspond à faire grandir petit à

petit le cluster correspondant aux feuilles, en terminant par le cluster complet de G) :

$$RP(D_1)) \sqsubset RP(D_2) \sqsubset \ldots) \sqsubset RP(D_i) \sqsubset RP(D_{i+1}) \sqsubset \ldots) \sqsubset G$$

Nous avons vu précédemment qu'il y a un nombre fini de grammaires telles que $H \sqsubset G$ (voir proposition 3). Il existe donc un $N \in \mathbb{N}$ tel que pour tout $n \geq N \quad RG(D_n) = RG(D_N)$.

On montre à présent que le langage appris par $RG(D_N)$ est celui appris par G, c'est à dire que $CL(RG(D_N)) = CL(G)$.

1. $CL(RG(D_N)) \subset CL(G)$. Etant donné que, par hypothèse, $RG(D_N) \sqsubset G$, la proposition 4 nous donne $CL(RG(D_N)) \subset CL(G)$.
2. $CL(RG(D_N)) \supset CL(G)$. On pose π_f une CL-structure telle que $\pi_f \in CL(G)$. Etant donné que $\bigcup_{i \in w} D_i = CL(G)$ il existe p tel que $\pi_f \in CL(D_p)$.
   - Si $p < N$, étant donné que $D_p \subset D_N$, on a $\pi_f \in D_N$ et, grâce à la proposition 6, $\pi_f \in CL(RG(D_N))$.
   - Si $p \geq N$, on a $RG(D_p) = RG(D_N)$, étant donné que les grammaires n'augmentent plus après N. Grâce à la proposition 6, on a $D_p \subset CL(RG(D_p))$, ainsi $\pi_f \in CL(RG(D_p) = CL(RG(D_N))$.
     Dans les deux cas, on a $\pi_f \in CL(RG(D_N))$.

On a ainsi démontré le théorème.□

4.4.3 Influence de la preuve de convergence

En début de section, nous avons montré comment nous pouvions passer des arbres du corpus aux arbres produits par notre grammaire de clusters.

Nous souhaitions rester dans un esprit d'apprentissage par unification en suivant l'algorithme de Buszkowski et Penn, tout en fixant plus de catégories que simplement le s à la racine : cette preuve de convergence montre que c'est bien le cas. Notre algorithme d'unification laisse totalement de côté les mots pour se concentrer uniquement sur les clusters, et la création du lexique se fait en suivant pour pouvoir appliquer notre travail à de l'analyse de phrases. Nous pouvons donc dire que notre méthode converge, étant donné que la grammaire de clusters converge. La convergence théorique est utilisable du moment que nous avons suffisamment d'exemples positifs.

4.5 Implémentation

Comme dit précédemment, notre logiciel récupère les clusters de R et des arbres d'entrée vus section 4.1.

Pour des soucis de différenciation dans R, nous avons dû donner à chaque mot une étiquette unique, de même pour les variables présentes dans les arbres. Les étiquettes associées à chaque mot sont composés du mot et de son ordre d'apparition : *le_421* sera donc l'identifiant unique du 421ème "le" apparaissant dans le corpus. Au niveau des variables, nous avons choisi de les faire commencer à *aa_n*, où *n* représente le numéro de la phrase. Au sein même d'une phrase, le reste de l'identifiant va agir comme un compteur, incrémentant d'abord la seconde lettre puis la première [3].

Lorsque nous chargeons le cluster et les arbres, il est capital que les arbres soient toujours dans le même ordre que lorsque nous en avons extrait les vecteurs, sinon un message d'erreur précisant qu'il y a une inadéquation entre les étiquettes des vecteurs et les mots des arbres apparaît.

L'algorithme décrit dans la section 4.3 est implémenté par la fonction `joinTypes`.

Les options disponibles sont :

-r : remplace, dans l'algorithme, la méthode du "premier arrivé" par "aléatoire parmi les choix possibles".

-l : impose une hauteur maximale à partir de laquelle on arrête l'unification.

--tree : par défaut, le programme envoie en sortie un lexique. Avec cette option, la sortie est sous forme d'arbres de dérivation, dont on peut ensuite extraire une grammaire.

4.6 Evaluation

Avant d'évaluer notre méthode, majoritairement en comparant les lexiques après clustering avec ceux après transduction, nous avons utilisé deux bases pour faire des comparatifs :

- Si l'on unifie les variables uniquement lorsqu'il n'y a qu'une seule solution, de nombreuses variables demeurent dans les fichiers de sortie (un ratio de 47%), réparties sur 80% des paires mot/type du lexique.
- Si l'on utilise uniquement la méthode du premier trouvé ou la méthode aléatoire pour l'étape d'unification, il n'y a plus de variable dans le résultat. Cependant, les comparaisons entre les lexiques sont très mauvaises, et nous avons moins de 50% de paires mot/type identiques.

Si l'on unifie sans l'étape de clustering, la méthode ne prend pas en compte le contexte des mots, et devient équivalente à essayer de réduire le nombre de types associés à chaque mot. Ce n'est pas l'objectif de notre travail : nous préférons avoir le même type pour des mots différents qui apparaissent dans le même contexte.

3. Nous avons calculé que si un des grands arbres de Paris VII ne contenait aucun type à part *txt*, nous avions besoin de plus de 26 variables. Cependant, le cas n'est jamais apparu.

Pour comparer notre méthode avec ces deux bases, nous avons d'abord dressé le tableau comparatif du nombre de variables restant après unification. Nous avons fait tourner notre algorithme sur 553 phrases du corpus de Paris VII, avec les trois méthodes. La table 4.3 montre l'efficacité des méthodes, calculée sur le pourcentage de variables restant après unification.

Méthode	Clustering normal	Seulement s'il y a une solution	Premier trouvé
non unifié	0	364	0
nombre de var.	686	686	686
ratio	100%	46,9%	100%

TABLE 4.3 – Ratio de variables restantes après les diverses unifications, testées sur notre extrait du corpus de Paris VII de 553 phrases.

Nous avons aussi rapidement comparé les lexiques produits par les trois méthodes d'unification avec celui venant du transducteur. Les résultats sont résumés dans la table 4.4.

Clustering normal	identique	4 832	85,1%
	équivalent	5 168	91,3%
Seulement s'il y a une solution	identique	728	12,7%
	équivalent	1 164	20,3%
Premier trouvé	identique	2 745	47,9%
	équivalent	3 576	62,4%

TABLE 4.4 – Comparaison avec le lexique venant du transducteur, exprimé en pourcentage de paires mot/type.

Cela montre que nos heuristiques donnent un résultat plus satisfaisant.

Si nous nous focalisons sur le lexique extrait de Paris VII, nous l'avons comparé plus en détail avec celui après transduction. Cela correspond à 2 076 mots, soit 5731 paires mot/type.

Les différences entre les deux lexiques correspondent à 899 paires qui s'étalent sur 379 mots, soit 14,9% des mots. Cela signifie que 85,1% des lexiques sont identiques. Dans ces 85,1% il faut noter cependant qu'il y a 2% de modifications mineures, telles qu'un np qui devient un n (majoritairement dans des cas tels que "Le président Merem") ou qu'une inversion entre les différents types des prépositions, pp, pp_{de} ou pp_a (les trois correspondent à des syntagmes prépositionnels, mais les deux derniers ajoutent comme information que la préposition utilisée est un *"de"* ou un *"à"*. Il faut noter cependant que certaines prépositions ne sont pas annotées comme ayant un *"à"* ou un *"de"* dans le corpus).

La table 4.5 trie les différences en deux catégories : d'un côté les types qui sont présents dans le lexique provenant du transducteur mais qui n'ont pas la même occurrence, et de l'autre ceux qui n'apparaissent pas dans le lexique de référence.

paires erronées	569	8,7%
paires équivalentes	336	6,2%
paires identiques	4 832	85,1%
paires utilisables	5 168	**91,3**%

TABLE 4.5 – Ratio entre les différences des lexiques, comptées en paires mot/type. On note que 91,3% du lexique après unification est sans erreur, donc utilisable en l'état.

La table 4.6 donne deux exemples extraits du lexique après unification : un mot équivalent et un considéré comme une erreur :

- Le participe passé *"accumulé"* obtient le type $np\backslash s_p$ avec l'unification, et $n\backslash n$ via le transducteur. Le premier correspond bien à celui donné à un *VPP* utilisé comme participe passé et non comme adjectif, ce qui est le cas dans le contexte. Cependant, Carpenter [1993] permet une translation de $np\backslash s_p$ vers $n\backslash n$, de même que le CCG Bank [Hockenmaier et Steedman, 2007], donc considérer les deux comme équivalents dans le cadre de cette évaluation nous semble justifié.
- Le verbe conjugué *"change"*, cependant, obtient le type $np\backslash s$ au lieu de $(np\backslash s)/np$. Nous considérons cela comme une vraie erreur : à la place d'être traité comme un verbe transitif (qui prend donc, d'un point de vue d'une grammaire AB, deux arguments), il est traité comme un verbe intransitif, qui a juste un sujet. Cette erreur vient de l'étape de clustering, où *"change"* est proche d'un verbe intransitif : l'utilisation des bigrammes engendre cette erreur, principalement parce qu'il y a souvent un modificateur entre le noyau verbal et son objet. Nous espérons que l'utilisation de trigrammes corrige cette erreur.

Mot	**Unification**	**Transduction**
accumulé	$np\backslash s_p$	$n\backslash n$
change	$np\backslash s$	$(np\backslash s)/np$

TABLE 4.6 – Exemples de ce que nous considérons comme une classe équivalente et une erreur.

Cependant, il est important de rappeler que, même avec ces incohérences, les arbres en sortie de l'unification sont toujours des arbres de dérivation d'une grammaire AB.

4.7 Extensions et perspectives

La méthode que nous utilisons est faite pour fonctionner avec des arbres de dérivation : il s'agit d'apprentissage non supervisé, mais avec des structures d'entrée contenant beaucoup d'informations dont des informations syntaxiques. Cependant, cela pourrait être étendu à n'importe quel ensemble de phrases qui ne sont pas sous forme d'arbres avec quelques modifications. L'idée serait d'utiliser à la fois des phrases simples et d'autres phrases sous forme d'arbres de dérivation.

Le problème est d'avoir les vecteurs de mots pour les phrases qui n'ont pas d'arbres syntaxiques attachés. On pourrait alors utiliser les vecteurs d'un sous-espace car certaines informations, comme le POS-tag des mots, peuvent être facilement retrouvées avec un tagger [Moot, 2010a] ou en utilisant le *Stanford Parser* [Green *et al.*, 2011]. L'utilisation des deux outils, conjoints, donnerait la totalité des informations nécessaires. Il suffit ensuite d'insérer ces vecteurs dans les clusters les plus proches, sans refaire toute l'étape de clustering.

Autrement, nous pouvons effectuer une étape de clustering avec des vecteurs partiels et ceux extraits d'arbres syntaxiques en faisant une projection sur les seconds pour diminuer le nombre de dimensions. Cela nous permettrait d'avoir une plus grande visibilité sur les mots que si nous leur donnions juste le type le plus utilisé dans un lexique de référence en fonction de leur POS-tag.

4.7.1 Application à de plus grands corpus

Nous souhaitons appliquer notre méthode actuelle à des ensembles plus larges, mais nous aurons alors affaire à des clusters beaucoup plus larges pour le corpus Sequoia complet (plus de 63 000 mots) ou encore pour le corpus de Paris VII (environ 300 000 mots). L'étape de clustering est, avec la méthode de Ward [1963], d'une complexité $O(n^3)$, et cela commence à devenir problématique pour ces grands ensembles. Il faut cependant noter que cela constitue une amélioration par rapport aux autres algorithmes d'apprentissage, étant donné que les grammaires k-valuées ont une complexité rédhibitoire lorsque l'on veut passer à l'implémentation [4], sauf si l'on paramétrise certaines valeurs telles que k, la taille de l'alphabet ou le nombre d'entrées [Costa-Florêncio et Fernau, 2012], auquel cas on a une complexité polynomiale.

Une autre solution est de faire le clustering en deux étapes : la première servirait à rassembler tous les nœuds qui ont un vecteur identique et de choisir un seul représentant parmi eux, quitte à faire une étape d'unification à ce moment là. Après avoir réduit le nombre de vecteurs à prendre en entrée, plus de

4. La complexité, pour un k donné, est le produit des coefficients binomiaux nécessaire à la sélection des n variables à unifier pour avoir au plus k types par mots, et ce pour chaque entrée lexicale.

données peuvent être traitées. La première étape peut aussi être effectuée avec un clustering non hiérarchique type k-means, dont la complexité est linéaire. Avec nos vecteurs actuels, la méthode de Ward peut gérer jusqu'à vingt-mille mots [5]. La méthode des k-means pourrait alors être appliquée sur les corpus complets (le corpus de Paris VII fait 339 521 mots) jusqu'à avoir vingt-mille clusters, et appliquer l'algorithme de Ward sur la moyenne de ces clusters-là, ou bien le centroïde, c'est à dire l'élément de chaque cluster qui est le plus significatif [Pantel, 2003].

Etant donné que nos vecteurs ont un grand nombre de dimensions et sont très vides, nous pourrions aussi appliquer la méthode de Kailing *et al.* [2004]. Celle-ci s'applique sur ce type de vecteurs et permet de ne garder que les informations importantes, en diminuant la dimension des vecteurs. Ainsi, l'étape de clustering actuelle pourrait gérer un plus grand nombre de données.

4.7.2 Amélioration de l'unification des types

Pour l'instant, nous utilisons le critère "premier trouvé" pour unifier les variables lorsque nous n'avons pas d'autre critère de choix. Une meilleure solution serait de regarder toutes les variables dans leur globalité, de leur assigner une liste d'unifications possibles et d'utiliser l'algorithme de Kuhn-Munkres [Kuhn, 1955; Munkres, 1957] pour choisir la meilleure unification globale, comme par exemple celle qui donne l'instantiation des variables avec les types les plus simples.

4.7.3 Grammaires AB du second ordre

Une classe de mots qui contient un grand nombre d'erreurs est celle des adverbes : sur un lexique de 103 types donnés à des adverbes choisis aléatoirement, il n'y a que 34,9% de similitude. Une majorité d'adverbes, hélas, ne sont pas de la forme a/a ou $a\backslash a$, avec a un type quelconque. Pour réduire ce pourcentage d'erreurs, nous proposons deux solutions.

Durant la phase d'extraction des arbres de dérivation, nous pouvons appliquer un traitement spécial sur les adverbes, comme montré en figure 4.10. Ainsi, nous sommes certains que les adverbes auront un type a/a ou $a\backslash a$. Cette étape lisse totalement le problème majeur des adverbes.

En utilisant cette solution, nous réduisons de moitié le nombre de variables présentes avant unification, en passant de 686 variables à 344. La différence entre les deux lexiques tombe alors à 12,8% (soit 732 paires mot/type), avec 1,1% d'erreurs (63 paires).

L'autre solution consiste à employer, comme proposé par [Cappelletti et Tamburini, 2009], un type X qui peut être remplacé par n'importe quel autre

5. Sur une machine ayant 8 giga-octets de mémoire vive.

type du lexique [Emms, 1993]. Par exemple, lorsqu'un nœud *PP-MOD* est utilisé en fin de phrase, la préposition aura le type $(s\backslash s)/np$, tandis qu'utilisé en tant que frère à droite d'un groupe verbal, il aura par exemple le type $((np\backslash s)\backslash(np\backslash s))/np$ S'il y a besoin d'unifier ces deux types, ils seraient alors tous les deux remplacés par : $\forall X.(X\backslash X)/np$.

Sur un extrait de 80 adverbes, cette solution effectuée manuellement lisse 56,3% des erreurs.

Il faut noter que ce traitement particulier pourrait aussi s'appliquer aux conjonctions de coordinations, qui auraient alors comme type $\forall X.(X\backslash X)/X$.

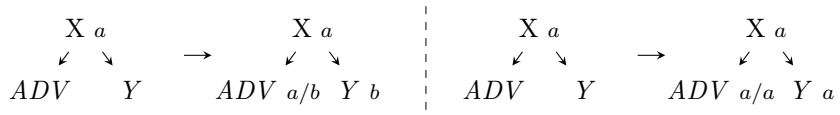


FIGURE 4.10 – Evolution du traitement des modificateurs : à gauche le traitement initial, à droite celui qui rend transparent la présence des adverbes dans la phrase.

Cette solution serait aussi applicable aux types donnés par le G-transducteur, en faisant attention toutefois à la sur-génération possible. Pour éviter ceci, il suffit de limiter les valeurs possibles de X à un ensemble donné.

4.8 Conclusion

Notre méthode, en mélangeant l'apprentissage par unification et le clustering pour guider celui-ci, c'est à dire de l'apprentissage statistique, donne des résultats prometteurs.

Le lexique, quant à lui, peut-être utilisé à 91,3% pour typer des phrases. Etant donné le nombre d'informations que nous utilisons, il sera de toute façon complexe de dépasser en terme de justesse des types donnés par le Supertagger, cependant la grammaire extraite peut permettre d'analyser des phrases, comme nous le verrons dans le chapitre suivant.

Nous utilisons actuellement très peu d'informations, c'est à dire des bigrammes lors de l'étape de clustering. Passer à l'utilisation de trigrammes devrait donner d'encore meilleurs résultats. De même, nous devons trouver un équilibre entre rajouter des informations et pouvoir gérer une certaine quantité de mots, car avec la complexité de l'algorithme que nous utilisons actuellement, chaque dimension que nous rajoutons diminue le nombre de mots passés en entrée.

Chapitre 5

Analyse de phrases

Après avoir utilisé différentes méthodes pour extraire une grammaire AB des différents corpus, nous avons voulu tester celles-ci, non pas uniquement en obtenant des arbres mais en les couplant à une étape de typage pour l'analyse syntaxique à large couverture. Cela permet donc dans un premier temps de tester notre grammaire en passant un premier lot de phrases correctes à l'analyseur et en regardant combien sont effectivement analysées puis dans un second temps d'analyser de nouvelles phrases pour savoir si elles sont correctes ou non.

Ces travaux ont fait l'objet de deux publications [Sandillon-Rezer, 2012a,b] et d'une démonstration [Sandillon-Rezer, 2012c].

L'organigramme de la figure 5.1 montre l'enchaînement des étapes à effectuer pour arriver à l'analyse de phrases et le chapitre suivra ce plan. Tout d'abord nous verrons le typage de phrases en vue de l'analyse, puis l'extraction de grammaires probabilistes (PCFG pour *Probabilistic Context Free Grammar*). Enfin nous nous focaliserons sur l'analyse et l'algorithme CYK, pour ensuite parler de l'implémentation et évaluer notre méthode.

5.1 Typage de phrases

L'étape de typage consiste à donner une ou plusieurs catégories aux mots d'une phrase, avec dans le cas où il y a plusieurs catégories possibles la probabilité de chacune.

Typage par arbres de dérivation Bien que cette solution n'ait pas été retenue finalement, nous avons la possibilité d'extraire la frontière des arbres de dérivation, composée des mots et de leur type.

Cela nous a permis, pendant la phase de développement du logiciel, de vérifier que l'algorithme de parsing fonctionnait correctement : en effet,

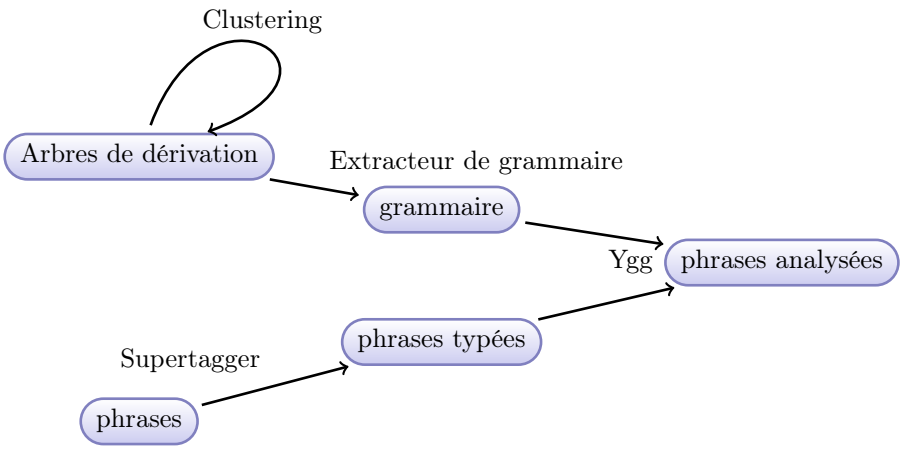


FIGURE 5.1 – Des arbres de dérivation, venant soit du transducteur soit du clustering, on extrait une grammaire qui servira, avec des phrases typées par *Grail*, d'entrée pour l'algorithme CYK probabiliste.

si la grammaire vient des mêmes arbres de dérivation que les phrases, le résultat doit être de 100% d'analyse. Nous nous sommes aussi servis de cette méthode pour comparer les couvertures des grammaires extraites des différents corpus, comme décrit dans la section 5.5.

De plus, sur certaines phrases complexes, cela nous a permis de voir comment était formé le nouvel arbre de dérivation associé et de corriger certaines règles du transducteur.

Nous aurions aussi pu décider de typer les phrases avec le lexique, cependant nous avons préféré utiliser le Supertagger [Moot, 2010a,b; Clark et Curran, 2007], entraîné avec les données extraites de nos arbres de dérivation.

Typage par *Supertagger* Cette méthode agit en deux étapes : tout d'abord on effectue une passe qui assignera un ou plusieurs POS-tags aux mots de la phrase et ensuite les mots seront "supertaggés" avec les informations venant du lexique après transduction. Prenant le contexte du mot en paramètre, il va donner des types qui devraient pouvoir se combiner. Le Supertagger, en fonction du contexte local, donne une première formule ainsi que sa probabilité p pour chaque mot de la phrase. C'est en fonction de cette probabilité p pour chaque mot qu'il peut y avoir d'autres types, en fonction d'un paramètre β. Tous les types ayant une probabilité plus grande que $\beta \times p$ seront proposés. Par conséquent, si $\beta = 1$ il n'y a qu'un seul type par mot et plus β est petit, plus il y a de types rares.

Un exemple de sortie du Supertagger, avec $\beta = 0,01$, est donné figure 5.2, tandis que la même phrase typée par le transducteur est donnée figure 5.3. La différence la plus notable est au niveau du verbe : dans le premier cas "avait" prendra deux arguments, le groupe nominal et le participe passé, alors que dans le second cas c'est le participe passé qui prendra en argument le groupe nominal et le verbe, lui, seulement le participe passé.

```
(SENT
 (Ce "np/n:0.933277" "(s/s)/n:0.0617")
 (procès "n:1")
 (gagné "n\\n:0.962703" "np\\s_p:0.0126507")
 (donne "((np\\s)/np)/pp_a:0.755706" "(np\\s)/np:0.151997"
              "(np\\s)/pp:0.0536585" "((np\\s)/pp_a)/np:0.0163681")
 (au "pp_a/n:0.699803" "(((np\\s)/np)\\((np\\s)/np))/n:0.170054"
       "pp/n:0.0887591" "(s\\s)/n:0.0163106")
 (Crédit_Lyonnais "n:0.999581")
 (les "np/n:0.966763" "(n\\n)/n:0.0162433")
 (coudées "n:0.998672")
 (franches "n\\n:0.98412")
 (pour "pp:0.718979" "((n\\n)\\(n\\n))/((n\\n)\\(n\\n)):0.113514"
          "(s\\s)/(s\\s):0.0722447" "(n\\n)/(n\\n):0.0456898"
          "(np\\s_i)/(np\\s_i):0.0132377"
          "((np\\s_i)\\(np\\s_i))/((np\\s_i)\\(np\\s_i)):0.0118367"
          "(n\\np)/(n\\np):0.00822545")
 (gérer "(pp\\(s\\s))/np:0.574272" "((n\\n)\\(n\\n))/np:0.117099"
            "(s\\s)/np:0.0539138" "(pp\\((np\\s_i)\\(np\\s_i)))/np:0.0395798"
            "(n\\n)/np:0.0353713" "(pp\\((n\\n)\\(n\\n)))/np:0.0269907"
            "((np\\s_i)\\(np\\s_i))/np:0.0155271" "(np\\s_i)/np:0.0153476"
            "pp\\(np\\s_i):0.0105502" "n\\n:0.00965271"
            "(pp\\((np\\s)\\(np\\s)))/np:0.00964481"
            "pp\\(s\\s):0.00930784" "s/np:0.0066719" "n/np:0.00652519")
 (MGM "np:0.968841" "n:0.0202698")
 (. "s\\txt:0.987188" "np\\txt:0.0118688")
)
```

FIGURE 5.2 – Phrase *"Ce procès gagné donne au Crédit_Lyonnais les coudées franches pour gérer MGM."* typée par le Supertagger.

5.2 Extraction d'une PCFG

Les grammaires AB sont usuellement représentées par le lexique de types associés aux mots. Cependant, cette représentation nous limitait aux mots présents dans les phrases analysées, ce qui nous semblait trop restrictif pour analyser des phrases venant d'autres horizons, avec un vocabulaire différent. Nous avons donc pris le parti d'extraire une grammaire probabiliste à partir des arbres de dérivation.

```
(SENT
(Ce "np/n")
(procès "n")
(gagné "n\\n")
(donne "((np\\s)/np)/pp_a")
(au "pp_a/n")
(Crédit_Lyonnais "n")
(les "np/n")
(coudées "n")
(franches "n\\n")
(pour "(s\\s)/(np\\s_i)")
(gérer "(np\\s_i)/np")
(MGM "np")
(. "s\\txt")
)
```

FIGURE 5.3 – Phrase *"Ce procès gagné donne au Crédit_Lyonnais les coudées franches pour gérer MGM."* typée par le transducteur.

Les arbres en sortie du transducteur donnent des informations à la fois syntaxiques, car nous gardons les étiquettes données par le corpus et, bien sûr, des informations structurelles. Une passe de prétraitement, avec l'extraction de la grammaire, permet de sélectionner parmi ces informations celles que nous souhaitons garder. Les types des nœuds sont, de toute façon, obligatoirement conservés.

Quels que soient les arbres de dérivation utilisés en entrée, la grammaire extraite sera hors contexte, avec une probabilité calculée sur les règles en fonction de leur racine. Pour plus de simplicité, on rappelle que les grammaires sont de la forme $\{N, F, S, R\}$:

N : l'ensemble des symboles non finaux, correspondant aux nœuds internes de l'arbre ;

F : l'ensemble des symboles finaux, correspondant à l'ensemble des mots typés ;

S : le symbole initial. On choisira, en fonction de la passe de pré-traitement, *TXT :txt* ou *txt* ;

R : l'ensemble des règles.

L'algorithme utilisé pour extraire la grammaire consiste à parcourir les arbres donnés en paramètre et à stocker les règles de dérivation que l'on rencontre. On considère qu'une règle de dérivation est constituée d'une racine et d'un ou deux fils :

Si la racine a deux fils : On est dans le cas de figure classique d'une règle d'élimination à droite ou à gauche ($a \rightarrow a/b \quad b$ ou $a \rightarrow b \quad a\backslash b$).

Si la racine a un seul fils : Il y a simplement transmission de type au fils. Ce cas de figure apparaît, par exemple, lorsqu'un groupe nominal est composé uniquement d'un nom propre, ou encore lorsqu'on est au niveau du nœud pré-terminal, c'est à dire l'étiquette de partie du discours (POS-tag) de la feuille. Dans ce cas, la feuille héritera directement du type du POS-tag.

Chaque règle est accompagnée d'un compteur et les probabilités sur les règles sont calculées par groupe ayant la même racine. On récupère aussi des informations de profondeur minimale et maximale d'apparition de la règle, cependant elles ne sont pas utilisées pour l'instant.

Ainsi, on résume dans le tableau 5.1 les différentes grammaires que peut générer l'extracteur. Chacune des versions montre un intérêt : autant la première, extraite des arbres juste après transduction, garde les informations syntaxiques données par le corpus ; autant les suivantes sont plus utiles pour appliquer un algorithme d'analyse sur des phrases non typées. Le tableau 5.2 montre des extraits des différentes grammaires en fonction des arbres donnés en entrée.

Forme des arbres	**Règles extraites**	**Spécifications**	**Nombre de règles**
Arbres de dérivation bruts	$n_1 \to n_2 \quad n_3$ $n_1 \to n_2$ $n_1 \to t_1$	Facilement normalisable en CNF : il suffit d'enlever les chaines unaires.	63 368
Retrait des chaines unaires et des étiquettes sauf les POS-tag	$n_1 \to n_2 \quad n_3$ $n_1 \to t_1$	La grammaire est en CNF.	59 505
Retrait de tous les labels et des chaines unaires. Il n'y a plus de différence entre N et T.	$n_1 \to n_2 \quad n_3$	Les mots n'apparaissent plus, ce qui laisse uniquement le squelette des arbres.	4 457

TABLE 5.1 – Grammaires extraites en fonction des arbres de dérivation donnés en entrée. On précise que $n_i \in N$ et $t_i \in T$.

5.3 Analyse : CYK

Pour l'algorithme de reconstruction des phrases, nous avons décidé d'utiliser l'algorithme CYK [Younger, 1966; Knuth, 1997; Hopcroft et Ullman, 1979]

Arbres de dérivation bruts		
Exemple de règles	$NP :np \rightarrow NPP :np$	$1,01 \times 10^{-1}$
	$NP :np \rightarrow DET :np/n\ NC :n$	$2,02 \times 10^{-1}$
	...	
Retrait des chaines unaires et des labels sauf les POS-tag		
Exemple de règles	$(np\backslash s_i)/(np\backslash s_p) \rightarrow VINF :(np\backslash s_i)/(np\backslash s_p)$	$9,53 \times 10^{-1}$
	$(np\backslash s)/(np\backslash s_p) \rightarrow CLR :cl_r$	$2,88 \times 10^{-2}$
	$cl_r\backslash((np\backslash s)/(np\backslash s_p))$	
	...	
Retrait de tous les labels et des chaines unaires		
Exemple de règles	$s \rightarrow np\ np\backslash s$	$3,81 \times 10^{-1}$
	$s \rightarrow s\ s\backslash s$	$2,65 \times 10^{-1}$
	$s \rightarrow np\backslash s_p\ (np\backslash s_p)\backslash s$	$1,13 \times 10^{-3}$
	$n \rightarrow n\ n\backslash n$	$7,97 \times 10^{-1}$
	$np \rightarrow np/n\ n$	$8,02 \times 10^{-1}$
	...	

TABLE 5.2 – Exemples des différentes règles que l'on peut extraire des arbres.

et d'en implémenter une version probabiliste : en effet, étant donné que cet algorithme a déjà été testé et est une référence, il nous a permis de tester l'efficacité de notre grammaire sans avoir à s'inquiéter de l'efficacité de l'algorithme. CYK permet, à partir d'une grammaire en forme normale de Chomsky, de savoir si une séquence appartient ou non à ladite grammaire, en travaillant de bas en haut, c'est à dire à partir des feuilles (ici les mots). Pour générer les arbres de dérivation il suffit de garder en mémoire les règles qui ont été appliquées et d'en dérouler le cheminement. La seule modification que nous avons effectuée fut de retirer la phase de typage des mots, initialement effectuée par CYK grâce aux règles de type $n_1 \rightarrow t_t$ (voir le tableau 5.1). Cependant, si la grammaire est ambiguë, donc qu'une même phrase a plusieurs arbres de dérivation possibles, CYK seul ne peut pas différencier ceux-ci.

Les probabilités permettent d'éliminer les arbres de dérivation les plus rares et sont calculées en multipliant la probabilité qu'une règle s'applique avec la probabilité de tous ses fils.

Par exemple, si l'on se focalise sur les deux nœuds :

```
(avait "((np\\s)/np)/(np\\s_p):0.929152"
           "((np\\(n\\n))/np)/(np\\s_p):0.0157694"
           "(np\\s)/(np\\s_p):0.0120462")
(dénombré "np\\s_p:0.999778")
```

On remarque que l'on peut lier les deux mots de trois manières différentes, car les trois règles existent dans les règles extraites du corpus de Paris VII. Le tableau 5.3 montre le calcul qui est effectué sur ces nœuds.

avait		dénombré		règle	proba. finale
$((np\backslash s)/np)/(np\backslash s_p)$	$0,92$	$np\backslash s_p$	1	$0,11$	$0,1$
$((np\backslash(n\backslash n))/np)/(np\backslash s_p)$	$0,02$	$np\backslash s_p$	1	$0,18$	$3,6 \times 10^{-3}$
$(np\backslash s)/(np\backslash s_p)$	$0,01$	$np\backslash s_p$	1	$0,2$	2×10^{-3}

TABLE 5.3 – C'est, dans le cas présent, la première règle qui va être gardée, du moment qu'un arbre de dérivation peut en résulter.

Il y a cependant un cas où les probabilités ne nous aident pas à différencier deux arbres de dérivation. Si l'on prend le groupe nominal *"le grand chat gris"*, on ne saura pas dire laquelle des deux analyses est la meilleure, entre rattacher d'abord gris ou grand à chat. Les probabilités ne nous sont d'aucune aide à ce niveau là. En effet dans le premier cas le calcul sera : $p(grand) \times (p(chat) \times p(gris) \times p(regle_1)) \times p(regle_2)$ et dans le second : $(p(grand) \times p(chat) \times p(regle_2)) \times p(gris) \times p(regle_1)$. La multiplication étant commutative, il n'y a pas de différence entre les deux arbres générés. Les deux arbres sont présentés figure 5.4, et dans un cas de figure comme celui-ci nous choisissons arbitrairement celui qui a une indexation plus faible pendant l'exécution de CYK.

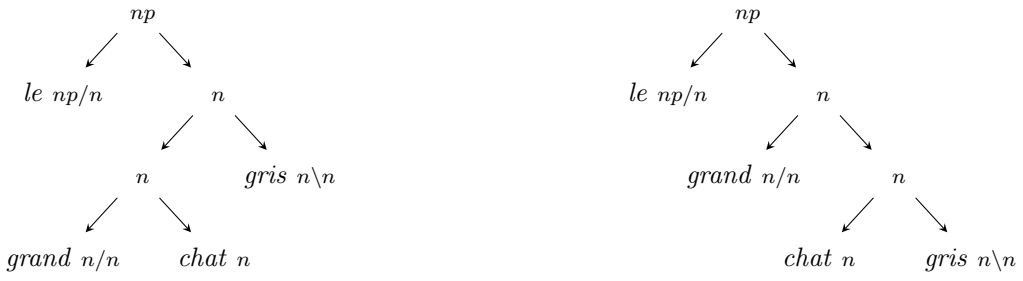


FIGURE 5.4 – Les deux arbres ont la probabilité $8,64 \times 10^{-2}$, cependant c'est celui de droite qui sera choisi.

Deux arbres de dérivation sont donnés en exemple. L'arbre de dérivation généré par CYK probabiliste avec notre grammaire, correspondant à la phrase *"Ce procès gagné donne au Crédit_ Lyonnais les coudées franches pour gérer MGM"*, montré figure 5.5. C'est majoritairement l'attachement du groupe prépositionnel final qui modifie la forme de l'arbre. L'attachement de la préposition à un groupe nominal est plus représentatif du corpus d'origine (voir section 5.5.1).

FIGURE 5.5 – Probabilité du premier arbre : $2,2 \times 10^{-9}$. Probabilité du second arbre : $1,5 \times 10^{-9}$.

On note que d'autres algorithmes auraient pu être utilisés, tel que celui d'Earley [Earley, 1973], cependant CYK demandait en entrée une grammaire très proche de celle que nous obtenions après extraction. De plus, l'ajout de l'aspect probabiliste était trivial sur cet algorithme.

5.4 Implémentation

Pour l'analyse de phrases, nous avons implémenté l'algorithme dans un programme qui prend en entrée les fichiers complets de phrases déjà supertaggées et une interface graphique qui demande simplement à l'utilisateur de taper sa phrase et s'occupe de passer celle-ci en paramètre au Supertagger puis au logiciel.

5.4.1 Le programme en ligne de commande

Outre le fait de charger les phrases typées et la grammaire, notre logiciel implémente CYK en gardant en mémoire les règles qui ont été appliquées, de manière à pouvoir recréer les différents arbres de dérivation.

Lorsque l'on ne précise pas de fichier de règle, le programme essaye simplement de construire, en fonction des types donnés aux mots, un arbre de dérivation. Il faut alors vérifier que les types sont compatibles les uns entre les autres, et les probabilités sont calculées uniquement en fonction des deux types qui se combinent.

Plusieurs options sont possibles :

-a : permet de spécifier un fichier de règles.

-d : génère le tableau de dérivation de CYK, lisible en HTML.

-b : ne calcule que le meilleur résultat. Sans elle, la sortie contient absolument tous les arbres de dérivation.

5.4.2 Ygg - interface graphique

Dans le cadre de démonstrations, nous avons créé une interface graphique, comme montré figure 5.6. Cela permet à l'utilisateur de taper la phrase qu'il veut analyser, choisir sa grammaire (ou bien décider de ne pas en utiliser, auquel cas l'algorithme va simplement combiner les types ensembles et générer un arbre, en calculant la probabilité en fonction de celles données aux types), décider s'il veut ou non un historique, qui contiendra les arbres moins probables. Le lien est fait ensuite directement avec le Supertagger et le programme en lignes de commandes.

Le résultat est affiché dans le cadre principal et le λ-terme associé dans celui juste en dessous.

5.5 Evaluation

Nous avons lancé notre logiciel sur trois corpus différents : le corpus de Paris VII, Séquoia, et 520 phrases tirées de l'Est Républicain. Les grammaires

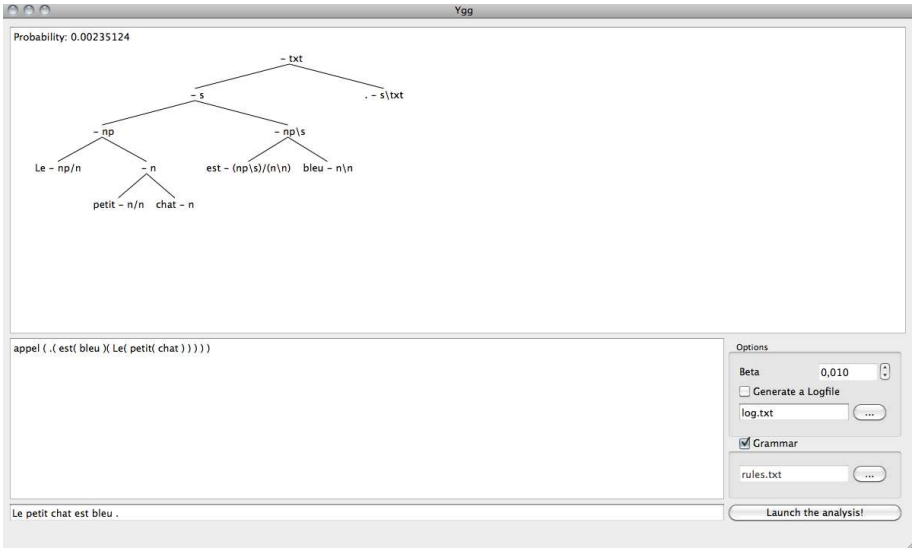


FIGURE 5.6 – Interface graphique

employées, elles, viennent de Paris VII et nous évaluons l'efficacité aussi bien de celle venant du transducteur directement que venant du clustering (voir section 4). Il est à noter, étant donné que le clustering a été effectué sur seulement 553 phrases de Paris VII, que nous avons aussi testé la grammaire extraite uniquement de ces phrases après transduction. L'analyse "avec lissage" correspond à l'utilisation de la grammaire extraite après transduction, dont les combinaisons possibles de types (lorsque celles-ci sont cohérentes au sens de Lambek) mais non représentées par une règle dans la grammaire se voient octroyées une probabilité très basse (la probabilité la plus basse parmi toutes les règles de la grammaire). Tous les résultats, exprimés en pourcentage de phrases analysées, sont résumés dans le tableau 5.4. Nous avons aussi calculé les F-scores, sur le corpus de l'Est Républicain, avec la grammaire de Paris VII (voir table 5.5).

	Paris VII 12 348	**Sequoia** 3 200	**Est Rép.** 520
transduction	11 238 (91,1%)	2 879 (89,9 %)	439 (84,4%)
avec lissage	11 439(92,6%)	2 912 (91%)	440 (84,6%)
Avant clust.	8 951 (70%)	2 415 (75,5%)	288 (55, 4%)
Après clust.	6 233 (50,5%)	1 969 (61,5%)	222 (42,7%)

TABLE 5.4 – Les phrases ont été typées par le Supertagger, avec $\beta = 0,01$.

	Précision	Rappel	F-score
Supertags corrects	89,18	84,70	86,88
Supertagger + parser	46,06	42,48	44,20

TABLE 5.5 – F-scores de la grammaire de Paris VII, calculés sur le corpus de l'Est-Républicain. Les résultats plus bas avec le Supertagger s'expliquent par le fait que celui-ci a été entraîné sur le lexique après transduction, qui contient de nombreux types par entrée lexicale.

5.5.1 Analyse du typage des prépositions

Nous nous sommes aussi focalisés sur l'analyse des syntagmes prépositionnels (*PP*, *PP-MOD*, *PP-OBJ*, etc.) et de l'attachement par rapport à la phrase. Dans un premier temps, nous avons étudié l'attachement des groupes prépositionnels dans le corpus d'origine, puis nous nous sommes focalisés sur les types des prépositions, via le transducteur et le Supertagger pour enfin nous pencher sur l'attachement dans les arbres de dérivation générés par l'algorithme CYK.

Attachement dans le corpus

Les groupes prépositionnels sont particulièrement nombreux dans le corpus (49 039 occurrences). Comme nous pouvons le voir dans le tableau 5.6, ils sont majoritairement étiquetés *PP*. Leur attachement de départ dans le corpus est aussi particulièrement important, car c'est celui-ci qui définira le type de la préposition. Le tableau 5.7 résume la répartition des syntagmes prépositionnels dans le corpus, en fonction de leurs parents. En effet, la transduction aura tendance à donner un type aux syntagmes prépositionnels qui correspond à leur place dans la structure de la phrase. Ainsi, dans un groupe nominal, le *PP* aura plus souvent le type $n\backslash n$, alors qu'au milieu d'une phrase le typage sera plus complexe. Lors de la transduction, on ne change pas l'ordre des mots, mais quelques fois leur attachement au sein de la structure. Cependant, on peut dire que les groupes prépositionnels ne bougent pas, sauf s'ils sont à l'extérieur d'un noyau verbal et que celui-ci se termine par un *VPP*, auquel cas on lie plus spécifiquement le participe passé au groupe prépositionnel, comme on a pu le voir section 3.2.

Etiquette	**occur.**	**Etiquette**	**occur.**	**Etiquette**	**occur.**
PP	32 023	*PP-MOD*	11 899	*PP-DE_OBJ*	1 668
PP-A_OBJ	1 565	*PP-P_OBJ*	1 389	*PP-ATS*	323
PP-OBJ	130	*PP-ATO*	30	*PP-SUJ*	12

TABLE 5.6 – Distribution des groupes prépositionnels en fonction de leur étiquette.

Syntagme parent	**occurrence**	**Etiquette la plus courante**	**pourcentage**
Syntagme Nominal	24 817	*PP*	99, 2%
Phrase complète	8 478	*PP-MOD*	72, 2%
Proposition rel. ou sub.	3 833	*PP-MOD*	57, 9%
Proposition participiale	3 552	*PP*	80, 1%
Proposition infinitive	3 190	*PP-MOD*	63, 5%
Syntagme prépositionnel	843	*PP*	89, 2%
Noyau verbal	45	*PP*	88, 9%

TABLE 5.7 – Distribution des groupes prépositionnels en fonction de leurs parents. On remarque que les groupes nominaux sont ceux qui regroupent le plus de *PP*, c'est à dire presque la moitié.

Typage des syntagmes prépositionnels

Pour étudier le typage, nous nous sommes focalisés sur les groupes prépositionnels dont, bien sûr, la transduction avait réussi. Cela fait tomber le nombre de syntagmes prépositionnels à 45 351 (92, 5% du total). Les quatre familles de types les plus donnés par le transducteur (au dessus de 2 000 fois) sont résumés dans le tableau 5.8. Ils couvrent 92, 2% des types que l'on peut trouver pour des prépositions. Les types restants sont marginaux et correspondent, par exemple, à un syntagme prépositionnel contenant uniquement un pronom relatif qui prend en argument une subordonnée.

Le typage effectué avant l'analyse via CYK, avec le Supertagger, nous permet de régler la précision que l'on souhaite sur les types en jouant sur le paramètre β. La table 5.9 résume la justesse des types donnés aux prépositions en fonction de β. On remarque que ce sont des mots difficiles à typer, étant donné que les résultats sont inférieurs aux résultats globaux, bien que les adverbes et les verbes soient encore plus complexes à typer de manière exacte.

Il faut cependant noter qu'il n'est pas nécessaire d'avoir une formule correcte pour que l'attachement du syntagme prépositionnel dans la phrase soit correct.

Famille de type	**occurrence**
n\n ou np\np ou n\np	23 901
a\a (ex. s\s)	8 548
pp ou pp_a ou pp_{de}	6 486
a/a (ex. s/s)	2 882

TABLE 5.8 – Les quatre familles de types les plus courants correspondent à un modificateur de groupe nominal, un groupe prépositionnel généralement argument d'un groupe verbal et des modificateurs de phrase, placés au début ou à la fin de la phrase.

β	**justesse des prépositions**	**justesse globale**
$1,0$	$61,0\%$	$76,9\%$
$0,1$	$83,1\%$	$87,0\%$
$0,05$	$86,2\%$	$88,9\%$
$0,01$	$90,2\%$	$91,7\%$

TABLE 5.9 – Justesse du typage via le Supertagger.

Attachement des syntagmes prépositionnels dans les arbres reconstitués

Pour cette partie, nous nous sommes focalisés sur cinquante-cinq *PP*, que nous avons sélectionnés dans le corpus d'origine, de manière à respecter le ratio présenté dans le tableau 5.6. Cela correspond à 21 phrases, dont l'analyse a réussi. Nous avons généré les types possibles avec $\beta = 0,05$. Ensuite, nous avons étudié la différence de types donnés aux prépositions ainsi que leur attachement. On remarque, dans le tableau 5.10, que les syntagmes prépositionnels liés aux groupes nominaux sont attachés sensiblement au même endroit. On note une différence faible entre les groupes prépositionnels qui seront arguments d'un verbe, un peu plus importante entre les modificateurs globaux qui agissent sur toute la phrase. Il y a quatre cas, dans les arbres régénérés via CYK, où l'algorithme a jugé plus pertinent de préférer le type n ou np pour le syntagme prépositionnel ("On ne porte pas impunément atteinte *à des tabous*."), alors qu'on s'attend plutôt à une analyse qui lierait "atteinte" et "à" et qui prendrait en argument le groupe nominal "des tabous"[1].

On peut dire que le typage et l'attachement des syntagmes prépositionnels semblent cohérents avec l'attachement présent dans le corpus d'origine, ainsi que le typage effectué par le transducteur.

Type	**occurrence après transduction**	**occurrence après CYK**
pp, pp_{de} ou pp_{a}	6	3
Modificateur de *NP*	35	37
Modificateur de *SENT*	9	4
np	0	4
Modificateur autre	5	7

TABLE 5.10 – Typage des prépositions dans le cadre d'une transduction comparé à celui effectué via le Supertagger avant reconstitution des arbres de dérivation avec CYK. Les modificateurs autres sont des modificateurs de propositions infinitives ou de syntagmes adjectivaux.

Le typage, cependant, n'est pas entièrement lié à l'attachement dans la

1. L'analyse CYK fait ressortir l'aspect idiomatique de "porter atteinte à".

phrase. Nous avons comparé l'attachement des syntagmes prépositionnels et nous pouvons dire que, sur les 55 cas, il y en a 37 placés de manière identique et 18 non, soit 67, 3% de ressemblance. Les différences majeures sont au niveau des prépositions qui sont le plus souvent attachées aux groupes nominaux et arguments des noyaux verbaux (ceux-ci peuvent alors prendre le type *np* plutôt que *pp*).

5.6 Perspectives

La mise en place d'un analyseur qui utilise une version probabiliste de CYK nous a permis de tester nos grammaires aussi bien que d'améliorer le typage de phrases, tout du moins de comparer celui produit par le transducteur et la méthode semi-automatique mise en place par Moot [2010b].

Il pourrait être intéressant d'utiliser d'autres algorithmes que CYK, tel que l'algorithme d'Earley [1973], ou de typer les phrases en utilisant un système tel que SYGFRAN [Chauché, 2011]. Pour améliorer la précision de l'analyse, il serait bon d'intégrer des techniques comme celles de Auli et Lopez [2011] ou Zhang et Stephen [2011] dans notre parser ou encore utiliser des grammaires du second ordre, comme vu précédemment (section 4.7.3).

Chapitre 6

Conclusion

Au cours de ces années de recherche, nous avons voulu étudier les différentes approches que nous pouvions avoir au sujet des grammaires AB et de leur apprentissage. Si le point de départ de ce travail a été le désir simple d'appliquer un algorithme déjà existant sur des corpus qui n'avaient pas le bon format pour servir d'entrée, nous nous sommes rapidement éloignés de celui-ci pour créer un ensemble d'outils permettant de manipuler les corpus d'une part pour en extraire une grammaire AB, d'autre part pour s'en servir avec les algorithmes plus classiques que nous avons pu voir.

Sans s'arrêter sur la simple extraction de grammaires, nous avons mis au point une chaîne de traitement pour tester celles-ci, en incluant des probabilités, ce qui gomme les défauts que l'on peut trouver à l'algorithme CYK lorsque celui-ci est utilisé, en plus de reconnaître des phrases, pour générer les arbres de dérivation.

Tous nos outils ont été implémentés dans l'idée et l'espoir d'être utilisés par d'autres chercheurs, en dissociant à chaque fois le cœur du programme et les règles prises en paramètre pour la langue. Pour enjoindre l'utilisation aussi bien que la modification de nos programmes, nous les avons publiés sous `GNU GPL`.

Contributions principales

La mise en place d'un transducteur généralisé, appelé G-transducteur, nous a permis de mettre au point une nouvelle méthode d'extraction de grammaires AB, tout en améliorant le système d'écriture de règles de transduction. La couverture actuelle est de 14 844 phrases sur les 16 053 des trois corpus, soit 92,5%. Cela laisse de côté certaines constructions rares et complexes que nous avons décidé de ne pas traiter car le gain au niveau de la grammaire serait minime : nous aurions dû rajouter au moins une règle complexe pour chaque phrase qui n'était pas analysée, ce que nous voulions justement éviter en créant

le G-transducteur.

Nous nous sommes aussi penchés sur une méthode d'apprentissage plus proche d'une méthode à la Gold. Comme initialement nous voulions utiliser notre transducteur pour convertir les arbres du corpus de Paris VII en format d'entrée des algorithmes d'apprentissage de grammaires k-valuées, nous avons décidé d'améliorer cette idée. Pour ceci, l'intégration de types fixés dans les FA-structures a été le premier pas. Cela permet d'utiliser un algorithme d'unification en ajoutant déjà des informations à la base. La seconde innovation de cette méthode a été l'utilisation de clustering, effectué en fonction des informations extraites des corpus, pour guider l'étape d'unification. C'est ainsi que nous avons alors pensé à réduire le nombre de types associés aux mots apparaissant dans un contexte similaire plutôt que simplement le nombre de catégories associées aux mots dans le lexique. En comparaison avec le lexique extrait des arbres de dérivation après transduction, on note 91,3% de similitude entre les deux.

Etant donné que l'extraction de grammaires a comme intérêt majeur de pouvoir faire l'analyse syntaxique à grand échelle, nous avons testé nos grammaires grâce à une version probabiliste de l'algorithme CYK. Nous pouvons ainsi recréer l'arbre le plus probable d'une phrase, après avoir effectué une étape de typage via le Supertagger ou en rassemblant les feuilles des arbres de dérivation avec leurs types. Nous permettons à l'utilisateur de choisir s'il veut utiliser un lissage de règles (c'est à dire considérer que lorsque deux types peuvent se combiner mais que la règle n'est pas présente dans la grammaire, il en existe cependant une avec une probabilité très faible) ou non. Les résultats, avec lissage, sont de 92,1% sur la totalités des corpus testés et sans lissage de 90,6%.

Perspectives

Les travaux de recherche que nous avons mené nous ont donné envie d'approfondir ceux-ci, de dépasser les limites que nous fixent parfois les outils que nous avons sélectionné et d'approfondir nos recherches dans ce domaine.

Passage aux grammaires plus complexes

Nous avons vu les limitations des grammaires AB : impossibilité de gérer l'extraction, problèmes pour les inversions et la quantification. Bien qu'elles soient élégantes et présentent l'avantage de représenter la syntaxe d'une langue (même si certains mots, comme les clitiques objet, se retrouvent avec des types particulièrement complexes) ainsi qu'une ébauche de la sémantique de celle-ci, nous prévoyons d'utiliser des grammaires plus évoluées.

L'évolution la plus simple serait de passer aux grammaires du second ordre, c'est à dire d'autoriser un type générique, X, qui pourrait être remplacé par n'importe quel autre type existant dans le lexique. Cela pourrait être utilisé dans deux cas principalement : les modificateurs (sous-arbres se terminant par *-MOD* et adverbes) et les conjonctions de coordinations. Nous avons vu précédemment que les adverbes possèdent de nombreux types. Pour 14 315 occurrences dans le corpus de Paris VII, il y a 846 types assignés. Bien que certaines catégories ne soient pas de la forme $X\backslash X$ ou X/X, car nous avons pris le parti de reporter la complexité des types sur les adverbes et non leurs arguments, l'utilisation d'une grammaire du second ordre réduirait, dans ce cas, le nombre de type à 199, ce qui équivaudrait à une réduction de 76,5% de la taille du lexique pour les adverbes. Si l'on se focalise sur le lexique, cela correspond à 626 adverbes, soit 4,4% de la totalité des adverbes. Ainsi, cela ferait grandement baisser la taille du lexique.

Cette amélioration n'est pas incompatible avec le passage aux grammaires de Lambek, bien qu'il faille faire attention à rester décidable lorsque l'on ajoute, avec des types génériques, les règles d'introduction. Les arbres de dérivation sont adaptés à ces grammaires, cependant les informations que nous avons ne sont pas suffisantes pour l'instant.

Enfin, la dernière étape serait de passer aux grammaires de Lambek multimodales, qui gèrent aussi bien les traces que les inversions et la quantification. Les avantages seraient de diminuer la complexité des types en utilisant les règles structurelles explicitement autorisées par la forme lexicale des formules. Celles-ci permettent de noter qu'un foncteur devrait avoir un argument à un endroit donné mais que cet argument n'est pas directement présent à sa droite ou sa gauche. Aussi bien dans le cas du calcul de Lambek que du calcul de Lambek multimodal, les dérivations ne sont plus sous forme d'arbres réguliers. Cela nécessiterait donc de mettre au point des modèles de transduction plus compliqués. Le choix le plus naturel serait alors des transducteurs d'arbres-vers-graphes. De plus, étant donné que pour extraire ces grammaires plus détaillées, il nous faut des informations qui ne sont pas présentes dans les annotations actuelles des corpus, ceci sera un projet de longue durée.

Gestion de plus grands corpus avec le clustering

L'utilisation du clustering pour guider l'unification nous a montré des résultats prometteurs. Le plus grand frein à cette méthode, cependant, est la limitation actuelle du nombre de vecteurs. Pour dépasser celle-ci, nous avons plusieurs idées.

Etant donné que nous pouvons gérer, avec la méthode de Ward, jusqu'à 20 000 vecteurs, nous pouvons utiliser en prétraitement la méthodes des k-means, dont la complexité est linéaire, jusqu'à avoir autant de rassemblements

de vecteurs que nous pouvons traiter. Ensuite, il nous suffirait de sélectionner le vecteur le plus représentatif de chaque ensemble de vecteurs. Il faudrait certainement faire une première passe d'unification pour chaque ensemble.

Nous avons aussi remarqué que de nombreux mots avaient des vecteurs identiques : cela se voit car, au niveau zéro du cluster, ce sont ceux qui sont rassemblés. Plutôt que de passer tous les vecteurs identiques à l'étape de clustering, étant donné qu'il est simple d'un point de vue informatique de lier un unique vecteur à plusieurs mots, nous pourrions gagner de la place mémoire.

Ces deux solutions demanderaient cependant de faire des ajustements au niveau du chargement du cluster dans notre programme d'unification, de manière à retrouver les informations de toutes les entrées du lexique.

Gestion de la sémantique

La sémantique est un point que nous n'avons pour ainsi dire pas abordé lors de cette thèse, préférant nous focaliser sur la syntaxe des phrases. Il faut rappeler que les grammaires AB permettent une gestion limitée de la sémantique : en effet, à partir des catégories données à chaque mot du lexique, il est possible d'obtenir le type d'un λ-terme correspondant. En pratique, bien que nous ne puissions pas réellement différencier les mots avec leurs λ-termes, il y a un nombre de possibilités sémantiques dans le lexique limité par le nombre de types présents dans le lexique. Les λ-termes que nous extrayons actuellement contiennent les mots en constante.

La quantification et l'extraction, d'un point de vue sémantique, sont deux bonnes raisons de passer à des grammaires plus complexes que les grammaires AB. Bien qu'on puisse passer d'une grammaire de Lambek à une grammaire AB en gardant la même sémantique (voir [Buszkowski, 1987]), les grammaires de Lambek permettent une factorisation des termes plus élégante. La factorisation est encore améliorée lorsqu'on travaille sur les grammaires multimodales, mais il n'y a alors plus nécessairement de passage aux grammaires AB possible [1].

Bien que la question de la sémantique des phrases soit un domaine totalement nouveau pour nous, sur lequel nous ne nous sommes pas encore penchés, cela nous semble être une étape incontournable que de nous focaliser sur les méthodes d'analyses sémantiques compatibles avec notre travail. Ainsi nous couvririons la totalité des aspects liés à l'analyse de phrases, c'est à dire la syntaxe aussi bien que la sémantique.

1. Par exemple, les grammaires multimodales peuvent générer des langages non-algébriques qui ne sont alors plus représentables en grammaires AB.

Bibliographie

Aarts, E. et Trautwein, K., 1995. Non-associative Lambek categorial grammar in polynomial time. *Mathematical Logic Quarterly*, 41 :476–484.

Abeillé, A. et Clément, L., 2003. Annotation morpho-syntaxique. URL `http://llf.linguist.jussieu.fr`

Abeillé, A., Clément, L. et Toussenel, F., 2003. Building a Treebank for French. *Treebanks : Building and Using Parsed Corpora*.

Adriaans, P. W., 2001. Learning Shallow Context-free Languages under Simple Distributions.

Adriaans, P. W., Trautwein, M. et Vervoort, M., 2000. Towards high speed grammar induction on large text corpora. 1963 :173–186.

Ajdukiewicz, K., 1935. Die syntaktische konnexität. *Studia Philosophica*, 1 :1–27.

Angluin, D., 1980. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1) :46–62.

Auber, D. et Mary, P., 2007. Tulip : Better visualization through research. URL `http://tulip.labri.fr/`

Auli, M. et Lopez, A., 2011. Efficient ccg parsing : A* versus adaptive supertagging. Dans Dekang Lin, Yuji Matsumoto et Rada Mihalcea, rédacteurs, *ACL*, pages 1577–1585. The Association for Computer Linguistics.

Autebert, J.M., Boasson, L. et Gabarro, J., 1984. Context-free grammars in greibach normal forms. *Bulletin of the EATCS*, 24 :44–47.

Baldridge, J., 2002. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. Thèse de doctorat, University of Edinburgh.

Bar-Hillel, Y., 1953. *A Quasi-Arithmetical Notation for Syntactic Description*, tome 29.

Bechet, D. et Foret, A., 2003a. k-valued non-associative Lambek categorial grammars are not learnable from strings.

Bechet, D. et Foret, A., 2003b. k-Valued Non-Associative Lambek Grammars are Learnable from Function-Argument Structures. *Electr. Notes Theor. Comput. Sci.*, 84 :60–72.

Bernardi, R. et Moortgat, M., 2010. Continuation semantics for the lambek-grishin calculus. *Inf. Comput.*, 208(5) :397–416.

Bernardi, R. et Moot, R., 2003. Generalized quantifiers in declarative and interrogative sentences. *Logic Journal of the IGPL*, 11(4) :419–434.

Bonato, R., 2006. A study on learnability for rigid lambek grammars. *CoRR*.

Bonato, R. et Retoré, C., 2001. Learning rigid lambek grammars and minimalist grammars from structured sentences. *Third workshop on learning language in logic, Strasbourg*, pages 23—34.

Buszkowski, W., 1987. Discovery procedures for categorial grammars. *Categories, Polymorphism and Uni- fication, Universiteit van Amsterdam*.

Buszkowski, W., 1996. Extending lambek grammars to basic categorial grammars. *Journal of Logic, Language and Information*, 5 :279–295.

Buszkowski, W. et Penn, G., 1990. Categorial grammars determined from linguistic data by unification. *Studia Logica*, 49(4) :431–454.

Béchet, D., Bonato, R., Dikovsky, A., Foret, A., Nir, Y. Le, Moreau, E., Retoré, C. et Tellier, I., 2007. "modèles algorithmiques de l'acquisition de la syntaxe : concepts et méthodes, résultats et problèmes". *Recherches linguistiques de Vincennes*. Vol. 37, Presses Universitaires de Vincennes.

Candito, M. et Seddah, D., 2012. Le corpus sequoia : annotation syntaxique et exploitation pour l'adaptation d'analyseur par pont lexical.

Cappelletti, M. et Tamburini, F., 2009. Parsing with polymorphic categorial grammars. *CICLing*.

Carpenter, B., 1993. The interpretation of lexical rules.

Chauché, J., 2011. Une application de la grammaire structurelle : L'analyseur syntaxique du francais sygfran.
URL `http://www.lirmm.fr/~chauche/SourcesAnalyse/SYGFRAN.html`

Chomsky, N., 1955. *The logical structure of linguistic theory*. M.I.T. Library.

Chomsky, N., 1965. *Aspects of the Theory of Syntax*. Cambridge : M.I.T. Press.

Clark, A. et Lappin, S., 2010. Unsupervised learning and grammar induction. pages 197–220.

Clark, S. et Curran, J. R., 2007. Wide-coverage efficient statistical parsing with ccg and log-linear. *Models, Computational Linguistics*, 33(4) :493–552.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S. et Tommasi, M., 2007. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`.

Costa-Florêncio, C., 2001. Consistent identification in the limit of any of the classes k-valued is np-hard. *Lecture Notes in Artificial Intelligence*, pages 125–138.

Costa-Florêncio, C., 2003. Learning categorial grammars. *PhD thesis*.

Costa-Florêncio, C. et Fernau, H., 2012. On families of categorial grammars of bounded value, their learnability and related complexity questions. *Theor. Comput. Sci.*, 452 :21–38.

de Groote, P., 1999. The non-associative lambek calculus with product in polynomial time. Dans N. V. Murray, rédacteur, *TABLEAUX*, tome 1617 de *Lecture Notes in Computer Science*, pages 128–139. Springer.

de Groote, P., 2001. Towards abstract categorial grammars. Dans *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155.

Earley, J., 1973. An efficient context-free parsing algorithm. *Commun. ACM*, pages 57–61.

Emms, M., 1993. Parsing with polymorphism. Dans Steven Krauwer, Michael Moortgat et Louis des Tombe, rédacteurs, *EACL*, pages 120–129. The Association for Computer Linguistics.

Engelfriet, J. et Vogler, H., 1994. The translation power of top-down tree-to-graph transducers. *Journal of Computer and System Sciences*, 49(2) :258–305.

Foret, A., 2001. Conjoinability and unification in Lambek categorial grammars.

Foret, A., Bechet, D. et Poupard, E., 2006. Categorial grammar acquisition from a french treebank. Dans *In Actes de la Conférence d'APprentissage 2006*.

Foret, A. et Le Nir, Y., 2002. Lambek rigid grammars are not learnable from strings. Dans *COLING'2002, 19th International Conference on Computational Linguistics*. Taipei, Taiwan.

Fülöp, Z. et Vogler, H., 1998. *Syntax-directed semantics : formal models based on tree transducers*. Monographs in theoretical computer science. Springer.

Gaiffe, B. et Nehbi, K., 2009. Tei est républicain.
URL `http://www.cnrtl.fr/corpus/estrepublicain/`

Gentzen, G., 1935. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39 :176–210.

Gold, E. M., 1967. Language identification in the limit. *Information and Control*, 10(5).

Green, S., Marneffe, M. C., Bauer, J. et Manning, C. D., 2011. Multiword expression identification with tree substitution grammars : A parsing tour de force with french. *EMNLP proceedings*, pages 725–735.

Hockenmaier, J., 2003. Data and models for statistical parsing with combinatory categorial grammar.

Hockenmaier, J., 2006. Creating a ccgbank and a wide-coverage ccg lexicon for german. *Proceedings of COLING/ACL, Sydney*, pages 505–512.

Hockenmaier, J. et Steedman, M., 2007. CCGbank : a corpus of CCG derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33(3) :355–396.

Hopcroft, J. E. et Ullman, J. D., 1979. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company.

Ihaka, R. et Gentleman, R., 1993. R project.
URL `http://www.r-project.org/`

Johnshon, M., 1998. PCFG models of linguistic tree representations. 24 :613–632.

Kailing, K., Kriegel, H. P. et Kröger, P., 2004. Density-connected subspace clustering for high-dimensional data. *Proceedings of the Fourth SIAM International Conference on Data Mining*, pages 246–257.

Kanazawa, M., 1998. *Learnable Classes of Categorial Grammars.* Studies in logic, language and information. Center for the Study of Language and Information, Stanford University.

Kaplan, R. M. et Kay, M., 1994. Regular models of phonological rule systems. *Comput. Linguist.*, 20(3) :331–378.

Knight, K. et Graehl, J., 2005. An overview of probabilistic tree transducers for natural language processing. Dans *Computational Linguistics and Intelligent Text Processing*, tome 3406 de *Lecture Notes in Computer Science*, pages 1–24. Springer.

Knuth, D. E., 1997. *The Art of Computer Programming Volume 2 : Seminumerical Algorithms (3rd ed.).* Adison-Wesley Professional.

Kraak, E., 1998. A deductive account of french object clitics. *SYntax and Semantics*, page 271–312.

Kubota, Y. et Levine, B., 2013. Coordination in Hybrid Type-Logical Categorial Grammar. tome 60. Ohio State University Working Papers in Linguistics.

Kuhn, H. W., 1955. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, (1–2) :83–97.

Kurtonina, N. et Moortgat, M., ? ? ? ? Structural control. *Specifying syntactic structures.*

Lamarche, F. et Retoré, C., 1996. Proof nets for the lambek calculus — an overview. Dans V. M. Abrusci et Claudia Casadio, rédacteurs, *Proofs and Linguistic Categories*, pages 241–262. CLUEB, Bologna.

Lambek, J., 1958. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3) :154–170.

Lambek, J., 1961. On the calculus of syntactic types. Dans R. Jakobson, rédacteur, *Structure of Language and its Mathematical Aspects, Proceedings of the Symposia in Applied Mathematics*, tome XII, pages 166–178. American Mathematical Society.

Levy, R. et Andrew, G., 2006a. Tregex and tsurgeon : tools for querying and manipulating tree data structures.
URL `http://nlp.stanford.edu/software/tregex.shtml`

Levy, R. et Andrew, G., 2006b. Tregex and tsurgeon : tools for querying and manipulating tree data structures.
URL `http://nlp.stanford.edu/software/tregex.shtml`

Maletti, A., Graehl, J., Hopkins, M. et Knight, K., 2009. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39(2) :410–430.

Marcus, M. P., Santorini, B. et Marcinkiewicz, M. A., 1993. Building a large annotated corpus of english : The penn treebank. *COMPUTATIONAL LINGUISTICS*, 19(2) :313–330.

Miller, P. et Monachesi, P., 2003. Les pronoms clitiques dans les langues romanes. pages 67–123.

Moortgat, M., 1997. Categorial type logics. *Handbook of logic language*, pages 93—177.

Moortgat, M. et Moot, R., 2001. Cgn to grail : Extracting a type-logical lexicon from the cgn annotation. *Language and Computers*, 37(1) :126–143.

Moortgat, M. et Morrill, G., 1991. Heads and phrases. type calculus for dependency and constituent structure. Dans *Journal of Language, Logic and Information*.

Moortgat, M. et Oehrle, R. T., 1994. Adjacency, dependency and order. pages 447–466.

Moot, R., 2010a. Automated extraction of type-logical supertags from the spoken dutch corpus. *Complexity of Lexical Descriptions and its Relevance to Natural Language Processing : A Supertagging Approach*.

Moot, R., 2010b. Semi-automated extraction of a wide-coverage type-logical grammar for french. *Proceedings TALN 2010, Monreal*.

Moot, R., 2013. A type-logical treebank for french. *High-Level Methods for Grammar Engineering*.

Moot, R. et Puite, Q., 2002. Proof nets for the multimodal lambek calculus. *Studia Logica*, 71(3) :415–442.

Moot, R. et Retoré, C., 2012. *The logic of Catégorial Grammars : A deductive account of Natural Language Syntax and Semantics*. Springer.

Moreau, E., 2006. Acquisition de grammaires lexicalisées pour les langues naturelles. *Thèse de Doctorat*.

Moreau, E., 2007. Apprentissage symbolique de grammaires et traitement automatique des langues. pages 213–222.

Morrill, G. V., 1994. *Type Logical Grammar : Categorial Logic of Signs*. Springer.

Morrill, G. V., 2001. *Categorial Grammar : Logical Syntax, Semantics, and Processing*, tome 37. Oxford University Press.

Morrill, G. V., 2013. *A Count Invariant for Lambek Calculus with Additives and Bracket Modalities*. Glyn Morrill and Mark-Jan Nederhof.

Munkres, J., 1957. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5 :32–38.

Muskens, R., 2001. Categorial grammar and lexical-functional grammar. Dans *Proceedings of the LFG01 Conference, University of Hong Kong, CSLI Publications, Stanford CA*, pages 259–279.

Oehrle, R. T., 1994. Term-labeled categorial type systems. *Linguistics and Philosophy*, 17 :633–678.

Pantel, P. A., 2003. Clustering by committee. *PhD thesis*.

Pentus, M., 1992. Equivalent types in Lambek calculus and linear logic. MIAN Prepublication Series for Logic and Computer Science LCS-92-02, Steklov Mathematical Institute.

Pentus, M., 1997. Product-free lambek calculus and context-free grammars. *Journal of Symbolic Logic*, 62(2) :648–660.

Retoré, C. et Bonato, R., 2013. Learning lambek grammars from proof frames. *Festschrift for Joachim Lambek 90th brithday*.

Roorda, D., 1991. *Resource Logics : A Proof-theoretical Study*. Thèse de doctorat, University of Amsterdam.

Rounds, W. C., 1970. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3) :257–287.

Sandillon-Rezer, N-F., 2011. Learning categorial grammar with tree transducers. *ESSLLI Student Session Proceedings*.

Sandillon-Rezer, N-F., 2012a. Extraction de PCFG et analyse de phrases pré-typées. *Actes de la conférence conjointe JEP-TALN-RECITAL 2012, volume 3 : RECITAL*, pages 205–218.

Sandillon-Rezer, N-F., 2012b. PCFG extraction and pre-typed sentence analysis. *ESSLLI Student Session Proceedings*, pages 164–173.

Sandillon-Rezer, N-F., 2012c. Ygg, parsing french text using AB grammars. *Logical Aspects of Computational Linguistics - Demo session*, pages 17–20.

Sandillon-Rezer, N-F., 2013a. Clustering for categorial grammar induction. *High-level Methodologies for Grammar Engineering (HMGE) Proceedings*.

Sandillon-Rezer, N-F., 2013b. Inférence grammaticale guidée par clustering. *Actes de la 15e Rencontres des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RECITAL'2013)*, pages 28–41.

Sandillon-Rezer, N-F. et Moot, R., 2011. Using tree transducers for grammatical inference. *Proceedings of Logical Aspects of Computational Linguistics 2011*, pages 235–250.

Sandillon-Rezer, N.F., 2013c. Clustering pour grammaires catégorielles du second ordre. Dans *Actes de MIXEUR 2013 : Méthodes mixtes pour l'analyse syntaxique et sémantique du français*, pages 88–91. Les Sables d'Olonne, France.

Steedman, M., 2001. *The Syntactic Process*. MIT Press, Cambridge, Massachusetts.

Tiede, H., 1998. Lambek calculus proofs and tree automata. 2014 :251–265.

Ward, J. H., 1963. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58 :236–244.

Younger, D. H., 1966. Context free grammar processing in n^3. pages 7–20.

Zhang, Y. et Stephen, C., 2011. Shift-reduce ccg parsing. Dans Dekang Lin, Yuji Matsumoto et Rada Mihalcea, rédacteurs, *ACL*, pages 683–692. The Association for Computer Linguistics.

Règles de transduction

Rule :
→ (NP:* NC PP)
← (NP:* NC:* PP:**)
Rule :
→ (NP:* DET tree)
← (NP:* DET:*/n NP:n)
Rule :
→ (PP:* P NP)
← (PP:* P:*/np NP:np)
Rule : use-ponct
→ (SENT tree PONCT)
← (TEXT:txt SENT:s PONCT:s\\txt)
Rule :
→ (NP:* NC AP)
← (NP:* NC:* AP:**)
Rule :
→ (NP:* DET NC)
← (NP:* DET:*/n NC:n)
Rule :
→ (PP-MOD:* P NP)
← (PP-MOD:* P:*/np NP:np)
Rule :
→ (NP-SUJ:* DET tree)
← (NP-SUJ:* DET:*/n NP:n)
Rule :
→ (NP-OBJ:* DET tree)
← (NP-OBJ:* DET:*/n NP:n)
Rule :
→ (SENT:* tree PP-MOD)
← (SENT:* SENT:* PP-MOD:**)
Rule :
→ (NP:* tree PP)
← (NP:* NP:n PP:n*)
Rule :
→ (PP:* P (NP NC))
← (PP:* P:*/n (NP:n NC:n))
Rule :
→ (NP:* ADJ tree)
← (NP:* ADJ:*/n NP:n)
Rule :
→ (PP:* P+D NP)
← (PP:* P+D:*/n NP:n)
Rule :
→ (SENT:* NP-SUJ VN)
← (SENT:* NP-SUJ:np VN:np*)
Rule :
→ (NP-SUJ:* DET NC)
← (NP-SUJ:* DET:*/n NC:n)
Rule :
→ (NP:* tree NPP)
← (NP:* NP:n NPP:n*)
Rule :
→ (NP:* tree VPpart)
← (NP:* NP:* VPpart:**)
Rule :
→ (VN:* tree VPP)
← (VN:* "VN:*/(np\\s_p)" VPP:np\\s_p)
Rule :
→ (PP:* P+D (NP NC))
← (PP:* P+D:*/n (NP:n NC:n))
Rule :
→ (SENT:* PP-MOD tree)
← (SENT:* PP-MOD:*/* SENT:*)
Rule :
→ (NP:* tree (COORD CC NP))
← (NP:* (:* NP:* (COORD:** "CC:(**)/np" NP:np)))
Rule :
→ (NP:* tree Srel)
← (NP:* NP:n Srel:n*)
Rule :
→ (NP-OBJ:* DET NC)
← (NP-OBJ:* DET:*/n NC:n)
Rule :
→ (NP:* tree NP)
← (NP:* NP:n NP:n*)
Rule :
→ (SENT:* tree ADV)
← (SENT:* SENT:* ADV:**)
Rule :
→ (VN:* ADV tree)
← (VN:* ADV:*/* VN:*)
Rule :
→ (NP-SUJ:* tree)
← (NP-SUJ:* NP:*)
Rule :
→ (VPpart:* tree)
← (VPpart:* SENT:*)
Rule :
→ (VN:* CLR tree)
← (VN:* CLR:cl_r VN:cl_r*)
Rule :
→ (VPpart:* VPP PP)
← (VPpart:* VPP:*/pp PP:pp)
Rule :
→ (Srel:* (NP-SUJ PROREL) tree)
← (Srel:* ("NP-SUJ:*/(np\\s)" "PROREL:*/(np\\s)")
SENT:np\\s)
Rule :
→ (NP:* tree NC)
← (NP:* NP:*/n NC:n)
Rule :
→ (SENT:* tree NP-OBJ)
← (SENT:* SENT:*/np NP-OBJ:np)

```
Rule :
→ (SENT:* NP-SUJ tree)
← (SENT:* NP-SUJ:np SENT:np\\*)
Rule :
→ (VN:* CLS-SUJ tree)
← (VN:* CLS-SUJ:np VN:np\\*)
Rule :
→ (SENT:* ADV tree)
← (SENT:* ADV:*/* SENT:*)
Rule :
→ (SENT:* VN NP-OBJ)
← (SENT:* VN:*/np NP-OBJ:np)
Rule : use-ponct
→ (SENT:* PONCT tree)
← (SENT:* PONCT:*/* SENT:*)
Rule :
→ (Ssub-MOD:* CS tree)
← (Ssub-MOD:* CS:*/s SENT:s)
Rule :
→ (PP-P_OBJ:* P NP)
← (PP-P_OBJ:* P:*/np NP:np)
Rule :
→ (VPinf-OBJ:* tree)
← (VPinf-OBJ:* SENT:*)
Rule : use-ponct
→ (SENT:* tree PONCT)
← (SENT:* SENT:* PONCT:*\\*)
Rule :
→ (NP:* tree AP)
← (NP:* NP:n AP:n\\*)
Rule :
→ (PP-MOD:* P (NP NC))
← (PP-MOD:* P:*/n (NP:n NC:n))
Rule :
→ (NP-MOD:* DET tree)
← (NP-MOD:* DET:*/n NP:n)
Rule :
→ (NP:* tree NP-MOD)
← (NP:* NP:* NP-MOD:*\\*)
Rule :
→ (Ssub-OBJ:* CS tree)
← (Ssub-OBJ:* CS:*/s SENT:s)
Rule : use-ponct
→ (NP-MOD:* PONCT NP PONCT)
← (NP-MOD:* (:*/pf "PONCT:(*/pf)/np" NP:np) PONCT:pf)
Rule :
→ (PP-A_OBJ:* P NP)
← (PP-A_OBJ:* P:*/np NP:np)
Rule :
→ (SENT:* NP-SUJ VN NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/np" NP-OBJ:np))
Rule :
→ (SENT:* tree VPinf-OBJ)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-OBJ:np\\s_i)
Rule :
→ (AP:* ADV ADJ)
← (AP:* ADV:*/* ADJ:*)
Rule :
→ (Sint-MOD:* tree)
← (Sint-MOD:* SENT:*)
Rule :
→ (SENT:* tree NP-MOD)
← (SENT:* SENT:* NP-MOD:*\\*)
Rule : use-ponct
→ (NP:* tree PONCT)
← (NP:* NP:* PONCT:*\\*)
Rule :
→ (PP-MOD:* tree)
← (PP-MOD:* PP:*)
Rule :
→ (SENT:* NP-MOD tree)
← (SENT:* NP-MOD:*/* SENT:*)
Rule :
→ (PP-DE_OBJ:* P NP)
← (PP-DE_OBJ:* P:*/np NP:np)
Rule :
→ (SENT:* tree VPinf-MOD)
← (SENT:* SENT:* VPinf-MOD:*\\*)
Rule :
→ (PP-MOD:* P+D NP)
← (PP-MOD:* P+D:*/n NP:n)
Rule :
→ (SENT:* tree Ssub-MOD)
← (SENT:* SENT:* Ssub-MOD:*\\*)
Rule :
→ (AP-ATS:* tree)
← (AP-ATS:* AP:*)
Rule :
→ (SENT:* NP-SUJ VN VPinf-OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_i)" VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* P VN)
← (SENT:* P:*/* VN:*)
Rule :
→ (Ssub:* CS tree)
← (Ssub:* CS:*/s SENT:s)
Rule : use-ponct
→ (SENT:* tree PONCT Sint-MOD)
← (SENT:* SENT:* (:*\\* "PONCT:(*\\*)/s" Sint-MOD:s))
Rule :
→ (SENT:* tree PP-A_OBJ)
← (SENT:* SENT:*/pp_a PP-A_OBJ:pp_a)
Rule :
→ (VN:* V ADV)
← (VN:* V:* ADV:*\\*)
Rule :
→ (NP-ATS:* DET tree)
← (NP-ATS:* DET:*/n NP:n)
Rule :
→ (NP-OBJ:* tree)
← (NP-OBJ:* NP:*)
Rule :
→ (VN:* tree ADV VPP)
← (VN:* "VN:*/(np\\s_p)"
    (:np\\s_p "ADV:(np\\s_p)/(np\\s_p)" VPP:np\\s_p))
Rule :
→ (Sint:* tree)
← (VPP:np\\s_p SENT:*)
Rule :
→ (NP:* tree ADV)
← (NP:* NP:* ADV:*\\*)
Rule :
→ (VPpart-MOD:* tree)
← (VPpart-MOD:* SENT:*)
Rule :
→ (SENT:* tree VPpart-MOD)
← (SENT:* SENT:* VPpart-MOD:*\\*)
Rule :
→ (SENT:* VN NP-SUJ)
← (SENT:* VN:*/np NP-SUJ:np)
Rule :
→ (SENT:* tree PP-P_OBJ)
← (SENT:* SENT:*/pp PP-P_OBJ:pp)
Rule :
→ (SENT:* tree PP-DE_OBJ)
← (SENT:* SENT:*/pp_de PP-DE_OBJ:pp_de)
```

Rule : use-ponct
→ (SENT (COORD CC tree) PONCT)
← (SENT:txt (COORD:s CC:s/s SENT:s) PONCT:s\\txt)
Rule :
→ (PP:* P NP (COORD CC PP))
← (PP:* (:* P:*/np NP:np)
(COORD:** "CC:(**)/*" PP:*))
Rule :
→ (VPpart:* tree PP)
← (VPpart:* VPpart:*/pp PP:pp)
Rule :
→ (NP-MOD:* tree)
← (NP-MOD:* NP:*)
Rule :
→ (VN:* CLO-OBJ tree)
← (VN:* "CLO-OBJ:*/(*/np)" VN:*/np)
Rule :
→ (VPinf:* P VN NP-OBJ)
← (VPinf:* "P:*/(np\\s_i)"
(:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
Rule :
→ (SENT:* tree Ssub-OBJ)
← (SENT:* SENT:*/cs Ssub-OBJ:cs)
Rule :
→ (NP:* ADV tree)
← (NP:* ADV:*/* NP:*)
Rule :
→ (VPinf-MOD:* P tree)
← (VPinf-MOD:* "P:*/(np\\s_i)" VPinf:np\\s_i)
Rule :
→ (NP:* tree VPinf)
← (NP:* NP:n VPinf:n*)
Rule :
→ (SENT:* Ssub-MOD tree)
← (SENT:* Ssub-MOD:*/* SENT:*)
Rule :
→ (VPinf-OBJ:* VN NP-OBJ)
← (VPinf-OBJ:* VN:*/np NP-OBJ:np)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-OBJ)
← (SENT:* NP-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/np" NP-OBJ:np)))
Rule :
→ (VPinf-MOD:* P VN NP-OBJ)
← (VPinf-MOD:* "P:*/(np\\s_i)"
(:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN AP-ATS)
← (SENT:* NP-SUJ:np
(:np* "VN:(np*)/(n\\n)" AP-ATS:n\\n))
Rule :
→ (VN:* tree VPP VPP)
← (VN:* "VN:*/(np\\s_p)"
(:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" VPP:np\\s_p))
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-OBJ)
← (SENT:* CLS-SUJ:np (:np* "VN:(np*)/np" NP-OBJ:np))

Rule :
→ (SENT:* VN VPinf-OBJ)
← (SENT:* "VN:*/(np\\s_i)" VPinf-OBJ:np\\s_i)
Rule : use-ponct
→ (SENT:* NP PONCT tree)
← (SENT:* NP:np (:np* "PONCT:(np*)/s" SENT:s))
Rule :
→ (VPinf:* tree)
← (SENT:s SENT:*)
Rule :
→ (NP-SUJ:* tree NPP)
← (NP-SUJ:* NP:n NPP:n*)
Rule :
→ (VPinf:* P tree)
← (VPinf:* "P:*/(np\\s_i)" VPinf:np\\s_i)
Rule :
→ (SENT:* VPpart-MOD tree)
← (SENT:* VPpart-MOD:*/s SENT:s)
Rule :
→ (Srel:* (NP-OBJ PROREL) tree)
← (Srel:* (NP-OBJ:*/s PROREL:*/s) SENT:s)
Rule : use-ponct
→ (SENT NP PONCT)
← (SENT:txt NP:np PONCT:np\\txt)
Rule :
→ (AP:* ADJ PP)
← (AP:* ADJ:*/pp PP:pp)
Rule :
→ (VPinf:* VN NP-OBJ)
← (VPinf:* VN:*/np NP-OBJ:np)
Rule :
→ (NP:* P tree)
← (NP:* P:*/n NP:n)
Rule :
→ (PP-A_OBJ:* P+D NP)
← (PP-A_OBJ:* P+D:*/n NP:n)
Rule :
→ (VPinf:* tree PP-MOD)
← (VPinf:* VPinf:* PP-MOD:**)
Rule :
→ (SENT NP)
← (SENT:np NP:np)
Rule : use-ponct
→ (SENT:* tree PONCT NP-MOD)
← (SENT:* SENT:* (:** "PONCT:(**)/np" NP-MOD:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) VPinf-OBJ)
← (SENT:* CLS-SUJ:np
(:np* "VN:(np*)/(np\\s_i)" VPinf-OBJ:np\\s_i))
Rule :
→ (VN:* CLR-OBJ tree)
← (VN:* "CLR-OBJ:*/(*/np)" VN:*/np)
Rule : use-ponct
→ (NP:* NC PONCT)
← (NP:* NC:n PONCT:n*)

Rule :
→ (SENT:* NP-SUJ VN ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np" "VN:(np\\s)/np" "ADV:((np\\s)/np)\\((np*)/np)") NP-OBJ:np))

Rule :
→ (SENT:* (VN tree VPP) NP-OBJ)
← (SENT:* (VN:* "VN:*/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/np" NP-OBJ:np)))
Rule :
→ (AP:* ADJ (COORD CC AP))
← (AP:* ADJ:* (COORD:** "CC:(**)/*" AP:*))

Rule :
→ (SENT:* NP-SUJ VN NP-ATS)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/np" NP-ATS:np))

Rule :
→ (SENT:* tree NP)
← (SENT:* SENT:*/np NP:np)

```
Rule :
→ (SENT:* VN ADV NP-OBJ)
← (SENT:* (:*/np VN:*/np "ADV:(*/np)\\(*/np)") NP-OBJ:np)
Rule :
→ (Srel:* tree)
← (Srel:* SENT:*)
Rule :
→ (PP:* P NP PP)
← (PP:* P:*/np (:np NP:np PP:np\\np))
Rule :
→ (AP-ATS:* ADV ADJ)
← (AP-ATS:* ADV:*/* ADJ:*)
Rule :
→ (SENT:* tree AP-ATS)
← (SENT:* "SENT:*/(n\\n)" AP-ATS:n\\n)
Rule :
→ (NP:* AP tree)
← (NP:* AP:*/n NP:n)
Rule :
→ (SENT:* tree NP-ATS)
← (SENT:* SENT:*/np NP-ATS:np)
Rule :
→ (NP:* tree (COORD CC tree))
← (NP:* NP:* (COORD:*\\* "CC:(*\\*)/*" NP:*))
Rule :
→ (VPinf:* P VN)
← (VPinf:* "P:*/(np\\s_i)" VN:np\\s_i)
Rule :
→ (PP:* P tree)
← (PP:* P:*/np NP:np)
Rule :
→ (VPinf-OBJ:* P VN NP-OBJ)
← (VPinf-OBJ:* "P:*/(np\\s_i)"
     (:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
Rule :
→ (PP-ATS:* P NP)
← (PP-ATS:* P:*/np NP:np)
Rule :
→ (SENT:* tree (COORD CC Sint))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/s" Sint:s))
Rule :
→ (NP-MOD:* tree PP)
← (NP-MOD:* NP:n PP:n\\*)
Rule : use-ponct
→ (NP:* tree PONCT NP PONCT)
← (NP:* NP:np (:np\\*
       (":(np\\*)/pf" "PONCT:((np\\*)/pf)/np" NP:np)
       PONCT:pf))
Rule :
→ (VPinf-A_OBJ:* tree)
← (PONCT:pf SENT:*)
Rule : use-ponct
→ (AP:* tree PONCT)
← (AP:* AP:* PONCT:*\\*)
Rule :
→ (NP-SUJ:* tree Srel)
← (NP-SUJ:* NP:n Srel:n\\*)
Rule :
→ (NP-MOD:* NC ADJ NC)
← (NP-MOD:* NC:*/n (:n ADJ:n/n NC:n))
Rule :
→ (AdP-MOD:* ADV ADV)
← (AdP-MOD:* ADV:*/* ADV:*)
Rule :
→ (PP:* tree (COORD CC PP))
← (PP:* PP:* (COORD:*\\* "CC:(*\\*)/*" PP:*))
Rule :
→ (VN:* tree CLS-SUJ)
← (VN:* VN:*/np CLS-SUJ:np)
Rule :
→ (SENT:* VPinf-MOD tree)
← (SENT:* VPinf-MOD:*/* SENT:*)
Rule :
→ (NP-SUJ:* tree NP)
← (NP-SUJ:* NP:n NP:n\\*)
Rule :
→ (SENT:* tree AdP-MOD)
← (SENT:* SENT:* AdP-MOD:*\\*)
Rule :
→ (PP:* P+D NP (COORD CC PP))
← (PP:* (:* P+D:*/n NP:n)
     (COORD:*\\* "CC:(*\\*)/*" PP:*))
Rule :
→ (Srel:* (NP-MOD PROREL) tree)
← (Srel:* (NP-MOD:*/s PROREL:*/s) SENT:s)
Rule :
→ (NP-SUJ:* tree VPpart)
← (NP-SUJ:* NP:n VPpart:n\\*)
Rule : use-ponct
→ (NP-MOD:* tree PONCT)
← (NP-MOD:* NP:* PONCT:*\\*)
Rule :
→ (SENT:* tree NP-SUJ)
← (SENT:* SENT:*/np NP-SUJ:np)
Rule :
→ (VPpart:* ADV tree)
← (VPpart:* ADV:*/* VPpart:*)
Rule :
→ (VPpart:* tree NP)
← (VPpart:* VPpart:*/np NP:np)
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* ("VN:(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (NP:* PREF tree)
← (NP:* PREF:*/n NP:n)
Rule :
→ (VPinf-OBJ:* VN PP-MOD)
← (VPinf-OBJ:* VN:* PP-MOD:*\\*)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-P_OBJ:pp)))
Rule :
→ (PP-MOD:* P+D (NP NC))
← (PP-MOD:* P+D:*/n (NP:n NC:n))
Rule :
→ (VPpart:* tree PP-MOD)
← (VPpart:* VPpart:* PP-MOD:*\\*)
Rule :
→ (VN:* (CLO en) tree)
← (VN:* (CLO:*/* en:*/*) VN:*)
Rule :
→ (VPinf:* tree NP-OBJ)
← (VPinf:* VPinf:*/np NP-OBJ:np)
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* "VN:(np\\*)/np" NP-ATS:np))
```

Rule :
→ (NP-OBJ:* tree PP)
← (NP-OBJ:* NP:n PP:n*)
Rule :
→ (AdP:* ADV ADV)
← (AdP:* ADV:*/* ADV:*)
Rule :
→ (NP:* ADV DET tree)
← (NP:* ADV:*/* (:* DET:*/n NP:n))
Rule :
→ (NP:* tree ADJ)
← (NP:* NP:n ADJ:n*)
Rule :
→ (VN:* (CLO y) tree)
← (VN:* (CLO:pp y:pp) VN:pp*)
Rule :
→ (Srel:* (PP-DE_OBJ PROREL) tree)
← (Srel:* (PP-DE_OBJ:*/s PROREL:*/s) SENT:s)
Rule : use-ponct
→ (SENT:* tree PONCT (COORD CC Sint))
← (SENT:* SENT:s (:s* "PONCT:(s*)/(s*)"
(COORD:s* "CC:(s*)/s" Sint:s)))
Rule :
→ (PP-DE_OBJ:* P+D NP)
← (Sint:s P+D:*/n NP:n)
Rule :
→ (NP-OBJ:* tree Srel)
← (NP-OBJ:* NP:n Srel:n*)
Rule : use-ponct
→ (SENT:* tree (Sint-MOD PONCT tree))
← (SENT:* SENT:* (Sint-MOD:** "PONCT:(**)/s" SENT:s))
Rule : use-ponct
→ (NP:* tree (PONCT -LBR-) NP (PONCT -RBR-))
← (NP:* NP:np (:np* (":(np*)/pf"
("PONCT:((np*)/pf)/np"
"-LBR-:((np*)/pf)/np")
NP:np)
(PONCT:pf -RBR-:pf)))

Rule :
→ (SENT:* NP-SUJ PP-MOD VN)
← (SENT:* NP-SUJ:np
(:np* "PP-MOD:(np*)/(np*)" VN:np*))
Rule :
→ (VPinf-OBJ:* tree PP-MOD)
← (VPinf-OBJ:* SENT:* PP-MOD:**)
Rule :
→ (VPinf:* VN PP-MOD)
← (VPinf:* VN:* PP-MOD:**)
Rule :
→ (PP-DE_OBJ:* tree)
← (PP-DE_OBJ:* PP:*)
Rule :
→ (VN:* tree ADV)
← (VN:* VN:* ADV:**)
Rule :
→ (Srel-MOD:* tree)
← (Srel-MOD:* SENT:*)
Rule :
→ (SENT:* VN AP-ATS)
← (SENT:* "VN:*/(n\\n)" AP-ATS:n\\n)
Rule :
→ (NP:* ADJ DET tree)
← (NP:* ADJ:*/* (:* DET:*/n NP:n))
Rule :
→ (SENT:* tree VPinf-A_OBJ)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-A_OBJ:np\\s_i)
Rule :
→ (NP-SUJ:* tree PP)
← (NP-SUJ:* NP:n PP:n*)
Rule :
→ (SENT:* tree Srel-MOD)
← (SENT:* SENT:* Srel-MOD:**)

Rule :
→ (SENT:* NP-SUJ VN ADV VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np*
(":(np*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "ADV:((np\\s)/(np\\s_i))\\((np*)/(np\\s_i))") VPinf-OBJ:np\\s_i))

Rule :
→ (VPinf-A_OBJ:* P VN NP-OBJ)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)"
(:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/pp" PP-P_OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) NP-OBJ)
← (SENT:* CLS-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/np" NP-OBJ:np)))
Rule :
→ (PP:* ADV tree)
← (NP-OBJ:np ADV:*/* PP:*)
Rule :
→ (SENT:* tree PP-ATS)
← (SENT:* SENT:*/pp PP-ATS:pp)
Rule :
→ (SENT:* P tree)
← (SENT:* P:*/* SENT:*)
Rule :
→ (PP-MOD:* P NP (COORD CC PP))
← (PP-MOD:* (:* P:*/np NP:np)
(COORD:** "CC:(**)/*" PP:*))

Rule :
→ (AP-ATS:* ADJ PP)
← (AP-ATS:* ADJ:* PP:**)
Rule :
→ (NP-ATS:* DET NC)
← (NP-ATS:* DET:*/n NC:n)
Rule :
→ (NP:* tree Ssub)
← (NP:* NP:n Ssub:n*)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/pp_de"
PP-DE_OBJ:pp_de)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/pp_a" PP-A_OBJ:pp_a)))
Rule :
→ (VPpart-MOD:* VPP PP)
← (PP-A_OBJ:pp_a VPP:*/pp PP:pp)
Rule :
→ (SENT:* NP-SUJ VN PP-DE_OBJ)
← (SENT:* NP-SUJ:np
(:np* "VN:(np*)/pp_de" PP-DE_OBJ:pp_de))

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) AP-ATS)
← (SENT:* CLS-SUJ:np
     (:np\\* "VN:(np\\*)/(n\\n)" AP-ATS:n\\n))
Rule :
→ (SENT:* VN NP-ATS)
← (SENT:* VN:*/np NP-ATS:np)
Rule :
→ (VPinf:* tree PP-P_OBJ)
← (VPinf:* VPinf:*/pp PP-P_OBJ:pp)
Rule :
→ (VPinf-DE_OBJ:* P tree)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)" SENT:np\\s_i)
Rule :
→ (SENT:* tree VPinf-DE_OBJ)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-DE_OBJ:np\\s_i)
Rule :
→ (PP-A_OBJ:* P+D (NP NC))
← (PP-A_OBJ:* P+D:*/n (NP:n NC:n))
Rule :
→ (NP-SUJ:* ADJ DET tree)
← (NP-SUJ:* ADJ:*/* (:* DET:*/n NP:n))
Rule :
→ (VPinf:* VN ADV)
← (VPinf:* VN:* ADV:*\\*)
Rule :
→ (PP:* P+D tree)
← (PP:* P+D:*/n NP:n)
Rule :
→ (VN:* (CLO-A_OBJ lui) tree)
← (VN:* ("CLO-A_OBJ:*/(*/pp)" "lui:*/(*/pp)") VN:*/pp)
Rule :
→ (VPpart-MOD:* tree PP-MOD)
← (VPpart-MOD:* VPpart:* PP-MOD:*\\*)
Rule :
→ (SENT:* NP-SUJ VN Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/cs" Ssub-OBJ:cs))
Rule :
→ (SENT:* NP-SUJ VN PP-A_OBJ)
← (SENT:* NP-SUJ:np
     (:np\\* "VN:(np\\*)/pp_a" PP-A_OBJ:pp_a))
Rule :
→ (NP-MOD:* tree Srel)
← (NP-MOD:* NP:n Srel:n\\*)
Rule :
→ (AP:* tree PP)
← (AP:* AP:* PP:*\\*)
Rule :
→ (VPpart:* ADV VPP)
← (VPpart:* ADV:*/* VPP:*)
Rule :
→ (PP:* P PP)
← (PP:* "P:*/(n\\n)" PP:n\\n)
Rule :
→ (VN:* CLO tree)
← (VN:* CLO:np VN:np\\*)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
         VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* (VN CLS-SUJ tree) Ssub-OBJ)
← (SENT:* CLS-SUJ:np
     (:np\\* "VN:(np\\*)/cs" Ssub-OBJ:cs))
Rule :
→ (VPinf:* VN PP-A_OBJ)
← (VPinf:* VN:*/pp_a PP-A_OBJ:pp_a)
Rule :
→ (SENT:* Sint-MOD tree)
← (SENT:* Sint-MOD:*/* SENT:*)
Rule :
→ (VPinf-OBJ:* VN NP-OBJ PP-MOD)
← (VPinf-OBJ:* (:* VN:*/np NP-OBJ:np) PP-MOD:*\\*)
Rule :
→ (NP-ATS:* tree)
← (NP-ATS:* NP:*)
Rule :
→ (VN:* (CLO-DE_OBJ en) tree)
← (VN:* ("CLO-DE_OBJ:*/(*/pp)" "en:*/(*/pp)") VN:*/pp)
Rule :
→ (SENT:* (VN tree VPP) PP-P_OBJ)
← (SENT:* (VN:* "VN:*/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-P_OBJ:pp)))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV VPinf-OBJ)
← (SENT:* CLS-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "ADV:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))") VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* NP-SUJ VN ADV AP-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "ADV:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* tree (COORD CC tree PP-MOD))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* SENT:* PP-MOD:*\\*)))
Rule :
→ (SENT:* VN NP-OBJ PP-A_OBJ)
← (SENT:* (:*/pp_a "VN:(*/pp_a)/np" NP-OBJ:np)
     PP-A_OBJ:pp_a)
Rule :
→ (NP:* NC ADJ NC)
← (NP:* NC:*/n (:n ADJ:n/n NC:n))
Rule : use-ponct
→ (VPpart:* tree PONCT)
← (VPpart:* VPpart:* PONCT:*\\*)
Rule :
→ (SENT:* tree (COORD CC Ssub))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/cs" Ssub:cs))
Rule :
→ (SENT tree NP-MOD)
← (SENT:s SENT:s NP-MOD:s\\s)
Rule :
→ (VPinf-DE_OBJ:* P VN NP-OBJ)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)"
     (:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
Rule :
→ (SENT:* tree VPinf-ATS)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-ATS:np\\s_i)
Rule : use-ponct
→ (VN:* tree PONCT)
← (VN:* VN:* PONCT:*\\*)
```

```
Rule :
→ (SENT:* VN VPinf-A_OBJ)
← (SENT:* "VN:*/(np\\s_i)" VPinf-A_OBJ:np\\s_i)
Rule :
→ (NP-SUJ:* tree (COORD CC NP))
← (NP-SUJ:* (:* NP:* (COORD:*\\* "CC:(*\\*)/np" NP:np)))
Rule :
→ (VPpart:* VPP ADV)
← (NP:np VPP:* ADV:*\\*)
Rule :
→ (NP-OBJ:* tree NP)
← (NP-OBJ:* NP:n NP:n\\*)
Rule :
→ (Srel:* tree PP-MOD)
← (Srel:* SENT:* PP-MOD:*\\*)
Rule : use-ponct
→ (NP:* PONCT tree)
← (NP:* PONCT:*/* NP:*)
Rule :
→ (NP-OBJ:* ADV tree)
← (NP-OBJ:* ADV:*/n NP:n)
Rule :
→ (SENT:* tree AP-ATO)
← (SENT:* "SENT:*/(n\\n)" AP-ATO:n\\n)
Rule :
→ (PP-MOD:* ADV P NP)
← (PP-MOD:* ADV:*/* (:* P:*/np NP:np))
Rule :
→ (NP:* NC DET)
← (NP:* NC:n DET:n\\*)
Rule :
→ (NP:* tree PP-MOD)
← (NP:* NP:* PP-MOD:*\\*)
Rule :
→ (NP-ATS:* tree PP)
← (NP-ATS:* NP:n PP:n\\*)
Rule :
→ (VN:* tree VINF)
← (VN:* "VN:*/(np\\s_i)" VINF:np\\s_i)
Rule :
→ (PP:* P tree NP)
← (PP:* P:*/np (:np AP:np/np NP:np))
Rule :
→ (VPpart:* P tree)
← (VPpart:* "P:*/(np\\s)" VPpart:np\\s)
Rule :
→ (SENT:* tree (COORD CC PP))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/(s\\s)" PP:s\\s))
Rule :
→ (NP-OBJ:* tree (COORD CC NP))
← (NP-OBJ:* (:* NP:* (COORD:*\\* "CC:(*\\*)/np" NP:np)))
Rule :
→ (VPpart:* VPP VPinf)
← (NP:np VPP:np\\s_p "VPinf:(np\\s_p)\\*")
Rule :
→ (VPinf-OBJ:* tree VPinf-MOD)
← (VPinf-OBJ:* SENT:* VPinf-MOD:*\\*)
Rule :
→ (NP:* NPP NP)
← (NP:* NPP:*/np NP:np)
Rule :
→ (PP-P_OBJ:* tree)
← (PP-P_OBJ:* PP:*)
Rule :
→ (VPinf-OBJ:* P VN)
← (VPinf-OBJ:* "P:*/(np\\s_i)" VN:np\\s_i)
Rule :
→ (SENT:* NP-SUJ VN PP-ATS)
← (SENT:* NP-SUJ:np
     (:np\\* "VN:(np\\*)/(n\\n)" PP-ATS:n\\n))
Rule :
→ (Srel:* (PP-MOD P (NP PROREL)) tree)
← (Srel:* (PP-MOD:* P:pp/np
       ("NP:(pp/np)\\*" "PROREL:((pp/np)\\*)/s" SENT:s)))
Rule :
→ (SENT:* tree (COORD CC VN NP-OBJ))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* VN:*/np NP-OBJ:np)))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-DE_OBJ)
← (SENT:* CLS-SUJ:np
     (:np\\* "VN:(np\\*)/pp_de" PP-DE_OBJ:pp_de))
Rule :
→ (PP-P_OBJ:* P (NP NC))
← (PP-P_OBJ:* P:*/n (NP:n NC:n))
Rule :
→ (PP-A_OBJ:* tree)
← (NC:n PP:*)
Rule :
→ (NP-MOD:* ADJ tree)
← (NP-MOD:* ADJ:*/n NP:n)
Rule :
→ (VPinf:* tree (COORD CC VPinf))
← (VPinf:* VPinf:*
     (COORD:*\\* "CC:(*\\*)/(np\\s_i)" VPinf:np\\s_i))
Rule :
→ (SENT:* tree (COORD CC tree NP-OBJ))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* SENT:*/np NP-OBJ:np)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) AP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/(n\\n)" AP-ATS:n\\n)))
Rule :
→ (SENT NP PP)
← (AP-ATS:n\\n NP:np PP:np\\np)
Rule :
→ (SENT:* tree Sint-MOD)
← (SENT:* SENT:* Sint-MOD:*\\*)
Rule :
→ (SENT:* tree Ssub)
← (SENT:* SENT:*/cs Ssub:cs)
Rule :
→ (VPinf-OBJ:* tree (COORD CC VPinf))
← (VPinf-OBJ:* VPinf:*
     (COORD:*\\* "CC:(*\\*)/*" VPinf:*))
Rule :
→ (PP-DE_OBJ:* P (NP NC))
← (PP-DE_OBJ:* P:*/n (NP:n NC:n))
Rule :
→ (VN:* VINF VINF)
← (VN:* "VINF:*/(np\\s_i)" VINF:np\\s_i)
Rule :
→ (VN:* V CLS-MOD)
← (VN:* V:* CLS-MOD:*\\*)
Rule :
→ (SENT:* tree AP-MOD)
← (SENT:* SENT:* AP-MOD:*\\*)
Rule :
→ (SENT:* VN VPinf-DE_OBJ)
← (SENT:* "VN:*/(np\\s_i)" VPinf-DE_OBJ:np\\s_i)
Rule :
→ (NP-OBJ:* tree VPpart)
← (NP-OBJ:* NP:n VPpart:n\\*)
```

```
Rule :
→ (Ssub-ATS:* CS tree)
← (Ssub-ATS:* CS:*/s SENT:s)

Rule : use-ponct
→ (PP:* PONCT PP PONCT)
← (PP:* (:*/pf "PONCT:(*/pf)/(n\\n)" PP:n\\n) PONCT:pf)

Rule :
→ (SENT:* AP-MOD tree)
← (SENT:* AP-MOD:*/* SENT:*)

Rule :
→ (SENT:* NP-SUJ VN VPinf-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_i)" VPinf-ATS:np\\s_i))

Rule :
→ (SENT:* NP-SUJ VN PP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "PP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))

Rule :
→ (VPinf-OBJ:* VN PP-A_OBJ)
← (VPinf-OBJ:* VN:*/pp_a PP-A_OBJ:pp_a)

Rule :
→ (VPinf-MOD:* P VN)
← (VPinf-MOD:* "P:*/(np\\s_i)" VN:np\\s_i)

Rule :
→ (NP:* ET tree)
← (NP:* ET:*/n NP:n)

Rule :
→ (VPinf:* tree VPinf-OBJ)
← (VPinf:* "VPinf:*/(np\\s_i)" VPinf-OBJ:np\\s_i)

Rule :
→ (SENT:* tree (COORD CC Srel))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/s" Srel:s))

Rule :
→ (AP-MOD:* tree)
← (Srel:s AP:*)

Rule :
→ (SENT:* tree (COORD CC VPinf))
← (SENT:* SENT:*
    (COORD:*\\* "CC:(*\\*)/(np\\s_i)" VPinf:np\\s_i))

Rule :
→ (VPinf-OBJ:* VN Ssub-OBJ)
← (VPinf-OBJ:* VN:*/cs Ssub-OBJ:cs)

Rule :
→ (AP-ATO:* tree)
← (AP-ATO:* AP:*)

Rule :
→ (VN:* tree VPP)
← (VN:* "VN:*/(np\\s_p)" VPP:np\\s_p)

Rule :
→ (VPinf:* ADV tree)
← (VPinf:* ADV:*/* VPinf:*)

Rule :
→ (AP:* ADJ VPinf)
← (AP:* ADJ:* VPinf:*\\*)

Rule :
→ (SENT (NP NC))
← (SENT:n (NP:n NC:n))

Rule :
→ (NP:* NC AP)
← (NP:* NC:n AP:n\\*)

Rule :
→ (NP-OBJ:* ADJ DET tree)
← (NP-OBJ:* ADJ:*/* (:* DET:*/n NP:n))

Rule :
→ (Srel:* PP-MOD tree)
← (Srel:* PP-MOD:*/* SENT:*)

Rule :
→ (AP-ATS:* ADJ VPinf)
← (AP-ATS:* ADJ:* VPinf:*\\*)

Rule :
→ (NP:* tree (COORD CC PP))
← (NP:* NP:* (COORD:*\\* "CC:(*\\*)/(n\\n)" PP:n\\n))

Rule :
→ (SENT:* AdP-MOD tree)
← (SENT:* AdP-MOD:*/* SENT:*)

Rule :
→ (SENT:* NP-SUJ VN ADV NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))

Rule :
→ (SENT:* tree VPinf)
← (SENT:* "SENT:*/(np\\s_i)" VPinf:np\\s_i)

Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-ATS)
← (SENT:* CLS-SUJ:np (VN:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))

Rule :
→ (VPpart-MOD:* tree PP)
← (VPpart-MOD:* VPpart:*/pp PP:pp)

Rule :
→ (VPinf-OBJ:* P VN NP-OBJ PP-MOD)
← (VPinf-OBJ:* (:* "P:*/(np\\s_i)"
      (:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))
    PP-MOD:*\\*)

Rule :
→ (VPpart-MOD:* P VN NP-OBJ)
← (VPpart-MOD:* "P:*/(np\\s_p)"
    (:np\\s_p "VN:(np\\s_p)/np" NP-OBJ:np))

Rule :
→ (PP-OBJ:* P NP)
← (PP-OBJ:* P:*/np NP:np)

Rule :
→ (NP:* tree (COORD CC ADV NP))
← (NP:* NP:n
    (COORD:n\\* "CC:(n\\*)/np" (:np ADV:np/np NP:np)))

Rule :
→ (VPinf:* tree NP-MOD)
← (NP:np VPinf:* NP-MOD:*\\*)

Rule :
→ (VPinf-ATS:* P tree)
← (VPinf-ATS:* "P:*/(np\\s_i)" SENT:np\\s_i)

Rule :
→ (AP:* ADJ tree)
← (AP:* ADJ:n\\n "AP:(n\\n)\\*")
```

```
Rule :
→ (NP:* tree VPpart-MOD)
← (NP:* NP:* VPpart-MOD:*\\*)
Rule : use-ponct
→ (NP:* tree (PONCT -LBR-) tree (PONCT -RBR-))
← (NP:* NP:np (:np\\* (":(np\\*)/pf"
        ("PONCT:((np\\*)/pf)/np"
          "-LBR-:((np\\*)/pf)/np")
        NP:np)
      (PONCT:pf -RBR-:pf)))
Rule : use-ponct
→ (VPinf:* tree PONCT)
← (VPinf:* VPinf:* PONCT:*\\*)
```

```
Rule :
→ (AP:* tree NP)
← (AP:* AP:*/np NP:np)
Rule :
→ (VPinf:* tree PP-A_OBJ)
← (VPinf:* VPinf:*/pp_a PP-A_OBJ:pp_a)
Rule :
→ (COORD:n CC tree)
← (COORD:n CC:n/np NP:np)
Rule :
→ (SENT:* (VN tree VPP) VPinf-OBJ)
← (SENT:* (VN:* "VN:*/(np\\s_p)" (VPP:np\\s_p
        "VPP:(np\\s_p)/(np\\s_i)" VPinf-OBJ:np\\s_i)))
```

```
Rule :
→ (VPinf-MOD:* P VN NP-OBJ PP-MOD)
← (VPinf-MOD:* "P:*/(np\\s_i)" (:np\\s_i (:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np) "PP-MOD:(np\\s_i)\\(np\\s_i)"))
```

```
Rule :
→ (AP:* tree Ssub)
← (AP:* AP:*/cs Ssub:cs)
Rule :
→ (VPinf-OBJ:* VN PP-DE_OBJ)
← (VPinf-OBJ:* VN:*/pp_de PP-DE_OBJ:pp_de)
Rule :
→ (SENT:* tree (COORD CC VPpart))
← (SENT:* SENT:*
    (COORD:*\\* "CC:(*\\*)/(np\\s_p)" VPpart:np\\s_p))
Rule :
→ (SENT:* tree (COORD CC NP-SUJ tree))
← (SENT:* SENT:*
    (COORD:*\\* "CC:(*\\*)/*" (:* NP-SUJ:np SENT:np\\*)))
Rule :
→ (PP:* P ADV NP)
← (PP:* P:*/np (:np ADV:np/np NP:np))
Rule :
→ (NP-OBJ:* tree AP)
← (NP-OBJ:* NP:n AP:n\\*)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p "VPP:(np\\s_p)/cs" Ssub-OBJ:cs)))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-A_OBJ)
← (SENT:* CLS-SUJ:np
    (:np\\* "VN:(np\\*)/pp_a" PP-A_OBJ:pp_a))
Rule :
→ (SENT:* NP-SUJ VN VPinf-A_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_i)" VPinf-A_OBJ:np\\s_i))
Rule : use-ponct
→ (NP:* tree PONCT PP PONCT)
← (NP:* NP:np (:np\\*
      (":(np\\*)/pf" "PONCT:((np\\*)/pf)/(n\\n)" PP:n\\n)
      PONCT:pf))
Rule :
→ (NP-SUJ:* tree AP)
← (NP-SUJ:* NP:n AP:n\\*)
Rule :
→ (NP-MOD:* NC NC)
← (NP-MOD:* NC:n NC:n\\*)
Rule :
→ (SENT:* tree PP-OBJ)
← (SENT:* SENT:*/pp PP-OBJ:pp)
Rule :
→ (SENT:* NP-SUJ VN VPinf-DE_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_i)" VPinf-DE_OBJ:np\\s_i))
```

```
Rule :
→ (VN:* (CLO-A_OBJ leur) tree)
← (VN:* ("CLO-A_OBJ:*/(*/pp)" "leur:*/(*/pp)") VN:*/pp)
Rule :
→ (VN:* CLR-A_OBJ tree)
← (VN:* "CLR-A_OBJ:*/(*/pp_a)" VN:*/pp_a)
Rule :
→ (VPinf-ATO:* tree)
← (VPinf-ATO:* SENT:*)
Rule :
→ (NP-OBJ:* ADV DET tree)
← (NP-OBJ:* ADV:*/* (:* DET:*/n NP:n))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
        (VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-P_OBJ:pp))))
Rule :
→ (Srel:* NP-OBJ tree)
← (Srel:* NP-OBJ:np SENT:np\\*)
Rule :
→ (Ssub-OBJ:* tree PP-MOD)
← (Ssub-OBJ:* Ssub-OBJ:* PP-MOD:*\\*)
Rule :
→ (PP-P_OBJ:* P+D NP)
← (PP-P_OBJ:* P+D:*/n NP:n)
Rule :
→ (NP-SUJ:* NC NPP NPP)
← (NP-SUJ:* NC:*/np (:np NPP:np/np NPP:np))
Rule :
→ (VPpart-MOD:* ADV tree)
← (VPpart-MOD:* ADV:*/* VPpart:*)
Rule :
→ (AP:* PREF ADJ)
← (AP:* PREF:*/* ADJ:*)
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) VPinf-OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
        VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ VN NP-OBJ PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/pp_a" "VN:((np\\*)/pp_a)/np" NP-OBJ:np)
      PP-A_OBJ:pp_a))
Rule :
→ (NP:* NC (COORD CC PP))
← (NP:* NC:n (COORD:n\\* "CC:(n\\*)/(n\\n)" PP:n\\n))
```

Rule :
→ (VPinf:* tree VPinf-MOD)
← (VPinf:* VPinf:* VPinf-MOD:**)

Rule :
→ (AP:* ADJ ADJ)
← (AP:* ADJ:*/* ADJ:*)

Rule :
→ (AP:* tree ADV)
← (AP:* AP:* ADV:**)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) VPinf-A_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
VPinf-A_OBJ:np\\s_i)))

Rule :
→ (PP:* tree (COORD CC ADV PP))
← (PP:* PP:*
(COORD:** "CC:(**)/*" (:* ADV:*/* PP:*)))

Rule :
→ (NP:* tree ET)
← (NP:* NP:*/np ET:np)

Rule :
→ (SENT:* NP-SUJ PP-MOD VN NP-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np" "PP-MOD:((np*)/np)/((np*)/np)" "VN:(np*)/np") NP-OBJ:np))

Rule :
→ (PP:* P VPpart)
← (PP:* "P:*/(np\\s_p)" VPpart:np\\s_p)

Rule :
→ (VN:* CLS-MOD V)
← (VN:* CLS-MOD:*/* V:*)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-ATS)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/np" NP-ATS:np)))

Rule :
→ (PP:* P ADJ (NP tree))
← (PP:* P:*/np (:np ADJ:np/n (NP:n NP:n)))

Rule :
→ (NP:* tree AdP)
← (NP:* NP:* AdP:**)

Rule :
→ (VN:* CLO-A_OBJ tree)
← (VN:* "CLO-A_OBJ:*/(*/pp)" VN:*/pp)

Rule : use-ponct
→ (SENT:* tree PONCT Sint)
← (SENT:* SENT:* (:** "PONCT:(**)/s" Sint:s))

Rule :
→ (PP-DE_OBJ:* P+D (NP NC))
← (PP-DE_OBJ:* P+D:*/n (NP:n NC:n))

Rule :
→ (VPinf:* VN Ssub-OBJ)
← (VPinf:* VN:*/cs Ssub-OBJ:cs)

Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV AP-ATS)
← (SENT:* CLS-SUJ:np
(:np* (":(np*)/(n\\n)" "VN:(np*)/(n\\n)" "ADV:((np*)/(n\\n))\\((np*)/(n\\n))") AP-ATS:n\\n))

Rule :
→ (SENT:* VPP AP)
← (SENT:* "VPP:*/(n\\n)" AP:n\\n)

Rule :
→ (AP:* tree (COORD CC AP))
← (AP:* AP:* (COORD:** "CC:(**)/*" AP:*))

Rule : use-ponct
→ (PP:* PONCT PP-MOD PONCT)
← (PP:* (:*/pf "PONCT:(*/pf)/*" PP-MOD:*) PONCT:pf)

Rule :
→ (NP:* tree PP (COORD CC PP))
← (NP:* NP:n (:n* PP:n\\n ("COORD:(n\\n)\\(n*)" "CC:((n\\n)\\(n*))/(n\\n)" PP:n\\n)))

Rule :
→ (VN:* tree (AdP ADV ADV))
← (VN:* VN:* (AdP:** "ADV:(**)/(**)" ADV:**))

Rule :
→ (VPinf-OBJ:* P VN PP-MOD)
← (VPinf-OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "PP-MOD:(np\\s_i)\\(np\\s_i)"))

Rule :
→ (VPinf-A_OBJ:* P VN)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)" VN:np\\s_i)

Rule :
→ (SENT:* tree VPinf-ATO)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-ATO:np\\s_i)

Rule :
→ (VPpart:* tree NP-MOD)
← (VPpart:* VPpart:* NP-MOD:**)

Rule :
→ (VPinf-ATS:* P VN NP-OBJ)
← (VPinf-ATS:* "P:*/(np\\s_i)"
(:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np))

Rule :
→ (VPinf-DE_OBJ:* tree)
← (VPinf-DE_OBJ:* SENT:*)

Rule :
→ (AP:* ADJ (COORD CC tree))
← (AP:* ADJ:* (COORD:** "CC:(**)/*" AP:*))

Rule :
→ (SENT:* NP-SUJ VN ADV PP-DE_OBJ)
← (SENT:* NP-SUJ:np
(:np* (":(np*)/pp_de" "VN:(np\\s)/pp_de" "ADV:((np\\s)/pp_de)\\((np*)/pp_de)") PP-DE_OBJ:pp_de))

Rule :
→ (NP:* NC (COORD CC tree))
← (NP:* NC:n (COORD:n* "CC:(n*)/np" NP:np))
Rule :
→ (NP-ATS:* tree Srel)
← (NP-ATS:* NP:n Srel:n*)
Rule :
→ (VPinf-ATS:* tree)
← (VPinf-ATS:* SENT:*)
Rule :
→ (SENT:* PP-A_OBJ tree)
← (SENT:* PP-A_OBJ:pp_a SENT:pp_a*)
Rule :
→ (PP-MOD:* P ADV)
← (PP-MOD:* "P:*/(n\\n)" ADV:n\\n)
Rule :
→ (PP:* P NP (COORD CC NP))
← (PP:* P:*/np
(:np NP:np (COORD:np\\np "CC:(np\\np)/np" NP:np)))
Rule :
→ (PP:* tree NP-MOD)
← (NP:np PP:* NP-MOD:**)
Rule :
→ (SENT:* NP-SUJ ADV VN)
← (SENT:* NP-SUJ:np
(:np* "ADV:(np*)/(np*)" VN:np*))
Rule : use-ponct
→ (SENT:* VPpart PONCT tree)
← (SENT:* VPpart:np\\s_p
(":(np\\s_p)*" "PONCT:((np\\s_p)*)/*" SENT:*))

Rule :
→ (PP-ATS:* P (NP NC))
← (PP-ATS:* P:*/n (NP:n NC:n))
Rule : use-ponct
→ (PP:* tree PONCT NP PONCT)
← (PP:* PP:* (:**
(":(**)/pf" "PONCT:((**)/pf)/np" NP:np)
PONCT:pf))
Rule :
→ (NP:* ET ET)
← (NP:* ET:*/n ET:n)
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) PP-P_OBJ)
← (SENT:* CLS-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-P_OBJ:pp)))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-P_OBJ)
← (SENT:* CLS-SUJ:np
(:np* "VN:(np*)/pp" PP-P_OBJ:pp))
Rule :
→ (NP:* tree Srel-MOD)
← (NP:* NP:* Srel-MOD:**)
Rule :
→ (AP:* ADV tree)
← (AP:* ADV:*/* AP:*)
Rule :
→ (AP:* AdP ADJ)
← (AP:* "AdP:*/(n/n)" ADJ:n/n)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "PP-MOD:((np\\s_p)/np)\\((np\\s_p)/np)") NP-OBJ:np)))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np" "VN:(np\\s)/np" "NP-MOD:((np\\s)/np)\\((np*)/np)") NP-OBJ:np))

Rule :
→ (NP:* tree (COORD CC Srel))
← (NP:* NP:np (COORD:np* "CC:(np*)/s" Srel:s))

Rule :
→ (NP-MOD:* tree AP)
← (NP-MOD:* NP:n AP:n*)

Rule :
→ (VN:* CLS tree)
← (VN:* CLS:np VN:np*)
Rule :
→ (VPpart:* tree (COORD CC VPpart))
← (VPpart:* VPpart:*
(COORD:** "CC:(**)/(np\\s_p)" VPpart:np\\s_p))

Rule :
→ (VPinf-A_OBJ:* P VN NP-OBJ PP-MOD)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)" (VN:np\\s_i (VN:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np) "PP-MOD:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (SENT:* NP-SUJ VN ADV PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV Ssub-OBJ)
← (SENT:* CLS-SUJ:np (:np* (":(np*)/cs" "VN:(np\\s)/cs" "ADV:((np\\s)/cs)\\((np*)/cs)") Ssub-OBJ:cs))

Rule :
→ (SENT:* NP-SUJ PONCT NP-SUJ VN)
← (SENT:* (:np NP-SUJ:np
(:np\\np "PONCT:(np\\np)/np" NP-SUJ:np))
VN:np*)

Rule :
→ (SENT:* tree Ssub-ATS)
← (SENT:* SENT:*/cs Ssub-ATS:cs)

Rule : use-ponct
→ (NP:* tree PONCT AP PONCT)
← (NP:* NP:np (:np*
(":(np*)/pf" "PONCT:((np*)/pf)/(n\\n)" AP:n\\n)
PONCT:pf))
Rule :
→ (VPpart:* tree ADV)
← (VPpart:* VPpart:* ADV:**)
Rule :
→ (AP-MOD:* ADV ADJ)
← (AP-MOD:* ADV:*/* ADJ:*)

Rule :
→ (AP-ATS:* ADV ADJ Ssub)
← (AP-ATS:* (:*/cs "ADV:(*/cs)/(n\\n)" ADJ:n\\n) Ssub:cs)

Rule :
→ (SENT:* VN AP-ATS VPinf-OBJ)
← (SENT:* (":*/(np\\s_i)" "VN:(*/(np\\s_i))/(n\\n)"
AP-ATS:n\\n)
VPinf-OBJ:np\\s_i)

Rule :
→ (PP:* P+D NP PP)
← (PP:* P+D:*/n (:n NP:n PP:n\\n))

Rule : use-ponct
→ (NP:* tree (COORD CC NP) PONCT)
← (NP:* (:n NP:n (COORD:n\\n "CC:(n\\n)/np" NP:np))
PONCT:n*)

Rule :
→ (VPinf-DE_OBJ:* P VN)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)" VN:np\\s_i)

Rule :
→ (VPinf-OBJ:* tree NP-MOD)
← (VPinf-OBJ:* SENT:* NP-MOD:**)

Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-ATS)
← (SENT:* CLS-SUJ:np
(:np* "VN:(np*)/(n\\n)" PP-ATS:n\\n))

Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-DE_OBJ)
← (SENT:* CLS-SUJ:np
(:np* (":(np*)/pp_de" "VN:(np\\s)/pp_de" "ADV:((np\\s)/pp_de)\\((np*)/pp_de)") PP-DE_OBJ:pp_de))

Rule :
→ (NP-OBJ:* tree NPP)
← (NP-OBJ:* NP:n NPP:n*)

Rule :
→ (VN:* tree (COORD CC VN))
← (VN:* VN:* (COORD:** "CC:(**)/*" VN:*))

Rule :
→ (AP:* tree ADJ)
← (AP:* "AP:*/(n\\n)" ADJ:n\\n)

Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np* ("VN:(np*)/np" "VN:(np*)/np" "PP-MOD:((np*)/np)\\((np*)/np)") NP-OBJ:np))

Rule :
→ (PP-ATS:* tree)
← (NP-OBJ:np PP:*)

Rule :
→ (AP:* ADJ Ssub)
← (AP:* ADJ:*/cs Ssub:cs)

Rule :
→ (SENT:* ADVWH tree)
← (SENT:* ADVWH:*/* SENT:*)

Rule :
→ (SENT:* NP PP)
← (SENT:* NP:np PP:np*)

Rule :
→ (SENT:* VPP PP)
← (SENT:* VPP:*/pp PP:pp)

Rule :
→ (PP:* P AP)
← (PP:* "P:*/(n\\n)" AP:n\\n)

Rule : use-ponct
→ (NP:* tree PONCT NP PP PONCT)
← (NP:* NP:np (:np* (":(np*)/pf"
"PONCT:((np*)/pf)/np" (:np NP:np PP:np\\np))
PONCT:pf))

Rule :
→ (VN:* (CLO-P_OBJ y) tree)
← (VN:* ("CLO-P_OBJ:*/(*/pp)" "y:*/(*/pp)") VN:*/pp)

Rule :
→ (AdP-MOD:* tree Ssub)
← (AdP-MOD:* AdP:*/cs Ssub:cs)

Rule :
→ (COORD:s CC tree)
← (COORD:s CC:s/s SENT:s)

Rule :
→ (SENT:* NP-SUJ VN ADV PP-ATS)
← (SENT:* NP-SUJ:np (:np* (":(np*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np*)/pp)") PP-ATS:pp))

Rule :
→ (NP-MOD:* tree VPpart)
← (NP-MOD:* NP:n VPpart:n*)

Rule :
→ (VN:* (CLO-A_OBJ y) tree)
← (VN:* ("CLO-A_OBJ:*/(*/pp)" "y:*/(*/pp)") VN:*/pp)

Rule :
→ (VPinf-OBJ:* VN AP-ATS)
← (VPinf-OBJ:* "VN:*/(n\\n)" AP-ATS:n\\n)

Rule :
→ (VPinf-OBJ:* VN ADV)
← (VPinf-OBJ:* VN:* ADV:**)

Rule :
→ (VPinf-SUJ:* tree)
← (VPinf-SUJ:* SENT:*)

Rule :
→ (NP-SUJ:* ADV tree)
← (NP-SUJ:* ADV:*/* NP:*)

Rule :
→ (NP-SUJ:* tree NP-MOD)
← (NP-SUJ:* NP:* NP-MOD:**)

Rule :
→ (VPinf-OBJ:* tree VPpart-MOD)
← (VPinf-OBJ:* SENT:* VPpart-MOD:**)

Rule :
→ (AdP-MOD:* tree PP)
← (AdP-MOD:* AdP:*/pp PP:pp)

Rule :
→ (AdP:* ADV tree)
← (AdP:* ADV:*/* AdP:*)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "ADV:((np\\s_p)/np)\\((np\\s_p)/np)") NP-OBJ:np)))
Rule :
→ (SENT:* NP-SUJ VN ADV PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/pp_a" "VN:(np\\s)/pp_a" "ADV:((np\\s)/pp_a)\\((np*)/pp_a)") PP-A_OBJ:pp_a))

Rule :
→ (SENT:* CS tree)
← (SENT:* CS:*/* SENT:*)
Rule :
→ (SENT:s tree (COORD CC (VN tree) VPinf-OBJ))
← (SENT:s SENT:s (COORD:s\\s "CC:(s\\s)/s"
(VN:s "VN:s/(np\\s_i)" VPinf-OBJ:np\\s_i)))
Rule :
→ (PP-DE_OBJ:* P NP (COORD CC PP))
← (PP-DE_OBJ:* (:* P:*/np NP:np)
(COORD:** "CC:(**)/*" PP:*))
Rule :
→ (NP:* tree (COORD CC NP PP))
← (NP:* (:* NP:*
(COORD:** "CC:(**)/np" (:np NP:np PP:np\\np))))
Rule :
→ (AP-ATS:* tree PP)
← (AP-ATS:* AP:* PP:**)
Rule :
→ (AdP:* ADV PP)
← (AdP:* ADV:*/pp PP:pp)
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) PP-DE_OBJ)
← (SENT:* CLS-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/pp_de"
PP-DE_OBJ:pp_de)))
Rule :
→ (VN:* CLO-MOD tree)
← (PP-DE_OBJ:pp_de CLO-MOD:*/* VN:*)
Rule :
→ (Srel:* NP-MOD tree)
← (Srel:* NP-MOD:*/* SENT:*)
Rule :
→ (VPinf:* VN PP-DE_OBJ)
← (VPinf:* VN:*/pp_de PP-DE_OBJ:pp_de)
Rule :
→ (AdP:* tree Ssub)
← (AdP:* AdP:*/cs Ssub:cs)

Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) Ssub-OBJ)
← (SENT:* CLS-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/cs" Ssub-OBJ:cs)))
Rule :
→ (SENT NP-SUJ tree)
← (Ssub-OBJ:cs NP-SUJ:np SENT:np\\s)
Rule :
→ (SENT:* tree VPpart-ATS)
← (SENT:* "SENT:*/(np\\s_p)" VPpart-ATS:np\\s_p)
Rule :
→ (NP:* NPP PONCT)
← (NP:* NPP:n PONCT:n*)
Rule :
→ (NP-MOD:* ADV DET tree)
← (NP-MOD:* ADV:*/* (:* DET:*/n NP:n))
Rule :
→ (Srel:* (PP-DE_OBJ PROREL) tree)
← (Srel:* (PP-DE_OBJ:*/s PROREL:*/s) SENT:s)
Rule :
→ (NP:* tree DET)
← (NP:* "NP:*/(np/n)" DET:np/n)
Rule :
→ (NP-MOD:* tree (COORD CC NP))
← (NP-MOD:* (:* NP:* (COORD:** "CC:(**)/np" NP:np)))
Rule :
→ (VN:* tree CLS)
← (NP:np VN:*/np CLS:np)
Rule :
→ (VN:* tree PP)
← (VN:* "VN:*/(n\\n)" PP:n\\n)
Rule :
→ (VPinf-OBJ:* tree Ssub-MOD)
← (VPinf-OBJ:* SENT:* Ssub-MOD:**)
Rule :
→ (Ssub:s tree)
← (Ssub:s SENT:s)

Rule :
→ (SENT:* NP-SUJ VN ADV VPinf-ATS)
← (SENT:* NP-SUJ:np (:np*
(":(np*)/(np\\s_i)" "VN:(np*)/(np\\s_i)" "ADV:((np*)/(np\\s_i))\\((np*)/(np\\s_i))") VPinf-ATS:np\\s_i))

Rule :
→ (SENT:* AP tree)
← (SENT:* AP:*/* SENT:*)

Rule :
→ (SENT:* COORD NP-SUJ tree)
← (SENT:* COORD:*/* (:* NP-SUJ:np SENT:np*))

Rule :
→ (SENT:* tree (COORD CC tree Ssub-MOD))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* SENT:* Ssub-MOD:**)))

Rule :
→ (NP-MOD:* tree NPP)
← (Ssub-MOD:** NP:n NPP:n*)
Rule :
→ (VPinf:* tree Ssub-MOD)
← (VPinf:* VPinf:* Ssub-MOD:**)
Rule : use-ponct
→ (SENT PONCT NP PONCT)
← (SENT:s (:s/gf "PONCT:(s/gf)/np" NP:np) PONCT:gf)
Rule :
→ (SENT NP)
← (SENT:np NP:np)

Rule :
→ (SENT:* NP-SUJ VN ADV Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/cs" "VN:(np\\s)/cs" "ADV:((np\\s)/cs)\\((np*)/cs)") Ssub-OBJ:cs))
Rule :
→ (SENT:* NP-SUJ VPpart-MOD VN)
← (SENT:* NP-SUJ:np (:np* "VPpart-MOD:(np*)/(np*)" VN:np*))

```
Rule :
→ (SENT:* tree Srel)
← (SENT:* SENT:*/s Srel:s)
Rule :
→ (PP:* NP P NP)
← (PP:* "NP:*/(n\\n)" (:n\\n "P:(n\\n)/np" NP:np))
Rule :
→ (PP:* P NP (COORD CC tree))
← (PP:* P:*/np
     (:np NP:np (COORD:np\\np "CC:(np\\np)/np" NP:np)))
Rule :
→ (NP:* NC VPpart)
← (NP:* NC:n VPpart:n\\*)
Rule :
→ (NP-SUJ:* ADJ tree)
← (NP-SUJ:* ADJ:*/n NP:n)
Rule :
→ (NP-OBJ:* P tree)
← (NP-OBJ:* P:*/n NP:n)
Rule :
→ (NP-OBJ:* tree VPinf)
← (NP-OBJ:* NP:n VPinf:n\\*)
Rule :
→ (VPinf:* P VN ADV)
← (VPinf:* "P:*/(np\\s_i)"
     (:np\\s_i VN:np\\s_i "ADV:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) PP-A_OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/pp_a" PP-A_OBJ:pp_a)))
Rule :
→ (SENT:* tree (COORD CC tree VPinf-OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
       (:* "SENT:*/(np\\s_i)" VPinf-OBJ:np\\s_i)))
Rule :
→ (NP:* DETWH tree)
← (VPinf-OBJ:np\\s_i DETWH:*/n NP:n)
Rule :
→ (VN:* tree (VPP été) ADV VPP)
← (VN:* "VN:*/(np\\s_p)" (:np\\s_p
       ("VPP:(np\\s_p)/(np\\s_p)"
         "été:(np\\s_p)/(np\\s_p)")
       (:np\\s_p "ADV:(np\\s_p)/(np\\s_p)" VPP:np\\s_p)))
Rule :
→ (Srel:* (NP-ATS PROREL) tree)
← (Srel:* (NP-ATS:*/s PROREL:*/s) SENT:s)
Rule :
→ (Srel:* NP-DE_OBJ tree)
← (Srel:* NP-DE_OBJ:np SENT:np\\*)
Rule :
→ (VPinf-MOD:* P VN PP-DE_OBJ)
← (VPinf-MOD:* "P:*/(np\\s_i)"
     (:np\\s_i "VN:(np\\s_i)/pp_de" PP-DE_OBJ:pp_de))
Rule :
→ (VPinf:* P VN PP-DE_OBJ)
← (VPinf:* "P:*/(np\\s_i)"
     (:np\\s_i "VN:(np\\s_i)/pp_de" PP-DE_OBJ:pp_de))
Rule :
→ (Ssub-DE_OBJ:* CS tree)
← (Ssub-DE_OBJ:* CS:*/s SENT:s)
Rule :
→ (SENT:* VPinf-SUJ tree)
← (SENT:* VPinf-SUJ:np\\s_i "SENT:(np\\s_i)\\*")
Rule :
→ (SENT:* tree (COORD CC tree VPinf-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
       (:* SENT:* VPinf-MOD:*\\*)))
Rule :
→ (PP-ATO:* P NP)
← (PP-ATO:* P:*/np NP:np)
Rule :
→ (PP:* P NP ADV)
← (PP:* (:n\\n "P:(n\\n)/np" NP:np) "ADV:(n\\n)\\*")
Rule :
→ (PP:* P NP (COORD CC tree PP))
← (PP:* (:* P:*/np NP:np)
     (COORD:*\\* "CC:(*\\*)/*" (:* AdP:*/* PP:*)))
Rule :
→ (NP-SUJ:* ADV DET tree)
← (NP-SUJ:* ADV:*/* (:* DET:*/n NP:n))
Rule :
→ (VPinf-ATS:* P VN)
← (VPinf-ATS:* "P:*/(np\\s_i)" VN:np\\s_i)
Rule :
→ (VPinf-MOD:* tree (COORD CC VPinf))
← (VPinf-MOD:* VPinf:*
     (COORD:*\\* "CC:(*\\*)/(np\\s_i)" VPinf:np\\s_i))
Rule :
→ (AP:* tree VPinf)
← (AP:* AP:* VPinf:*\\*)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) VPinf-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
         VPinf-DE_OBJ:np\\s_i)))
Rule :
→ (SENT:* tree COORD-MOD)
← (VPinf-DE_OBJ:np\\s_i SENT:* COORD-MOD:*\\*)
Rule :
→ (SENT:* NP-SUJ VN ADV ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
        "ADV:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN NP-OBJ VPinf-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/np"
       (:np NP-OBJ:np VPinf-A_OBJ:np\\np)))
Rule :
→ (SENT:* tree (COORD CC tree PP-DE_OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
       (:* SENT:*/pp_de PP-DE_OBJ:pp_de)))
Rule :
→ (NP:* tree AP (COORD CC PP))
← (NP:* NP:n (:n\\* AP:n\\n ("COORD:(n\\n)\\(n\\*)" "CC:((n\\n)\\(n\\*))/(n\\n)" PP:n\\n)))
```

Rule :
→ (NP:* NPP ET)
← (PP:n\\n NPP:n ET:n*)

Rule :
→ (Srel-MOD:* (NP-OBJ PROREL) tree)
← (Srel-MOD:* (NP-OBJ:*/s PROREL:*/s) SENT:s)

Rule :
→ (Srel:* (PP-DE_OBJ (NP PROREL)) tree)
← (Srel:* (PP-DE_OBJ:*/s (NP:*/s PROREL:*/s)) SENT:s)

Rule : use-ponct
→ (SENT NP PP PONCT)
← (SENT:txt (:np NP:np PP:np\\np) PONCT:np\\txt)

Rule :
→ (SENT:* tree Ssub-DE_OBJ)
← (SENT:* SENT:*/cs Ssub-DE_OBJ:cs)

Rule :
→ (NP-OBJ:* ADJ tree)
← (NP-OBJ:* ADJ:*/n NP:n)

Rule :
→ (NP-MOD:* ADV tree)
← (NP-MOD:* ADV:*/n NP:n)

Rule :
→ (SENT VPinf)
← (SENT:np\\s_i VPinf:np\\s_i)

Rule :
→ (PP:* P ADV)
← (PP:* "P:*/(n\\n)" ADV:n\\n)

Rule :
→ (NP:* DET (COORD CC DET) tree)
← (NP:* (:*/n DET:*/n ("COORD:(*/n)\\(*/n)"
"CC:((*/n)\\(*/n))/(*/n)" DET:*/n))
NP:n)

Rule :
→ (NP:* tree Sint)
← (NP:* NP:np Sint:np*)

Rule :
→ (Srel:* (PP PROREL) NP)
← (Srel:* (PP:*/np PROREL:*/np) NP:np)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "NP-MOD:((np\\s_p)/np)\\((np\\s_p)/np)") NP-OBJ:np)))

Rule :
→ (VN:* CLO-ATS tree)
← (NP-OBJ:np "CLO-ATS:*/(*/np)" VN:*/np)

Rule :
→ (VPinf:* tree VPpart-MOD)
← (VPinf:* VPinf:* VPpart-MOD:**)

Rule :
→ (VPinf:* tree ADV)
← (VPinf:* VPinf:* ADV:**)

Rule :
→ (AP-ATO:* ADV ADJ)
← (AP-ATO:* ADV:*/* ADJ:*)

Rule :
→ (AP-ATS:* AdP ADJ)
← (AP-ATS:* "AdP:*/(n/n)" ADJ:n/n)

Rule :
→ (Ssub-MOD:* ADV tree)
← (Ssub-MOD:* ADV:*/s SENT:s)

Rule :
→ (SENT NP PONCT NP)
← (SENT:np NP:np (:np\\np "PONCT:(np\\np)/np" NP:np))

Rule :
→ (SENT:* VN AP-ATS Ssub-OBJ)
← (SENT:* (:*/cs "VN:(*/cs)/(n\\n)" AP-ATS:n\\n)
Ssub-OBJ:cs)

Rule :
→ (SENT:* CS NP-SUJ tree)
← (SENT:* CS:*/* (:* NP-SUJ:np SENT:np*))

Rule :
→ (SENT:* tree (COORD CC VN))
← (SENT:* SENT:* (COORD:** "CC:(**)/*" VN:*))

Rule :
→ (PP-A_OBJ:* P NP (COORD CC PP))
← (PP-A_OBJ:* (:* P:*/np NP:np)
(COORD:** "CC:(**)/*" PP:*))

Rule :
→ (PP-MOD:* P+D NP (COORD CC PP))
← (PP-MOD:* (:* P+D:*/n NP:n)
(COORD:** "CC:(**)/*" PP:*))

Rule :
→ (PP:* P NP NP)
← (PP:* P:*/np (:np NP:np NP:np\\np))

Rule :
→ (VN:* (CLO-A_OBJ me) tree)
← (VN:* ("CLO-A_OBJ:*/(*/pp)" "me:*/(*/pp)") VN:*/pp)

Rule :
→ (Srel:* tree VPinf-MOD)
← (Srel:* SENT:* VPinf-MOD:**)

Rule :
→ (VPpart-MOD:* VN PP-DE_OBJ)
← (VPpart-MOD:* VN:*/pp_de PP-DE_OBJ:pp_de)

Rule :
→ (SENT:* NP-SUJ VN Ssub-ATS)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/cs" Ssub-ATS:cs))

Rule :
→ (SENT:* NP-SUJ VN NP-OBJ PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np*
(":(np*)/pp" "VN:((np*)/pp)/np" NP-OBJ:np)
PP-P_OBJ:pp))

Rule :
→ (SENT:* tree VPinf-P_OBJ)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-P_OBJ:np\\s_i)

Rule :
→ (SENT:* tree (COORD CC ADV VPinf))
← (SENT:* SENT:* (COORD:** "CC:(**)/(np\\s_i)"
(:np\\s_i "ADV:(np\\s_i)/(np\\s_i)"
VPinf:np\\s_i)))

Rule :
→ (PP-P_OBJ:* P+D (NP NC))
← (PP-P_OBJ:* P+D:*/n (NP:n NC:n))

Rule :
→ (PP-MOD:* P PRO)
← (PP-MOD:* P:*/np PRO:np)

Rule : use-ponct
→ (NP:* tree (Sint-MOD PONCT Sint PONCT))
← (NP:* NP:n (Sint-MOD:n*
(":(n*)/pf" "PONCT:((n*)/pf)/s" Sint:s)
PONCT:pf))

Rule :
→ (Srel:* tree VPpart-MOD)
← (Srel:* SENT:* VPpart-MOD:**)

Rule :
→ (VPpart-MOD:* tree NP-MOD)
← (VPpart-MOD:* VPpart:* NP-MOD:**)

Rule :
→ (VPpart:* VPP (COORD CC VPpart))
← (VPpart:* VPP:np\\s ("COORD:(np\\s)*"
"CC:((np\\s)*)/(np\\s)" VPpart:np\\s))

```
Rule :
→ (VPinf-DE_OBJ:* P VN PP-MOD)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "PP-MOD:(np\\s_i)\\(np\\s_i)"))
```

```
Rule :
→ (VPinf:* ADVWH tree)
← (VPinf:* ADVWH:*/* VPinf:*)
Rule :
→ (VPinf:* tree PP-DE_OBJ)
← (VPinf:* VPinf:*/pp_de PP-DE_OBJ:pp_de)
Rule :
→ (VPinf-P_OBJ:* tree)
← (VPinf-P_OBJ:* SENT:*)
Rule :
→ (AP-MOD:* ADJ PP)
← (AP-MOD:* ADJ:*/pp PP:pp)
Rule :
→ (AP:* ADJ (COORD CC AP) (COORD CC AP))
← (AP:* ADJ:* (COORD:*\\* "CC:(*\\*)/*"
      (:* AP:* (COORD:*\\* "CC:(*\\*)/*" AP:*))))
Rule :
→ (AP-ATS:* tree VPinf)
← (AP-ATS:* AP:* VPinf:*\\*)
Rule :
→ (Ssub-OBJ:* ADVWH tree)
← (Ssub-OBJ:* ADVWH:*/s SENT:s)
Rule :
→ (SENT:* (VN CLS-SUJ tree) VPinf-ATS)
← (SENT:* CLS-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_i)" VPinf-ATS:np\\s_i))
Rule :
→ (SENT:* NP-SUJ NP-MOD VN)
← (SENT:* NP-SUJ:np
    (:np\\* "NP-MOD:(np\\*)/(np\\*)" VN:np\\*))
Rule :
→ (SENT:* PP NP)
← (SENT:* PP:*/np NP:np)
Rule :
→ (PP-ATS:* P+D NP)
← (PP-ATS:* P+D:*/n NP:n)
Rule :
→ (NP:* tree (COORD CC AP))
← (NP:* NP:* (COORD:*\\* "CC:(*\\*)/(n\\n)" AP:n\\n))
Rule :
→ (NP-OBJ:* tree ADJ)
← (NP-OBJ:* NP:n ADJ:n\\*)
Rule :
→ (NP-MOD:* tree NP)
← (NP-MOD:* NP:n NP:n\\*)
Rule :
→ (VN:* tree NP)
← (VN:* VN:*/np NP:np)
Rule : use-ponct
→ (VPpart-MOD:* PONCT VPpart PONCT)
← (VPpart-MOD:*
    (:*/pf "PONCT:(*/pf)/(np\\s)" VPpart:np\\s) PONCT:pf)
Rule :
→ (SENT:* AP-ATS VN NP-SUJ)
← (SENT:* (:*/np AP-ATS:n\\n "VN:(n\\n)\\(*/np)")
    NP-SUJ:np)
Rule :
→ (SENT:* tree AdP-ATS)
← (SENT:* SENT:* AdP-ATS:*\\*)
Rule :
→ (SENT:* tree (COORD CC ADV))
← (SENT:* SENT:*
    (COORD:*\\* "CC:(*\\*)/(*\\*)" ADV:*\\*))
```

```
Rule : use-ponct
→ (NP:* tree PONCT Srel PONCT)
← (NP:* NP:np (:np\\*
      (":(np\\*)/pf" "PONCT:((np\\*)/pf)/s" Srel:s)
      PONCT:pf))
Rule :
→ (NP:* tree (COORD CC CC NP))
← (NP:* NP:* (COORD:*\\* (":(*\\*)/*"
       "CC:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      NP:*))
Rule :
→ (VPinf:* VN AP-ATS)
← (VPinf:* "VN:*/(n\\n)" AP-ATS:n\\n)
Rule :
→ (VPinf-MOD:* tree PP-MOD)
← (VPinf-MOD:* VPinf:* PP-MOD:*\\*)
Rule :
→ (SENT:* NP-SUJ VN VPpart-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_p)" VPpart-ATS:np\\s_p))
Rule :
→ (SENT:* NP-SUJ VN NP-OBJ PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp_de"
       "VN:((np\\*)/pp_de)/np" NP-OBJ:np)
      PP-DE_OBJ:pp_de))
Rule :
→ (SENT:* NP-SUJ VN PP-A_OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np"
       "VN:((np\\*)/np)/pp_a" PP-A_OBJ:pp_a)
      NP-OBJ:np))
Rule :
→ (SENT:* NP VN)
← (SENT:* NP:np VN:np\\*)
Rule :
→ (PP-A_OBJ:* P+D NP (COORD CC PP))
← (PP-A_OBJ:* (:* P+D:*/n NP:n)
    (COORD:*\\* "CC:(*\\*)/*" PP:*))
Rule :
→ (PP:* P P NP)
← (PP:* P:*/pp (:pp P:pp/np NP:np))
Rule :
→ (PP:* P PRO)
← (PP:* P:*/np PRO:np)
Rule :
→ (NP-MOD:* ADJ DET tree)
← (NP-MOD:* ADJ:*/* (:* DET:*/n NP:n))
Rule :
→ (NP-MOD:* PRO ADV)
← (NP-MOD:* PRO:np ADV:np\\*)
Rule :
→ (VN:* CLR-DE_OBJ tree)
← (VN:* CLR-DE_OBJ:cl_r VN:cl_r\\*)
Rule :
→ (VPpart:* NP-SUJ VN)
← (VPpart:* NP-SUJ:np VN:np\\*)
Rule :
→ (AP-ATS:* tree Ssub)
← (AP-ATS:* AP:*/cs Ssub:cs)
Rule :
→ (Ssub-MOD:* ADV CS tree)
← (Ssub-MOD:* ADV:*/* (:* CS:*/s SENT:s))
Rule :
→ (AdP:* NP tree)
← (AdP:* NP:np AdP:np\\*)
Rule :
→ (COORD:* CC NP)
← (COORD:* CC:*/np NP:np)
```

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "PP-MOD:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-P_OBJ:pp)))

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-ATS)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-ATS:pp)))

Rule :
→ (PP-OBJ:* P (NP NC))
← (PP-OBJ:* P:*/n (NP:n NC:n))

Rule :
→ (PP:* P ADJ)
← (PP:* P:*/n ADJ:n)

Rule :
→ (PP-OBJ:* tree)
← (PP-OBJ:* PP:*)

Rule : use-ponct
→ (NP:* NC AP PP PONCT)
← (NP:* (:n (:n NC:n AP:n\\n) PP:n\\n) PONCT:n*)

Rule :
→ (NP-ATS:* tree VPpart)
← (NP-ATS:* NP:n VPpart:n*)

Rule :
→ (AP-ATS:* ADJ (COORD CC tree))
← (AP-ATS:* ADJ:* (COORD:** "CC:(**)/*" AP-ATS:*))

Rule :
→ (AdP-MOD:* ADV ADV Ssub)
← (AdP-MOD:* ADV:*/* (:* ADV:* Ssub:**))

Rule :
→ (AdP-MOD:* ADV (COORD CC ADV))
← (AdP-MOD:* ADV:* (COORD:** "CC:(**)/*" ADV:*))

Rule :
→ (COORD:* CC PP)
← (COORD:* CC:*/pp PP:pp)

Rule :
→ (SENT:* Srel-MOD tree)
← (SENT:* Srel-MOD:*/* SENT:*)

Rule :
→ (SENT:* NP-SUJ VN PP-OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/pp" PP-OBJ:pp))

Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-P_OBJ)
← (SENT:* CLS-SUJ:np (:np* (":(np*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np*)/pp)") PP-P_OBJ:pp))

Rule :
→ (SENT:* tree Sint-OBJ)
← (SENT:* SENT:*/s Sint-OBJ:s)

Rule :
→ (SENT:* VPP PP VPinf)
← (SENT:* (":*/(np\\s_i)" "VPP:(*/(np\\s_i))/pp" PP:pp)
VPinf:np\\s_i)

Rule :
→ (SENT:* tree (COORD CC CC Sint))
← (SENT:* SENT:* (COORD:** (":(**)/*"
"CC:((**)/*)/((**)/*)" "CC:(**)/*")
Sint:*))

Rule :
→ (SENT:* tree (COORD CC tree NP-MOD))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* SENT:* NP-MOD:**)))

Rule :
→ (PP:* P NC)
← (NP-MOD:** P:*/n NC:n)

Rule :
→ (PP:* tree PP-MOD)
← (PP:* PP:* PP-MOD:**)

Rule :
→ (NP:* NC (COORD CC AP))
← (NP:* NC:n (COORD:n* "CC:(n*)/(n\\n)" AP:n\\n))

Rule :
→ (NP:* NC P)
← (NP:* NC:n P:n*)

Rule : use-ponct
→ (NP:* tree (PONCT -LBR-) VPpart (PONCT -RBR-))
← (NP:* NP:np (:np* (":(np*)/pf" ("PONCT:((np*)/pf)/(n\\n)" "-LBR-:((np*)/pf)/(n\\n)") VPpart:n\\n)
(PONCT:pf -RBR-:pf)))

Rule :
→ (VPinf-OBJ:* VN PP-MOD PP-MOD)
← (VPinf-OBJ:* (:* VN:* PP-MOD:**) PP-MOD:**)

Rule :
→ (VPinf-OBJ:* P VN PP-DE_OBJ)
← (VPinf-OBJ:* "P:*/(np\\s_i)"
(:np\\s_i "VN:(np\\s_i)/pp_de" PP-DE_OBJ:pp_de))

Rule :
→ (VPinf-OBJ:* tree AdP-MOD)
← (VPinf-OBJ:* SENT:* AdP-MOD:**)

Rule : use-ponct
→ (AP-ATS:* tree PONCT)
← (AP-ATS:* AP:* PONCT:**)

Rule :
→ (Sint-OBJ:* tree)
← (Sint-OBJ:* SENT:*)

Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "ADV:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-P_OBJ:pp)))

Rule :
→ (SENT:* NP-SUJ VN ADV ADV VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/(np\\s_i)"
(":(np*)/(np\\s_i)" "VN:(np*)/(np\\s_i)" "ADV:((np*)/(np\\s_i))\\((np*)/(np\\s_i))")
"ADV:((np*)/(np\\s_i))\\((np*)/(np\\s_i))")
VPinf-OBJ:np\\s_i))

Rule :
→ (SENT:* NP-SUJ VN PP-A_OBJ VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/(np\\s_i)" "VN:((np*)/(np\\s_i))/pp_a" PP-A_OBJ:pp_a) VPinf-OBJ:np\\s_i))

Rule :
→ (SENT:* tree (COORD-OBJ CC NP))
← (SENT:* SENT:*/np (COORD-OBJ:np CC:np/np NP:np))
Rule :
→ (SENT:* tree (COORD CC (VN tree) VPinf-OBJ))
← (SENT:* SENT:* (COORD:** "CC:(**)/(np\\s_i)"
(VN:np\\s_i "VN:(np\\s_i)/(np\\s_i)"
VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* NP tree PP)
← (SENT:* (:np NP:np NP:np\\np) PP:np*)

Rule :
→ (SENT:* tree (COORD CC tree PP-P_OBJ))
← (SENT:* SENT:* (COORD:** "CC:(**)/*"
(:* SENT:*/pp PP-P_OBJ:pp)))
Rule :
→ (NP:* AdP P)
← (PP-P_OBJ:pp AdP:*/* P:*)
Rule :
→ (VN:* tree NP-MOD)
← (VN:* VN:* NP-MOD:**)

Rule : use-ponct
→ (VN:* tree PONCT VPP PONCT)
← (VN:* "VN:*/(np\\s_p)" (:np\\s_p (":(np\\s_p)/gf" "PONCT:((np\\s_p)/gf)/(np\\s_p)" VPP:np\\s_p) PONCT:gf))

Rule :
→ (VPpart-MOD:* tree VPinf-MOD)
← (VPpart-MOD:* VPpart:* VPinf-MOD:**)
Rule :
→ (VPpart:* NP tree)
← (VPpart:* NP:np VPpart:np*)
Rule : use-ponct
→ (VPinf-MOD:* tree PONCT)
← (VPinf-MOD:* VPinf:* PONCT:**)
Rule :
→ (VPinf:* tree VPinf-A_OBJ)
← (VPinf:* "VPinf:*/(np\\s_i)" VPinf-A_OBJ:np\\s_i)
Rule :
→ (VN:* VINF ADV)
← (VN:* VINF:* ADV:**)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
"VPP:(np\\s_p)/pp_a" PP-A_OBJ:pp_a))))

Rule :
→ (SENT:* (VN CLS-SUJ tree) Ssub-ATS)
← (SENT:* CLS-SUJ:np
(:np* "VN:(np*)/cs" Ssub-ATS:cs))
Rule :
→ (SENT:* tree NP-OBJ (COORD CC NP))
← (SENT:* SENT:*/np (:np NP-OBJ:np
(COORD:np\\np "CC:(np\\np)/np" NP:np)))
Rule :
→ (SENT:* tree (COORD CC VN Ssub-OBJ))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* VN:*/cs Ssub-OBJ:cs)))
Rule :
→ (SENT:* tree (COORD CC tree AP-ATS))
← (SENT:* SENT:* (COORD:** "CC:(**)/*"
(:* "SENT:*/(n\\n)" AP-ATS:n\\n)))
Rule :
→ (PP-A_OBJ:* P (NP NC))
← (PP-A_OBJ:* P:*/n (NP:n NC:n))
Rule :
→ (NP-ATS:* tree AP)
← (NP-ATS:* NP:n AP:n*)

Rule :
→ (VN:* tree VPP (COORD CC VPP))
← (VN:* "VN:*/(np\\s_p)"
(:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)" VPP:np\\s_p)))

Rule :
→ (Srel:* NP-SUJ VN NP-OBJ)
← (Srel:* NP-SUJ:np (:np* "VN:(np*)/np" NP-OBJ:np))

Rule :
→ (VPpart-ATS:* VPP PP)
← (VPpart-ATS:* VPP:*/pp PP:pp)

Rule :
→ (VPpart-MOD:* NP tree)
← (VPpart-MOD:* NP:np VPpart:np*)

Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) VPinf-A_OBJ)
← (SENT:* CLS-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)"
(VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
VPinf-A_OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
"VPP:(np\\s_p)/pp_de" PP-DE_OBJ:pp_de))))

Rule :
→ (SENT:* NP-SUJ VN ADV VPinf-A_OBJ)
← (SENT:* NP-SUJ:np (:np*
(":(np*)/(np\\s_i)" "VN:(np*)/(np\\s_i)" "ADV:((np*)/(np\\s_i))\\((np*)/(np\\s_i))")
VPinf-A_OBJ:np\\s_i))

```
Rule :
→ (SENT:* PP NP PP)
← (SENT:* PP:*/np (:np NP:np PP:np\\np))
Rule :
→ (SENT:* VN PP-ATO)
← (SENT:* VN:*/pp PP-ATO:pp)
Rule :
→ (SENT:* tree (COORD CC tree Ssub-OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* SENT:*/cs Ssub-OBJ:cs)))
Rule :
→ (SENT:* tree (COORD CC tree VPinf-A_OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* "SENT:*/(np\\s_i)" VPinf-A_OBJ:np\\s_i)))
Rule :
→ (PP:* tree Srel)
← (PP:* PP:*/s Srel:s)
Rule :
→ (PP:* tree (COORD CC (PP P NP) PP))
← (PP:* PP:* (COORD:*\\* "CC:(*\\*)/*"
      (PP:* P:*/np (:np NP:np PP:np\\np))))
Rule :
→ (NP:* tree PRO)
← (NP:* NP:n PRO:n\\*)
Rule :
→ (NP:* NC AP (COORD CC tree))
← (NP:* (:n NC:n AP:n\\n)
    (COORD:n\\* "CC:(n\\*)/np" NP:np))
```

```
Rule :
→ (VN:* tree VPinf)
← (VN:* "VN:*/(np\\s_i)" VPinf:np\\s_i)
Rule :
→ (VN:* V ADJ)
← (VN:* "V:*/(n\\n)" ADJ:n\\n)
Rule :
→ (Srel:* tree NP-MOD)
← (Srel:* SENT:* NP-MOD:*\\*)
Rule :
→ (VPinf-OBJ:* ADV tree)
← (VPinf-OBJ:* ADV:*/* SENT:*)
Rule :
→ (AP:* ADV ADV ADJ)
← (AP:* ADV:*/* (:* "ADV:*/(n/n)" ADJ:n/n))
Rule : use-ponct
→ (AP-MOD:* tree PONCT)
← (AP-MOD:* AP:* PONCT:*\\*)
Rule :
→ (AdP-ATS:* ADV tree)
← (AdP-ATS:* ADV:*/* AdP:*)
Rule : use-ponct
→ (SENT NP Ssub PONCT)
← (TEXT:txt NP:np
    (SENT:np\\txt Ssub:cs "PONCT:cs\\(np\\txt)"))
```

```
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "AdP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "PP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-A_OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "ADV:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
Rule :
→ (SENT:* VN ADV VPinf-DE_OBJ)
← (SENT:* (":*/(np\\s_i)" "VN:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VPinf-DE_OBJ:np\\s_i)
```

```
Rule :
→ (SENT:* VN NP-OBJ AP-ATO)
← (SENT:* VN:*/np (:np NP-OBJ:np AP-ATO:np\\np))
Rule :
→ (SENT:* NP NP-SUJ tree)
← (SENT:* (:np NP:np/np NP-SUJ:np) SENT:np\\*)
Rule :
→ (SENT:* I tree)
← (SENT:* I:*/* SENT:*)
Rule :
→ (SENT:* tree PP-ATO)
← (SENT:* SENT:*/pp PP-ATO:pp)
Rule :
→ (SENT:* VPP PP PP)
← (SENT:* (:*/pp "VPP:(*/pp)/pp" PP:pp) PP:pp)
Rule :
→ (SENT:* (COORD CC NP) tree)
← (SENT:* (COORD:*/* "CC:(*/*)/np" NP:np) SENT:*)
Rule :
→ (SENT:* tree (COORD CC tree PP-A_OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* SENT:*/pp_a PP-A_OBJ:pp_a)))
Rule :
→ (NP:* tree (COORD CC CC tree))
← (NP:* NP:* (COORD:*\\* (":(*\\*)/*"
        "CC:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      NP:*))
```

```
Rule :
→ (NP-ATS:* tree (COORD CC NP))
← (NP-ATS:* (:* NP:* (COORD:*\\* "CC:(*\\*)/np" NP:np)))
Rule :
→ (VPinf-ATS:* VN NP-OBJ)
← (NP:np VN:*/np NP-OBJ:np)
Rule :
→ (VPinf-MOD:* ADV tree)
← (VPinf-MOD:* ADV:*/* VPinf:*)
Rule :
→ (AP:* ADJ NP)
← (AP:* ADJ:*/np NP:np)
Rule :
→ (AP:* PONCT tree)
← (AP:* PONCT:*/* AP:*)
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) VPinf-DE_OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_i)"
        VPinf-DE_OBJ:np\\s_i)))
Rule :
→ (SENT tree Sint-MOD)
← (VPinf-DE_OBJ:np\\s_i SENT:s Sint-MOD:s\\s)
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD PP-P_OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "PP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
       (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "PP-MOD:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))")
       VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)"
         "ADV:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np\\*)/pp)") PP-ATS:pp))
Rule :
→ (SENT:* VN ADV VPinf-A_OBJ)
← (SENT:* (":*/(np\\s_i)" "VN:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VPinf-A_OBJ:np\\s_i)
```

```
Rule :
→ (SENT:* PP-DE_OBJ tree)
← (SENT:* PP-DE_OBJ:pp_de SENT:pp_de\\*)
Rule :
→ (SENT:* tree Sint)
← (SENT:* SENT:* Sint:*\\*)
Rule :
→ (SENT:* tree NP-OBJ (COORD CC VPpart))
← (SENT:* (:* SENT:*/np (:np NP-OBJ:np (COORD:np\\np
          "CC:(np\\np)/(n\\n)" VPpart:n\\n))))
Rule :
→ (SENT:* NP VPinf)
← (SENT:* NP:np VPinf:np\\*)
Rule :
→ (SENT:* VINF PP)
← (SENT:* VINF:*/pp PP:pp)
```

```
Rule :
→ (SENT:* tree (COORD CC VN PP-P_OBJ))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* VN:*/pp PP-P_OBJ:pp)))
Rule :
→ (PP:* tree Ssub)
← (PP-P_OBJ:pp PP:*/cs Ssub:cs)
Rule :
→ (PP:* tree (COORD CC (PP P NP)) PP)
← (PP:* PP:* (COORD:*\\* "CC:(*\\*)/*"
       (PP:* P:*/np (:np NP:np PP:np\\np))))
Rule :
→ (PP:* tree ADV)
← (PP:np\\np PP:* ADV:*\\*)
Rule :
→ (PP:* P+D NP (COORD CC tree))
← (PP:* P+D:*/n
     (:n NP:n (COORD:n\\n "CC:(n\\n)/np" NP:np)))
```

```
Rule :
→ (NP:* tree AP (COORD CC AP))
← (NP:* NP:* (:*\\* AP:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" AP:*\\*)))
Rule :
→ (NP:* tree AP (COORD CC VPpart))
← (NP:* NP:* (:*\\* AP:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" VPpart:*\\*)))
```

```
Rule :
→ (NP:* tree P)
← (VPpart:*\\* NP:n P:n\\*)
Rule :
→ (NP:* tree (COORD CC VPpart))
← (NP:* NP:* (COORD:*\\* "CC:(*\\*)/(n\\n)" VPpart:n\\n))
Rule :
→ (NP-SUJ:* P tree)
← (NP-SUJ:* P:*/n NP:n)
Rule :
→ (NP-MOD:* tree ADV)
← (NP-MOD:* NP:n ADV:n\\*)
```

```
Rule :
→ (VPinf:* tree VPinf-DE_OBJ)
← (VPinf:* "VPinf:*/(np\\s_i)" VPinf-DE_OBJ:np\\s_i)
Rule :
→ (VPinf-MOD:* tree)
← (VPinf-MOD:* SENT:*)
Rule :
→ (COORD-MOD:* PONCT COORD PONCT)
← (COORD-MOD:* (:*/pf "PONCT:(*/pf)/*" COORD:*) PONCT:pf)
Rule :
→ (SENT PP)
← (SENT:pp PP:pp)
```

```
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p (":(np\\s_p)/pp_a" "VPP:(np\\s_p)/pp_a" "ADV:((np\\s_p)/pp_a)\\((np\\s_p)/pp_a)") PP-A_OBJ:pp_a)))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "PP-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
```

Rule :
→ (SENT:* NP-SUJ VN ADV PP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np" (":(np*)/np" "VN:(np*)/np" "ADV:((np*)/np)\\((np*)/np)")
"PP-MOD:((np*)/np)\\((np*)/np)")
NP-OBJ:np))

Rule :
→ (SENT:* CC tree)
← (SENT:* CC:*/* SENT:*)
Rule :
→ (PP:* AdP tree)
← (PP:* AdP:*/* PP:*)
Rule :
→ (NP:* tree AP-MOD)
← (NP:* NP:* AP-MOD:**)
Rule :
→ (NP-ATS:* tree NP)
← (NP-ATS:* NP:n NP:n*)
Rule :
→ (VN:* tree PRO)
← (VN:* VN:*/np PRO:np)
Rule :
→ (Srel-MOD:* CS tree)
← (Srel-MOD:* CS:*/* SENT:*)
Rule :
→ (Srel:* tree Ssub-MOD)
← (Srel:* SENT:* Ssub-MOD:**)
Rule :
→ (Srel:* NP tree)
← (Srel:* NP:*/* SENT:*)

Rule :
→ (VPpart:* NP-SUJ VN NP-OBJ)
← (VPpart:* (:*/np NP-SUJ:np "VN:np\\(*/np)") NP-OBJ:np)
Rule :
→ (VPinf-MOD:* P VN ADV)
← (VPinf-MOD:* "P:*/(np\\s_i)"
(VN:np\\s_i VN:np\\s_i "ADV:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf:* tree PP-OBJ)
← (VPinf:* VPinf:*/pp PP-OBJ:pp)
Rule :
→ (Ssub-SUJ:* CS tree)
← (Ssub-SUJ:* CS:*/s SENT:s)
Rule :
→ (AdP-MOD:* ADV tree)
← (AdP-MOD:* ADV:*/* AdP:*)
Rule :
→ (SENT (NP NPP))
← (SENT:np (NP:np NPP:np))
Rule :
→ (SENT:* NP-OBJ VN)
← (SENT:* NP-OBJ:np VN:np*)

Rule :
→ (SENT:* NP-SUJ PP-MOD VN AP-ATS)
← (SENT:* NP-SUJ:np
(:np* (":(np*)/(n\\n)" "PP-MOD:((np*)/(n\\n))/((np*)/(n\\n))" "VN:(np*)/(n\\n)") AP-ATS:n\\n))
Rule :
→ (SENT:* NP-SUJ PP-MOD VN PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/pp" "PP-MOD:((np*)/pp)/((np*)/pp)" "VN:(np*)/pp") PP-P_OBJ:pp))

Rule :
→ (SENT:* tree AdP-OBJ)
← (SENT:* "SENT:*/(s\\s)" AdP-OBJ:s\\s)
Rule :
→ (SENT:* NP NP (PONCT -COL-) tree)
← (SENT:* (:*/* NP:np (":np\\(*/*)" NP:np
("PONCT:np\\(np\\(*/*))"
"-COL-:np\\(np\\(*/*))")))
SENT:*)
Rule :
→ (SENT:* tree NP-ATO)
← (SENT:* SENT:*/np NP-ATO:np)
Rule :
→ (SENT:* tree NP-OBJ PP-MOD (COORD CC NP PP))
← (SENT:* SENT:*/np (:np (:np NP-OBJ:np PP-MOD:np\\np)
(COORD:np\\np "CC:(np\\np)/np"
(:np NP:np PP:np\\np))))
Rule :
→ (SENT:* VPP ADV PP)
← (PP:np\\np (:*/pp VPP:*/pp "ADV:(*/pp)\\(*/pp)") PP:pp)
Rule :
→ (SENT:* tree (VPinf-OBJ-MOD PONCT tree PONCT))
← (SENT:* SENT:* (VPinf-OBJ-MOD:** (":(**)/pf"
"PONCT:((**)/pf)/(np\\s)" VPinf-OBJ:np\\s)
PONCT:pf))
Rule :
→ (PP:* P PP NP)
← (PP:* P:*/np (:np PP:np/np NP:np))

Rule : use-ponct
→ (NP:* tree (COORD-MOD PONCT COORD PONCT))
← (NP:* NP:* (COORD-MOD:** (":(**)/pf"
"PONCT:((**)/pf)/(n\\n)" COORD:n\\n)
PONCT:pf))
Rule : use-ponct
→ (NP:* tree PONCT NP ADV PONCT)
← (NP:* NP:np (:np* (":(np*)/pf"
"PONCT:((np*)/pf)/np" (:np NP:np ADV:np\\np))
PONCT:pf))
Rule : use-ponct
→ (NP:* tree PONCT Sint PONCT)
← (NP:* NP:np (:np*
(":(np*)/pf" "PONCT:((np*)/pf)/s" Sint:s)
PONCT:pf))
Rule :
→ (NP-SUJ:* NC NC)
← (NP-SUJ:* NC:* NC:**)
Rule :
→ (NP:* tree VN)
← (NP:* NP:np VN:np*)
Rule :
→ (VN:* V tree V)
← (VN:* "V:*/(np\\s)"
(:np\\s "AdP:(np\\s)/(np\\s)" V:np\\s))
Rule :
→ (VPpart-MOD:* tree ADV)
← (VPpart-MOD:* VPpart:* ADV:**)

```
Rule :
→ (VPinf-MOD:* P VN VPinf-A_OBJ)
← (VPinf-MOD:* "P:*/(np\\s_i)" (VN:np\\s_i
      "VN:(np\\s_i)/(np\\s_i)" VPinf-A_OBJ:np\\s_i))
Rule :
→ (VPinf-A_OBJ:* P VN PP-DE_OBJ)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)"
    (:np\\s_i "VN:(np\\s_i)/pp_de" PP-DE_OBJ:pp_de))
Rule :
→ (VPinf:* tree AdP-MOD)
← (VPinf:* VPinf:* AdP-MOD:*\\*)
Rule :
→ (AP:* tree NC)
← (AP:* AP:*/n NC:n)
Rule :
→ (AdP-MOD:* tree)
← (AdP-MOD:* AdP:*)
```

```
Rule :
→ (AdP-OBJ:* tree)
← (AdP-OBJ:* AdP:*)
Rule :
→ (SENT tree NP-SUJ tree)
← (SENT:s NP-MOD:s/s (:s NP-SUJ:np SENT:np\\s))
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP VPP) PP-P_OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
        (:np\\s_p "VPP:(np\\s_p)/pp" PP-P_OBJ:pp))))
Rule :
→ (SENT:* (NP-OBJ PROWH) tree)
← (PP-P_OBJ:pp (NP-OBJ:*/s PROWH:*/s) SENT:s)
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* "VN:(np\\*)/pp" PP-OBJ:pp))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV Ssub-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)") Ssub-ATS:cs))
```

```
Rule :
→ (SENT:* NP-SUJ VN PP-P_OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/np" "VN:((np\\*)/np)/pp" PP-P_OBJ:pp)
      NP-OBJ:np))
```

```
Rule :
→ (SENT:* NP-SUJ VPpart-MOD VN NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VPpart-MOD:((np\\*)/np)/((np\\*)/np)" "VN:(np\\*)/np") NP-OBJ:np))
```

```
Rule :
→ (SENT:* PP NP-SUJ tree)
← (SENT:* (:np PP:np/np NP-SUJ:np) SENT:np\\*)
Rule :
→ (SENT:* CLR tree)
← (SENT:* CLR:cl_r SENT:cl_r\\*)
Rule :
→ (SENT:* tree (COORD CC tree VPpart-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* SENT:* VPpart-MOD:*\\*)))
Rule :
→ (PP-ATS:* P+D (NP NC))
← (PP-ATS:* P+D:*/n (NP:n NC:n))
Rule :
→ (PP-OBJ:* P+D NP)
← (PP-OBJ:* P+D:*/n NP:n)
Rule : use-ponct
→ (NP:* (PONCT -LBR-) tree (PONCT -RBR-))
← (NP:*
    (:*/pf ("PONCT:(*/pf)/np" "-LBR-:(*/pf)/np") NP:np)
    (PONCT:pf -RBR-:pf))
Rule :
→ (NP-SUJ:* tree ADV)
← (NP-SUJ:* NP:* ADV:*\\*)
Rule :
→ (NP-MOD:* tree NP-MOD)
← (NP-MOD:* NP:* NP-MOD:*\\*)
Rule :
→ (VN:* tree PP-MOD)
← (VN:* VN:* PP-MOD:*\\*)
```

```
Rule :
→ (Srel-MOD:* NP-SUJ VN)
← (Srel-MOD:* NP-SUJ:np VN:np\\*)
Rule :
→ (VPinf-OBJ:* P VN ADV)
← (VPinf-OBJ:* "P:*/(np\\s_i)"
    (:np\\s_i VN:np\\s_i "ADV:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf-MOD:* tree NP-OBJ)
← (VPinf-MOD:* VPinf:*/np NP-OBJ:np)
Rule :
→ (VPinf-OBJ:* tree ADV)
← (VPinf-OBJ:* SENT:* ADV:*\\*)
Rule :
→ (VPinf-ATS:* tree VPinf-OBJ)
← (VPinf-ATS:* "VPinf:*/(np\\s_i)" VPinf-OBJ:np\\s_i)
Rule :
→ (AP:* tree (COORD CC ADJ))
← (AP:* AP:* (COORD:*\\* "CC:(*\\*)/*" ADJ:*))
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) NP-ATS)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p "VPP:(np\\s_p)/np" NP-ATS:np)))
Rule : use-ponct
→ (SENT:* NP-SUJ PONCT (VN tree VPP) VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "PONCT:(np\\*)/(np\\*)"
      (VN:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
          "VPP:(np\\s_p)/(np\\s_i)" VPinf-OBJ:np\\s_i))))
```

```
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_de" "VN:(np\\s)/pp_de" "PP-MOD:((np\\s)/pp_de)\\((np\\*)/pp_de)") PP-DE_OBJ:pp_de))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD Ssub-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "PP-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
```

Rule :
→ (SENT:* NP-SUJ VN ADV ADV NP-ATS)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np" (":(np*)/np" "VN:(np*)/np" "ADV:((np*)/np)\\((np*)/np)")
"ADV:((np*)/np)\\((np*)/np)")
NP-ATS:np))

Rule :
→ (SENT:* NP-SUJ VN PP-DE_OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np* (":(np*)/np"
"VN:((np*)/np)/pp_de" PP-DE_OBJ:pp_de)
NP-OBJ:np))

Rule : use-ponct
→ (SENT:* VN NP-OBJ PONCT NP-OBJ)
← (SENT:* VN:*/np (:np NP-OBJ:np
(:np\\np "PONCT:(np\\np)/np" (:np NP-OBJ:np))))

Rule :
→ (SENT:* VN NP-OBJ VPinf-DE_OBJ)
← (SENT:*
(":*/(np\\s_i)" "VN:(*/(np\\s_i))/np" NP-OBJ:np)
VPinf-DE_OBJ:np\\s_i)
Rule :
→ (SENT:* NP-SUJ (COORD CC NP) tree)
← (SENT:*
(:np NP-SUJ:np (COORD:np\\np "CC:(np\\np)/np" NP:np))
SENT:np*)
Rule :
→ (SENT:* PP-P_OBJ tree)
← (SENT:* PP-P_OBJ:pp SENT:pp*)

Rule :
→ (SENT:* tree VPinf-OBJ (COORD CC VPinf))
← (SENT:* "SENT:*/(np\\s_i)"
(:np\\s_i VPinf-OBJ:np\\s_i ("COORD:(np\\s_i)\\(np\\s_i)" "CC:((np\\s_i)\\(np\\s_i))/(np\\s_i)" VPinf:np\\s_i)))

Rule :
→ (SENT:* VN NP-ATS (Srel tree))
← (SENT:* (:*/s "VN:(*/s)/np" NP-ATS:np) (Srel:s SENT:s))
Rule :
→ (SENT:* tree (COORD CC VN PP-A_OBJ))
← (SENT:* SENT:* (COORD:** "CC:(**)/*"
(:* VN:*/pp_a PP-A_OBJ:pp_a)))
Rule :
→ (PP-DE_OBJ:* P+D NP (COORD CC PP))
← (PP-DE_OBJ:* (:* P+D:*/n NP:n)
(COORD:** "CC:(**)/*" PP:*))
Rule :
→ (PP-MOD:* P AP)
← (PP-MOD:* "P:*/(n\\n)" AP:n\\n)
Rule :
→ (PP:* P tree VPpart)
← (PP:* P:*/np (:np NP:np VPpart:np\\np))
Rule :
→ (NP-ATS:* ADV tree)
← (NP-ATS:* ADV:*/n NP:n)
Rule :
→ (NP-ATS:* tree NPP)
← (NP-ATS:* NP:n NPP:n*)
Rule :
→ (VN:* CLR-SUJ tree)
← (VN:* CLR-SUJ:cl_r VN:cl_r*)
Rule :
→ (Srel:* CS tree)
← (Srel:* CS:*/* SENT:*)
Rule :
→ (VPpart-MOD:* P VN PP-P_OBJ)
← (VPpart-MOD:* "P:*/(np\\s)"
(:np\\s "VN:(np\\s)/pp" PP-P_OBJ:pp))
Rule :
→ (VPinf-OBJ:* VN ADV PP-MOD)
← (VPinf-OBJ:* (:* VN:* ADV:**) PP-MOD:**)

Rule :
→ (VPinf-OBJ:* VINF NP)
← (VPinf-OBJ:* VINF:*/np NP:np)
Rule :
→ (VPinf:* VINF NP)
← (VPinf:* VINF:*/np NP:np)
Rule :
→ (VPinf:* tree VPinf-ATO)
← (VPinf:* "VPinf:*/(np\\s_i)" VPinf-ATO:np\\s_i)
Rule :
→ (VPinf-ATS:* tree PP-MOD)
← (VPinf-ATS:* SENT:* PP-MOD:**)
Rule :
→ (AP-ATS:* tree (COORD CC AP))
← (AP-ATS:* AP:* (COORD:** "CC:(**)/*" AP:*))
Rule :
→ (AP-MOD:* tree PP)
← (AP-MOD:* AP:* PP:**)
Rule :
→ (Ssub-MOD:* P tree)
← (Ssub-MOD:* P:*/s SENT:s)
Rule :
→ (AdP-MOD:* NP ADV ADV)
← (AdP-MOD:* (:* NP:np ADV:np*) ADV:**)
Rule :
→ (AdP-MOD:* tree Srel)
← (AdP-MOD:* AdP:*/s Srel:s)
Rule :
→ (COORD-MOD:* CC PP)
← (COORD-MOD:* CC:*/pp PP:pp)
Rule :
→ (SENT ADV tree)
← (SENT:s ADV:s/s SENT:s)
Rule : use-ponct
→ (SENT PONCT PONCT NP-MOD)
← (SENT:np PONCT:np/np (:np PONCT:np/np NP-MOD:np))

Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (VN:np* "VN:(np*)/(np\\s_p)" (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
(:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "PP-MOD:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-P_OBJ:pp))))

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-A_OBJ NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np"
        "VN:((np\\*)/np)/pp_a" PP-A_OBJ:pp_a)
      NP-OBJ:np))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "NP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-A_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "PP-MOD:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "NP-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD VPinf-OBJ)
← (SENT:* CLS-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "PP-MOD:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-OBJ:np\\s_i))
```

```
Rule :
→ (SENT:* AP-ATS tree)
← (SENT:* AP-ATS:n\\n "SENT:(n\\n)\\*")
Rule :
→ (SENT:* NP-OBJ tree)
← (SENT:* NP-OBJ:np SENT:np\\*)
Rule :
→ (SENT:* ADJ tree)
← (SENT:* ADJ:*/* SENT:*)
```

```
Rule :
→ (SENT:* NP NP PP)
← (SENT:* (:np NP:np/np NP:np) PP:np\\*)
Rule :
→ (SENT:* NP PP PP)
← (SENT:* (:np NP:np PP:np\\np) PP:np\\*)
Rule :
→ (SENT:* VPP AdP)
← (SENT:* VPP:* AdP:*\\*)
```

```
Rule :
→ (SENT:* tree (COORD CC CC Ssub))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/cs" "CC:((*\\*)/cs)/((*\\*)/cs)" "CC:(*\\*)/cs") Ssub:cs))
```

```
Rule :
→ (SENT:* VN PP)
← (SENT:* VN:*/pp PP:pp)
Rule :
→ (PP:* DET NP)
← (PP:* DET:*/n NP:n)
Rule :
→ (PP:* P DET NP)
← (PP:* P:*/np (:np DET:np/n NP:n))
Rule :
→ (PP:* P ADV PP)
← (PP:* "P:*/(n\\n)" (:n\\n "ADV:(n\\n)/(n\\n)" PP:n\\n))
Rule :
→ (NP:* (COORD CC ADJ) tree)
← (NP:* (COORD:*/n "CC:(*/n)/(*/n)" ADJ:*/n) NP:n)
```

```
Rule :
→ (NP:* NC (COORD CC NP))
← (NP:* NC:n (COORD:n\\* "CC:(n\\*)/np" NP:np))
Rule : use-ponct
→ (NP:* tree PONCT (PP P NP) PP PONCT)
← (NP:* NP:np (:np\\* (":(np\\*)/pf"
        "PONCT:((np\\*)/pf)/(n\\n)"
        (PP:n\\n "P:(n\\n)/np" (:np NP:np PP:np\\np)))
      PONCT:pf))
Rule :
→ (NP:* tree (COORD CC NP-MOD))
← (NP:* NP:n (COORD:n\\* "CC:(n\\*)/np" NP-MOD:np))
```

```
Rule :
→ (NP:* NC (COORD CC PP) (COORD CC PP))
← (NP:* NC:n
    (COORD:n\\* "CC:(n\\*)/(n\\n)" (:n\\n PP:n\\n ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/(n\\n)" PP:n\\n))))
```

```
Rule :
→ (NP-OBJ-MOD:* PONCT NP-OBJ PONCT)
← (NP-OBJ-MOD:* (:*/pf "PONCT:(*/pf)/np" NP-OBJ:np)
    PONCT:pf)
```

```
Rule :
→ (VN:* V ADV VINF)
← (VN:* (":*/(np\\s_i)" "V:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VINF:np\\s_i)
```

Rule :
→ (VN:* tree ADV ADJ)
← (VN:* "VN:*/(n/n)" (:n/n "ADV:(n/n)/(n/n)" ADJ:n/n))
Rule :
→ (VPpart-MOD:* ADV VPP)
← (VPpart-MOD:* ADV:*/* VPP:*)
Rule :
→ (VPinf-OBJ:* VN NP-OBJ NP-MOD)
← (VPinf-OBJ:* (:* VN:*/np NP-OBJ:np) NP-MOD:**)
Rule :
→ (VPinf-OBJ:* VN PP-DE_OBJ PP-MOD)
← (VPinf-OBJ:* (:* VN:*/pp_de PP-DE_OBJ:pp_de)
PP-MOD:**)
Rule :
→ (VPinf:* tree NP)
← (VPinf:* VPinf:*/np NP:np)
Rule : use-ponct
→ (VPinf:* PONCT tree)
← (VPinf:* PONCT:*/* VPinf:*)
Rule :
→ (AP-MOD:* tree Ssub)
← (AP-MOD:* AP:*/cs Ssub:cs)
Rule :
→ (Ssub:* ADV CS tree)
← (Ssub:* ADV:*/* (:* CS:*/s SENT:s))

Rule :
→ (AdP:* P tree)
← (AdP:* P:*/* AdP:*)
Rule :
→ (AdP:* ADV (COORD CC ADV))
← (AdP:* ADV:* (COORD:** "CC:(**)/*" ADV:*))
Rule :
→ (AdP:* DET tree)
← (AdP:* DET:*/* AdP:*)
Rule : use-ponct
→ (SENT:* tree (Sint PONCT tree))
← (SENT:* SENT:* (Sint:** "PONCT:(**)/s" SENT:s))
Rule : use-ponct
→ (SENT NP Srel PONCT)
← (TEXT:txt NP:np
(SENT:np\\txt Srel:s "PONCT:s\\(np\\txt)"))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP ADV VPP) PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np* "VN:(np*)/(np\\s_p)"
(:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
"ADV:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
"VPP:(np\\s_p)/pp" PP-P_OBJ:pp)))))
Rule :
→ (SENT tree PP-MOD)
← (SENT:s SENT:s PP-MOD:s\\s)

Rule :
→ (SENT:* (VN CLS-SUJ tree) AdP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np* (":(np*)/np" "VN:(np\\s)/np" "AdP-MOD:((np\\s)/np)\\((np*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN ADV ADV AP-ATS)
← (SENT:* NP-SUJ:np (:np* (":(np*)/(n\\n)"
(":(np*)/(n\\n)" "VN:(np*)/(n\\n)" "ADV:((np*)/(n\\n))\\((np*)/(n\\n))")
"ADV:((np*)/(n\\n))\\((np*)/(n\\n))")
AP-ATS:n\\n))

Rule :
→ (SENT:* NP-SUJ VN VPpart-ATO)
← (SENT:* NP-SUJ:np
(:np* "VN:(np*)/(np\\s_p)" VPpart-ATO:np\\s_p))
Rule :
→ (SENT:* tree NP-OBJ (COORD CC VPP NP))
← (SENT:* SENT:*/np (:np NP-OBJ:np (COORD:np\\np
"CC:(np\\np)/np" (:np VPP:np/np NP:np))))
Rule :
→ (SENT:* tree NP-P_OBJ)
← (NP:np SENT:*/np NP-P_OBJ:np)
Rule :
→ (SENT:* tree (COORD CC NP VPinf))
← (SENT:* SENT:*
(COORD:** "CC:(**)/np" (:np NP:np VPinf:np\\np)))
Rule :
→ (SENT:* VN AdP)
← (VPinf:np\\np VN:* AdP:**)
Rule :
→ (SENT:* Ssub-SUJ tree)
← (SENT:* Ssub-SUJ:cs SENT:cs*)
Rule :
→ (SENT:* tree (COORD CC tree VPinf-DE_OBJ))
← (SENT:* SENT:* (COORD:** "CC:(**)/*"
(:* "SENT:*/(np\\s_i)" VPinf-DE_OBJ:np\\s_i)))
Rule :
→ (SENT:* PP-SUJ VN)
← (VPinf-DE_OBJ:np\\s_i PP-SUJ:pp VN:pp*)
Rule :
→ (SENT:* PP VN)
← (SENT:* PP:pp VN:pp*)

Rule :
→ (SENT AP)
← (SENT:n\\n AP:n\\n)
Rule :
→ (PP-OBJ:* P+D (NP NC))
← (PP-OBJ:* P+D:*/n (NP:n NC:n))
Rule :
→ (PP:* tree (COORD CC CC PP))
← (PP:* PP:* (COORD:** (":(**)/*"
"CC:((**)/*)/((**)/*)" "CC:(**)/*")
PP:*))
Rule :
→ (PP:* P VN)
← (PP:* "P:*/(np\\s)" VN:np\\s)
Rule :
→ (NP:* (COORD CC DET) tree)
← (NP:* (COORD:*/n "CC:(*/n)/(*/n)" DET:*/n) NP:n)
Rule : use-ponct
→ (NP:* NC AP AP PONCT)
← (NP:* (:n NC:n AP:n\\n)
(:n* AP:n\\n "PONCT:(n\\n)\\(n*)"))
Rule : use-ponct
→ (NP:* NC AP NP PONCT)
← (NP:* (:n NC:n AP:n\\n)
(:n* NP:np "PONCT:np\\(n*)"))
Rule :
→ (NP:* tree (COORD CC NP NP))
← (NP:* NP:*
(COORD:** "CC:(**)/np" (:np NP:np NP:np\\np)))
Rule :
→ (NP:* tree Srel (COORD CC tree))
← (NP:* NP:* (:** Srel:** ("COORD:(**)\\(**)"
"CC:((**)\\(**))/s" SENT:s)))

```
Rule :
→ (NP-SUJ:* tree ADJ)
← (SENT:s NP:n ADJ:n\\*)
Rule :
→ (VN:* tree NC)
← (VN:* VN:*/n NC:n)
Rule :
→ (VN:* V V)
← (VN:* "V:*/(np\\s)" V:np\\s)
Rule :
→ (Srel:* (PP-A_OBJ P (NP PROREL)) tree)
← (Srel:* (PP-A_OBJ:pp/np P:pp/np)
     ("NP:(pp/np)\\*" "PROREL:((pp/np)\\*)/s" SENT:s))
Rule :
→ (Srel:* PP-MOD VN NP-OBJ)
← (Srel:* PP-MOD:*/* (:* VN:*/np NP-OBJ:np))
Rule :
→ (Srel-MOD:* NP-MOD tree)
← (Srel-MOD:* NP-MOD:*/* SENT:*)
Rule :
→ (Srel:* NP NP)
← (Srel:* NP:*/np NP:np)
Rule :
→ (VPpart:* AdP tree)
← (VPpart:* AdP:*/* VPpart:*)
Rule :
→ (VPpart:* ADJ tree)
← (VPpart:* ADJ:*/* VPpart:*)
```

```
Rule :
→ (VPinf-DE_OBJ:* P VN PP-DE_OBJ)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)"
     (:np\\s_i "VN:(np\\s_i)/pp_de" PP-DE_OBJ:pp_de))
Rule :
→ (VPinf-MOD:* tree NP-MOD)
← (VPinf-MOD:* VPinf:* NP-MOD:*\\*)
Rule :
→ (VPinf:* tree Ssub-OBJ)
← (VPinf:* VPinf:*/cs Ssub-OBJ:cs)
Rule :
→ (VPinf-DE_OBJ:* tree (COORD CC VPinf))
← (VPinf-DE_OBJ:* VPinf:*
     (COORD:*\\* "CC:(*\\*)/*" VPinf:*))
Rule :
→ (AP-ATS:* ADJ Ssub)
← (AP-ATS:* ADJ:*/cs Ssub:cs)
Rule : use-ponct
→ (AP:* PONCT tree PONCT)
← (AP:* (:*/gf "PONCT:(*/gf)/*" AP:*) PONCT:gf)
Rule :
→ (AP-ATS:* tree ADV)
← (AP-ATS:* AP:* ADV:*\\*)
Rule :
→ (AP-ATS:* ADV tree)
← (AP-ATS:* ADV:*/* AP:*)
Rule :
→ (Ssub-MOD:* ADVWH tree)
← (Ssub-MOD:* ADVWH:*/s SENT:s)
Rule :
→ (Ssub-OBJ:* PP-MOD tree)
← (Ssub-OVJ:* PP-MOD:*/s SENT:s)
```

```
Rule :
→ ("COORD:(n\\n)\\(n\\n)" CC tree)
← ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/(n\\n)" AP:n\\n)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD PP-A_OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
       (":(np\\s_p)/pp_a" "VPP:(np\\s_p)/pp_a" "PP-MOD:((np\\s_p)/pp_a)\\((np\\s_p)/pp_a)") PP-A_OBJ:pp_a)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
       (":(np\\s_p)/pp_de" "VPP:(np\\s_p)/pp_de" "ADV:((np\\s_p)/pp_de)\\((np\\s_p)/pp_de)") PP-DE_OBJ:pp_de)))
Rule :
→ (SENT ADV tree)
← (PP-DE_OBJ:pp_de ADV:s/s SENT:s)
```

```
Rule :
→ (SENT:* NP-SUJ VN NP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "NP-MOD:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-OBJ:np\\s_i))
```

```
Rule :
→ (SENT:* VPinf-SUJ VN NP-ATS)
← (SENT:* VPinf-SUJ:np\\s_i
     (":(np\\s_i)\\*" "VN:((np\\s_i)\\*)/np" NP-ATS:np))

Rule : use-ponct
→ (SENT:* VN NP-OBJ PONCT NP-OBJ PONCT NP-OBJ)
← (SENT:* VN:*/np (:np NP-OBJ:np (:np\\np
        "PONCT:(np\\np)/np" (:np NP-OBJ:np
          (:np\\np "PONCT:(np\\np)/np" NP-OBJ:np)))))

Rule : use-ponct
→ (SENT:* VN NP-ATS PONCT NP-ATS)
← (SENT:* VN:*/np (:np NP-ATS:np
      (:np\\np "PONCT:(np\\np)/np" NP-ATS:np)))
```

```
Rule :
→ (SENT:* ADJ-MOD tree)
← (NP-ATS:np ADJ-MOD:*/* SENT:*)

Rule :
→ (SENT:* AdP tree)
← (SENT:* AdP:*/* SENT:*)

Rule :
→ (SENT:* tree AdP-P_OBJ)
← (SENT:* "SENT:*/(s\\s)" AdP-P_OBJ:s\\s)

Rule :
→ (SENT:* (COORD CC PP) tree)
← (SENT:* (COORD:*/* "CC:(*/*)/pp" PP:pp) SENT:*)
```

Rule :
→ (SENT:* tree (COORD CC VN ADV))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* VN:* ADV:**)))
Rule :
→ (SENT:* tree (COORD CC tree ADV))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* SENT:* ADV:**)))

Rule :
→ (PP-MOD:* P AdP)
← (ADV:** "P:*/(n\\n)" AdP:n\\n)

Rule :
→ (PP:* P ADV P NP)
← (PP:* (:*/np P:*/np (":(*/np)\\(*/np)" "ADV:((*/np)\\(*/np))/(*/np)" P:*/np)) NP:np)

Rule :
→ (PP:* P VPinf)
← (PP:* "P:*/(np\\s_i)" VPinf:np\\s_i)

Rule :
→ (PP:* P Ssub)
← (PP:* P:*/cs Ssub:cs)

Rule :
→ (NP:* tree VPinf-MOD)
← (NP:* NP:* VPinf-MOD:**)

Rule : use-ponct
→ (NP:* tree (NC-MOD PONCT NC PONCT))
← (NP:* NP:* (NC-MOD:**
(":(**)/pf" "PONCT:((**)/pf)/n" NC:n)
PONCT:pf))

Rule : use-ponct
→ (NP:* tree (COORD CC NP PONCT))
← (NP:* NP:n
(COORD:n* "CC:(n*)/np" (:np NP:np PONCT:np\\np)))

Rule :
→ (NP:* NC CC)
← (PONCT:np\\np NC:n CC:n*)
Rule :
→ (NP-SUJ:* NPP NC)
← (NP-SUJ:* NPP:*/n NC:n)
Rule :
→ (NP-OBJ:* tree ADV)
← (NP-OBJ:* NP:n ADV:n*)
Rule :
→ (NP-OBJ:* NC NPP NPP)
← (NP-OBJ:* NC:*/np (:np NPP:np/np NPP:np))
Rule :
→ (NP-MOD:* P tree)
← (NP-MOD:* P:*/n NP:n)
Rule :
→ (NP-MOD:* tree ADJ)
← (NP-MOD:* NP:n ADJ:n*)
Rule :
→ (VN:* tree AdP)
← (VN:* VN:* AdP:**)

Rule :
→ (VN:* tree ADV VINF)
← (VN:* (":*/(np\\s_i)" "VN:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VINF:np\\s_i)

Rule :
→ (VPpart-MOD:* VPP VPinf)
← (VPpart-MOD:* VPP:np\\s_p "VPinf:(np\\s_p)*")
Rule :
→ (VPpart-MOD:* tree Ssub-MOD)
← (VPpart-MOD:* VPpart:* Ssub-MOD:**)

Rule :
→ (VPinf-OBJ:* VN ADV NP-OBJ PP-MOD)
← (VPinf-OBJ:*
(:* (:*/np VN:*/np "ADV:(*/np)\\(*/np)") NP-OBJ:np)
PP-MOD:**)

Rule :
→ (VPinf-OBJ:* P VN AdP-MOD)
← (VPinf-OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "AdP-MOD:(np\\s_i)\\(np\\s_i)"))

Rule :
→ (VPinf:* P VN VPinf-A_OBJ)
← (VPinf:* "P:*/(np\\s_i)" (:np\\s_i
"VN:(np\\s_i)/(np\\s_i)" VPinf-A_OBJ:np\\s_i))
Rule :
→ (VPinf-OBJ:* NP-MOD tree)
← (VPinf-OBJ:* NP-MOD:*/* SENT:*)
Rule :
→ (AP:* ADJ NC)
← (AP:* ADJ:*/n NC:n)
Rule :
→ (AP:* PP (COORD CC tree))
← (AP:* PP:n\\n
("COORD:(n\\n)*" "CC:((n\\n)*)/(n\\n)" AP:n\\n))
Rule : use-ponct
→ (ADJ-MOD:* PONCT ADJ PONCT)
← (ADJ-MOD:* (:*/pf "PONCT:(*/pf)/(n/n)" ADJ:n/n)
PONCT:pf)

Rule :
→ (Ssub:* ADV tree)
← (Ssub:* ADV:*/s SENT:s)
Rule : use-ponct
→ (Ssub-MOD:* PONCT Ssub PONCT)
← (Ssub-MOD:* (:*/pf "PONCT:(*/pf)/cs" Ssub:cs) PONCT:pf)
Rule :
→ (AdP:* ADV ADV ADV)
← (AdP:* ADV:*/* (:* ADV:*/* ADV:*))
Rule :
→ (AdP:* tree NP)
← (AdP:* AdP:*/np NP:np)
Rule :
→ (AdP-MOD:* ADV ADV ADV)
← (AdP-MOD:* ADV:*/* (:* ADV:*/* ADV:*))
Rule :
→ (AdP-P_OBJ:* tree)
← (ADV:* AdP:*)

```
Rule :
→ ("COORD:(n\\n)\\(n\\n)" CC AP)
← ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/(n\\n)" AP:n\\n)
Rule :
→ (SENT tree NP-MOD)
← (SENT:s SENT:s NP-MOD:s\\s)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/cs" "VPP:(np\\s_p)/cs" "PP-MOD:((np\\s_p)/cs)\\((np\\s_p)/cs)") Ssub-OBJ:cs)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/pp_a" "VPP:(np\\s_p)/pp_a" "NP-MOD:((np\\s_p)/pp_a)\\((np\\s_p)/pp_a)") PP-A_OBJ:pp_a)))
Rule :
→ (SENT:* NP-SUJ VN ADV PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np\\*)/pp)") PP-OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD PP-DE_OBJ)
← (SENT:* CLS-SUJ:np
    (:np\\* (":(np\\*)/pp_de" "VN:(np\\s)/pp_de" "PP-MOD:((np\\s)/pp_de)\\((np\\*)/pp_de)") PP-DE_OBJ:pp_de))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)"
       "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
      NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD AP-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "PP-MOD:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV VPinf-ATS)
← (SENT:* CLS-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))") VPinf-ATS:np\\s_i))
Rule :
→ (SENT:* (COORD CC VPinf) tree)
← (SENT:* (COORD:*/s "CC:(*/s)/(np\\s_i)" VPinf:np\\s_i)
    SENT:s)
Rule :
→ (SENT:np\\s tree (COORD CC (VN tree) VPinf-OBJ))
← (SENT:np\\s SENT:np\\s
    ("COORD:(np\\s)\\(np\\s)" "CC:((np\\s)\\(np\\s))/(np\\s)" (VN:np\\s "VN:(np\\s)/(np\\s_i)" VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* VN VPpart-ATO)
← (VPinf-OBJ:np\\s_i "VN:*/(n\\n)" VPpart-ATO:n\\n)
Rule :
→ (SENT:* VN VPP)
← (SENT:* "VN:*/(np\\s_p)" VPP:np\\s_p)
Rule :
→ (SENT:* tree (COORD CC VN ADV PP-A_OBJ))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*" (:*
       (:*/pp_a VN:*/pp_a "ADV:(*/pp_a)\\(*/pp_a)")
       PP-A_OBJ:pp_a)))
Rule :
→ (SENT:* VPpart tree)
← (PP-A_OBJ:pp_a VPpart:np\\s_p "SENT:(np\\s_p)\\*")
Rule :
→ (SENT:* PP PP)
← (SENT:* PP:pp PP:pp\\*)
Rule :
→ (SENT:* tree NP-DE_OBJ)
← (SENT:* SENT:*/np NP-DE_OBJ:np)
Rule :
→ (SENT:* CLO VN)
← (SENT:* "CLO:*/(*/np)" VN:*/np)
Rule :
→ (SENT:* PRO VN)
← (SENT:* PRO:np VN:np\\*)
Rule :
→ (PP:* P AdP)
← (PP:* "P:*/(n\\n)" AdP:n\\n)
Rule :
→ (NP:* tree VPP)
← (NP:* NP:* VPP:*\\*)
Rule :
→ (NP:* tree Ssub-MOD)
← (NP:* NP:* Ssub-MOD:*\\*)
Rule :
→ (NP:* NPP DET)
← (NP:* NPP:n DET:n\\*)
Rule : use-ponct
→ (NP:* tree PP PONCT (COORD CC PP))
← (NP:* NP:n
    (:n\\* (:n\\n PP:n\\n "PONCT:(n\\n)\\(n\\n)") ("COORD:(n\\n)\\(n\\*)" "CC:((n\\n)\\(n\\*))/(n\\n)" PP:n\\n)))
```

```
Rule :
→ (NP:* NC P+D)
← (PP:n\\n NC:n P+D:n\\*)
Rule :
→ (NP:* NPP NC)
← (NP:* NPP:*/n NC:n)
Rule :
→ (NP-OBJ:* tree Ssub)
← (NP-OBJ:* NP:n Ssub:n\\*)
Rule :
→ (NP-OBJ:* tree (COORD CC NP PP))
← (NP-OBJ:* (:* NP:*
       (COORD:*\\* "CC:(*\\*)/np" (:np NP:np PP:np\\np))))
Rule :
→ (NP-OBJ:* NC ADJ NC)
← (PP:np\\np NC:*/n (:n ADJ:n/n NC:n))
Rule :
→ (NP-ATS:* ADJ DET tree)
← (NP-ATS:* ADJ:*/* (:* DET:*/n NP:n))
Rule :
→ (NP-ATO:* DET tree)
← (NP-ATO:* DET:*/n NP:n)
Rule :
→ (VN:* tree CLR)
← (VN:* VN:*/cl_r CLR:cl_r)
Rule :
→ (VN:* tree CLO-A_OBJ)
← (VN:* VN:*/pp CLO-A_OBJ:pp)
Rule :
→ (VN:* P V)
← (VN:* P:pp V:pp\\*)
Rule :
→ (VN:* NC tree)
← (VN:* NC:n VN:n\\*)
Rule :
→ (VN:* PRO tree)
← (VN:* PRO:np VN:np\\*)
Rule :
→ (Srel:* (PP-P_OBJ P (NP PROREL)) tree)
← (Srel:* (PP-P_OBJ:* P:pp/np
       ("NP:(pp/np)\\*" "PROREL:((pp/np)\\*)/s" SENT:s)))
Rule :
→ (VPpart-ATS:* VPP VPinf)
← (SENT:s VPP:np\\s_p "VPinf:(np\\s_p)\\*")
Rule :
→ (VPpart:* tree VPinf-MOD)
← (VPpart:* VPpart:* VPinf-MOD:*\\*)
Rule :
→ (VPinf-OBJ:* VN NP-MOD NP-OBJ)
← (VPinf-OBJ:* (:*/np VN:*/np "NP-MOD:(*/np)\\(*/np)")
     NP-OBJ:np)
Rule :
→ (VPinf:* P VN NP-MOD)
← (VPinf:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "NP-MOD:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf-MOD:* ADVWH tree)
← (VPinf-MOD:* ADVWH:*/* VPinf:*)
Rule :
→ (VPinf-MOD:* tree Ssub-MOD)
← (VPinf-MOD:* VPinf:* Ssub-MOD:*\\*)
Rule :
→ (VPinf:* tree AP-MOD)
← (VPinf:* VPinf:* AP-MOD:*\\*)
Rule :
→ (AP-ATS:* ADV ADV ADJ)
← (AP-ATS:* ADV:*/* (:* "ADV:*/(n/n)" ADJ:n/n))
Rule :
→ (AP:* tree Srel)
← (AP:* AP:*/s Srel:s)
Rule :
→ (AP-ATS:* tree VPinf-OBJ)
← (AP-ATS:* "AP:*/(np\\s_i)" VPinf-OBJ:np\\s_i)
Rule :
→ (AP:* CC tree)
← (AP:* "CC:*/(n\\n)" AP:n\\n)
Rule :
→ (AP:* tree Sint-MOD)
← (AP:* AP:* Sint-MOD:*\\*)
Rule :
→ (AdP:* ADV AP)
← (AdP:* "ADV:*/(n\\n)" AP:n\\n)
Rule :
→ (COORD:* CC ADV)
← (COORD:* "CC:*/(s/s)" ADV:s/s)
Rule :
→ (PP-SUJ:* P tree)
← (PP-SUJ:* P:*/np NP:np)
Rule :
→ (SENT tree PP-MOD)
← (SENT:s SENT:s PP-MOD:s\\s)
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
         (VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-ATS:pp))))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/pp_de" "VPP:(np\\s_p)/pp_de" "PP-MOD:((np\\s_p)/pp_de)\\((np\\s_p)/pp_de)") PP-DE_OBJ:pp_de)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "NP-MOD:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-P_OBJ:pp)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV AP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(n\\n)" "VPP:(np\\s_p)/(n\\n)" "ADV:((np\\s_p)/(n\\n))\\((np\\s_p)/(n\\n))") AP-ATS:n\\n)))
```

```
Rule :
→ (SENT:* COORD-MOD tree)
← (AP-ATS:n\\n COORD-MOD:*/* SENT:*)

Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "PP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "ADV:((np\\s)/pp)\\((np\\*)/pp)") PP-OBJ:pp))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
       (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "AdP-MOD:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))")
       VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* NP-SUJ VN ADV VPpart-ATS)
← (SENT:* NP-SUJ:np (:np\\*
       (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "ADV:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
       VPpart-ATS:np\\s_p))

Rule :
→ (SENT:* NP-SUJ VN NP-OBJ VPinf-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/np"
       (:np NP-OBJ:np VPinf-DE_OBJ:np\\np)))
Rule :
→ (SENT:* tree VPP NP-OBJ)
← (SENT:* "SENT:*/(np\\s)"
     (:np\\s "VPP:(np\\s)/np" NP-OBJ:np))
Rule :
→ (SENT:* tree NP-OBJ (COORD CC NP NP))
← (SENT:* SENT:*/np (:np NP-OBJ:np (COORD:np\\np
         "CC:(np\\np)/np" (:np NP:np NP:np\\np))))
Rule :
→ (SENT:* tree NP (COORD CC NP))
← (SENT:* SENT:*/np
     (:np NP:np (COORD:np\\np "CC:(np\\np)/np" NP:np)))

Rule :
→ (SENT:* VPinf PONCT VPinf)
← (SENT:* VPinf:np\\s_i (":(np\\s_i)\\*" "PONCT:((np\\s_i)\\*)/(np\\s_i)" VPinf:np\\s_i))

Rule :
→ (SENT:* NP VPpart)
← (SENT:* NP:np VPpart:np\\*)
Rule :
→ (SENT:* VN NP-SUJ VN)
← (SENT:* VN:*/* (:* NP-SUJ:np VN:np\\*))

Rule :
→ (SENT:* tree (COORD CC ADV Ssub))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/cs" "CC:(*\\*)/cs" "ADV:((*\\*)/cs)\\((*\\*)/cs)") Ssub:cs))

Rule :
→ (SENT:* tree Ssub-SUJ)
← (SENT:* SENT:*/cs Ssub-SUJ:cs)
Rule :
→ (SENT:* tree (COORD CC tree NP-ATS))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* SENT:*/np NP-ATS:np)))
Rule :
→ (SENT:* tree (COORD CC tree VPinf-ATS))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
       (:* "SENT:*/(np\\s_i)" VPinf-ATS:np\\s_i)))
Rule :
→ (SENT:* tree (COORD CC tree PP-ATS))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/*" (:* SENT:*/pp PP-ATS:pp)))
Rule :
→ (SENT:* VN VN)
← (PP-ATS:pp "VN:*/(np\\s)" VN:np\\s)
Rule :
→ (SENT:* VN (COORD CC NP))
← (SENT:* VN:*/np (COORD:np CC:np/np NP:np))

Rule :
→ (PP:* P ADV VPinf)
← (PP:* (":*/(np\\s_i)" "P:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VPinf:np\\s_i)

Rule :
→ (PP:* NP tree)
← (PP:* NP:np PP:np\\*)
Rule :
→ (PP:* tree COORD-MOD)
← (PP:* PP:* COORD-MOD:*\\*)
Rule :
→ (PP:* P ADJ tree)
← (PP:* P:*/np (:np ADJ:np AP:np\\np))

Rule :
→ (NP:* tree ADV (COORD CC PP))
← (NP:* NP:* (:*\\* ADV:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" PP:*\\*)))
```

```
Rule :
→ (NP:* NPP AP)
← (PP:*\\* NPP:n AP:n\\*)
Rule :
→ (NP:* tree PREF)
← (NP:* NP:n PREF:n\\*)
Rule :
→ (NP:* tree NP-SUJ)
← (NP:* NP:*/np NP-SUJ:np)
Rule :
→ (NP-OBJ:* NC NC)
← (NP-OBJ:* NC:n NC:n\\*)
Rule :
→ (NP-ATS:* ADJ tree)
← (NP-ATS:* ADJ:*/n NP:n)
Rule :
→ (NP-ATS:* tree VPinf)
← (NP-ATS:* NP:n VPinf:n\\*)
Rule :
→ (NP-SUJ-MOD:* PONCT NP-SUJ PONCT)
← (NP-SUJ-MOD:* (:*/pf "PONCT:(*/pf)/np" NP-SUJ:np)
     PONCT:pf)
Rule :
→ (NP-ATO:* tree)
← (NP-ATO:* NP:*)
Rule :
→ (NP:* NC VPinf-DE_OBJ)
← (NP:* NC:n VPinf-DE_OBJ:n\\*)
Rule :
→ (VN:* CLO-DE_OBJ tree)
← (VN:* "CLO-DE_OBJ:*/(*/pp)" VN:*/pp)
Rule :
→ (VN:* tree CLO-OBJ)
← (VN:* VN:*/np CLO-OBJ:np)
Rule :
→ (VN:* tree CLO)
← (VN:* VN:*/np CLO:np)
Rule :
→ (VN:* tree AdP-MOD)
← (VN:* VN:* AdP-MOD:*\\*)
Rule :
→ (VN:* CLS-MOD tree)
← (VN:* CLS-MOD:*/* VN:*)
Rule :
→ (VPpart-MOD:* VN PP-P_OBJ)
← (VPpart-MOD:* VN:*/pp PP-P_OBJ:pp)
Rule :
→ (VPinf-ATS:* VN PP-MOD)
← (VPinf-ATS:* VN:* PP-MOD:*\\*)
Rule :
→ (VPinf:* VN ADV NP-OBJ PP-MOD)
← (VPinf:*
     (:* (:*/np VN:*/np "ADV:(*/np)\\(*/np)") NP-OBJ:np)
     PP-MOD:*\\*)
Rule :
→ (VPinf:* P VN AdP-MOD)
← (VPinf:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "AdP-MOD:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf:* PP-MOD tree)
← (VPinf:* PP-MOD:*/* VPinf:*)
Rule :
→ (VPinf:* tree PP)
← (VPinf:* "VPinf:*/(n\\n)" PP:n\\n)
Rule :
→ (AP-ATO:* ADJ PP)
← (AP-ATO:* ADJ:*/pp PP:pp)
Rule :
→ (AP-MOD:* tree Srel)
← (AP-MOD:* AP:*/s Srel:s)
Rule :
→ (AP:* tree VPpart)
← (AP:* "AP:*/(np\\s_p)" VPpart:np\\s_p)
Rule :
→ (AP:* tree (COORD CC PP))
← (AP:* AP:* (COORD:*\\* "CC:(*\\*)/*" PP:*))
Rule :
→ (AP:* tree NP-MOD)
← (AP:* AP:* NP-MOD:*\\*)
Rule :
→ (Ssub-ATS:* CS NP)
← (Ssub-ATS:* CS:*/np NP:np)
Rule :
→ (AdP-MOD:* NP tree)
← (AdP-MOD:* NP:np AdP:np\\*)
Rule :
→ (COORD-MOD:* CC Sint)
← (COORD-MOD:* CC:*/s Sint:s)
Rule :
→ (SENT:* (VN tree VPP) NP-OBJ VPinf-A_OBJ)
← (SENT:* (VN:* "VN:*/(np\\s_p)"
     (VPP:np\\s_p ("VPP:(np\\s_p)/(np\\s_i)" "VPP:((np\\s_p)/(np\\s_i))/np" NP-OBJ:np) VPinf-A_OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) AP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
     (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
         "VPP:(np\\s_p)/(n\\n)" AP-ATS:n\\n))))
Rule :
→ (SENT tree NP-OBJ-MOD)
← (SENT:s SENT:s NP-OBJ-MOD:s\\s)
Rule :
→ (SENT:* NP-ATS VN NP-SUJ)
← (SENT:* NP-ATS:np (:np\\* "VN:(np\\*)/np" NP-SUJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP)
← (SENT:* CLS-SUJ:np (:np\\* "VN:(np\\*)/np" NP:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "ADV:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD PP-A_OBJ)
← (SENT:* CLS-SUJ:np
     (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "PP-MOD:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
```

```
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-A_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "NP-MOD:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
       "ADV:((np\\*)/np)\\((np\\*)/np)")
      NP-OBJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-MOD Ssub-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
       "PP-MOD:((np\\*)/cs)\\((np\\*)/cs)")
      Ssub-OBJ:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV VPpart-ATS)
← (SENT:* CLS-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPpart-ATS:np\\s_i))
```

```
Rule : use-ponct
→ (SENT:* NP-SUJ PONCT NP PONCT tree)
← (SENT:* (NP-SUJ:np NP-SUJ:np (:np\\np
       "PONCT:(np\\np)/np" (:np NP:np PONCT:np\\np)))
    SENT:np\\*)
Rule :
→ (SENT:* NP-SUJ VN PP-MOD VN)
← (SENT:* (:* NP-SUJ:np VN:np\\*)
    (:*\\* "PP-MOD:(*\\*)/(*\\*)" VN:*\\*))
Rule :
→ (SENT:* tree NP-OBJ-MOD)
← (SENT:* SENT:* NP-OBJ-MOD:*\\*)
Rule : use-ponct
→ (SENT:* tree PONCT (COORD-OBJ CC VPinf))
← (SENT:* SENT:* (:*\\* "PONCT:(*\\*)/(np\\s_i)"
      (COORD-OBJ:np\\s_i "CC:(np\\s_i)/(np\\s_i)"
       VPinf:np\\s_i)))
```

```
Rule :
→ (SENT:* tree (COORD-P_OBJ CC PP))
← (SENT:* SENT:*/pp (COORD-P_OBJ:pp CC:pp/pp PP:pp))
Rule :
→ (SENT:* tree NP-OBJ (COORD CC NP PP))
← (SENT:* SENT:*/np (:np NP-OBJ:np (COORD:np\\np
       "CC:(np\\np)/np" (:np NP:np PP:np\\np))))
Rule :
→ (SENT:* NP (COORD CC NP))
← (SENT:* NP:np (COORD:np\\* "CC:(np\\*)/np" NP:np))
Rule :
→ (SENT:* NP-SUJ-MOD tree)
← (SENT:* NP-SUJ-MOD:*/* SENT:*)
```

```
Rule :
→ (SENT:* VN ADV NP-ATS (Srel tree))
← (SENT:* (:*/s (":(*/s)/np" "VN:(*/s)/np" "ADV:((*/s)/np)\\((*/s)/np)") NP-ATS:np) (Srel:s SENT:s))
```

```
Rule :
→ (SENT:* VN AP)
← (SENT:* "VN:*/(n\\n)" AP:n\\n)
```

```
Rule :
→ (SENT:* VPP PP PP VPinf)
← (SENT:* (":*/(np\\s_i)" (":(*/(np\\s_i))/pp"
       "VPP:((*/(np\\s_i))/pp)/pp" PP:pp)
     PP:pp)
    VPinf:np\\s_i)
```

```
Rule :
→ (SENT:* VPP ADV VPinf)
← (SENT:* (":*/(np\\s_i)" "VPP:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VPinf:np\\s_i)
```

```
Rule :
→ (SENT:* VINF VPP PP)
← (SENT:* "VINF:*/(np\\s)"
    (:np\\s "VPP:(np\\s)/pp" PP:pp))
```

```
Rule :
→ (SENT:* tree NP-ATS PP-MOD (COORD CC tree))
← (SENT:* SENT:*/np (:np (:np NP-ATS:np
       (:np\\np PP-MOD:np\\np ("COORD:(np\\np)\\(np\\np)" "CC:((np\\np)\\(np\\np))/(np\\np)" PP:np\\np)))))
```

```
Rule :
→ (SENT:* VN VPpart)
← (SENT:* "VN:*/(np\\s_p)" VPpart:np\\s_p)
```

```
Rule :
→ (SENT:* VPP ADV AP)
← (SENT:* (":*/(n\\n)" "VPP:*/(n\\n)" "ADV:(*/(n\\n))\\(*/(n\\n))") AP:n\\n)
```

Rule : use-ponct
```
→ (SENT:* tree (COORD CC Sint PONCT Sint))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
       (:* Sint:* (:*\\* "PONCT:(*\\*)/*" Sint:*))))
```
Rule :
```
→ (SENT:* (COORD CC ADV NP) tree)
← (SENT:* (COORD:np CC:np/np (:np ADV:np/np NP:np))
     SENT:np\\*)
```
Rule :
```
→ (SENT:* VPpart VN)
← (SENT:* VPpart:np\\s_p "VN:(np\\s_p)\\*")
```
Rule :
```
→ (SENT:* VN PRO)
← (SENT:* VN:*/np PRO:np)
```
Rule : use-ponct
```
→ (SENT:* VN PONCT VN)
← (SENT:* VN:* (:*\\* "PONCT:(*\\*)/*" VN:*))
```
Rule : use-ponct
```
→ (SENT:* VN PONCT VN PONCT VN)
← (SENT:* VN:* (:*\\* "PONCT:(*\\*)/*"
       (:* VN:* (:*\\* "PONCT:(*\\*)/*" VN:*))))
```
Rule :
```
→ (PP:* tree (COORD CC AdP PP))
← (PP:* PP:*
     (COORD:*\\* "CC:(*\\*)/*" (:* AdP:*/* PP:*)))
```
Rule :
```
→ (PP:* P PRO PP)
← (PP:* P:*/np (:np PRO:np PP:np\\np))
```
Rule :
```
→ (NP:* tree (COORD CC PP NP))
← (NP:* (:* NP:*
       (COORD:*\\* "CC:(*\\*)/np" (:np PP:np/np NP:np))))
```
Rule :
```
→ (NP:* NC PP PP)
← (NP:np (:n NC:n PP:n\\n) PP:n\\*)
```
Rule : use-ponct
```
→ (NP:* NPP PONCT NP PONCT)
← (NP:* NPP:np
     (:np\\* "PONCT:(np\\*)/n" (:n NP:n PONCT:n\\n)))
```
Rule : use-ponct
```
→ (NP:* NPP PONCT NPP PONCT)
← (NP:* NPP:np (:np\\*
       (":(np\\*)/pf" "PONCT:((np\\*)/pf)/np" NPP:np)
       PONCT:pf))
```
Rule :
```
→ (NP:* NPP (COORD CC NP))
← (NP:* NPP:n (COORD:n\\* "CC:(n\\*)/np" NP:np))
```
Rule :
```
→ (NP:* tree AP (COORD CC ADV))
← (NP:* NP:n (:n\\* AP:n\\n ("COORD:(n\\n)\\(n\\*)" "CC:((n\\n)\\(n\\*))/(n\\n)" ADV:n\\n)))
```
Rule :
```
→ (NP:* NC NC)
← (NP:* NC:n NC:n\\*)
```
Rule : use-ponct
```
→ (NP:* tree (PONCT -LBR-) ADV PP (PONCT -RBR-))
← (NP:* NP:np (:np\\*
       (":(np\\*)/pf" ("PONCT:((np\\*)/pf)/(n\\n)" "-LBR-:((np\\*)/pf)/(n\\n)") (:n\\n "ADV:(n\\n)/(n\\n)" PP:n\\n))
       (PONCT:pf -RBR-:pf)))
```
Rule :
```
→ (NP-SUJ:* NC ADJ NC)
← (NP-SUJ:* NC:*/n (:n ADJ:n/n NC:n))
```
Rule : use-ponct
```
→ (NP-SUJ:* NC NPP NPP PONCT NP PONCT)
← (NP-SUJ:* (:np NC:np/np (:np NPP:np/np NPP:np)) (:np\\*
       (":(np\\*)/pf" "PONCT:((np\\*)/pf)/np" NP:np)
       PONCT:pf))
```
Rule :
```
→ (NP-OBJ:* NPP NC)
← (NP-OBJ:* NPP:*/n NC:n)
```
Rule :
```
→ (VN:* tree CLO-DE_OBJ)
← (VN:* VN:*/pp CLO-DE_OBJ:pp)
```
Rule :
```
→ (VN:* tree ADV PP)
← (VN:* "VN:*/(n\\n)"
     (:n\\n "ADV:(n\\n)/(n\\n)" PP:n\\n))
```
Rule :
```
→ (Srel:* PONCT tree)
← (Srel:* PONCT:*/* SENT:*)
```
Rule :
```
→ (Srel:* NP NP tree)
← (Srel:* NP:*/np (:np NP:np AP:np\\np))
```
Rule :
```
→ (VPpart-MOD:* P VN NP-MOD)
← (VPpart-MOD:* "P:*/(np\\s_p)" (:np\\s_p VN:np\\s_p "NP-MOD:(np\\s_p)\\(np\\s_p)"))
```
Rule :
```
→ (VPpart-ATS:* ADV VPP)
← (VPpart-ATS:* ADV:*/* VPP:*)
```
Rule :
```
→ (VPpart:* tree AdP-MOD)
← (VPpart:* VPpart:* AdP-MOD:*\\*)
```
Rule :
```
→ (VPinf-OBJ:* VN ADV NP-MOD)
← (VPinf-OBJ:* (:* VN:* ADV:*\\*) NP-MOD:*\\*)
```
Rule :
```
→ (VPinf-OBJ:* VN PP-MOD NP-MOD)
← (VPinf-OBJ:* (:* VN:* PP-MOD:*\\*) NP-MOD:*\\*)
```
Rule :
```
→ (VPinf-OBJ:* VN AP-MOD)
← (VPinf-OBJ:* VN:* AP-MOD:*\\*)
```
Rule :
```
→ (VPinf-ATS:* P VN NP-OBJ PP-MOD)
← (VPinf-ATS:* "P:*/(np\\s_i)" (VN:np\\s_i (VN:np\\s_i "VN:(np\\s_i)/np" NP-OBJ:np) "PP-MOD:(np\\s_i)\\(np\\s_i)"))
```

```
Rule :
→ (VPinf-A_OBJ:* P VN ADV)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)"
     (:np\\s_i VN:np\\s_i "ADV:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf-DE_OBJ:* P VN ADV)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)"
     (:np\\s_i VN:np\\s_i "ADV:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf-MOD:* tree VPpart-MOD)
← (VPinf-MOD:* VPinf:* VPpart-MOD:*\\*)
Rule :
→ (VPinf:* NP-MOD tree)
← (VPinf:* NP-MOD:*/* VPinf:*)
Rule :
→ (VPinf:* tree Ssub-DE_OBJ)
← (VPinf:* VPinf:*/cs Ssub-DE_OBJ:cs)
Rule :
→ (VPinf-OBJ:* tree Sint-MOD)
← (VPinf-OBJ:* SENT:* Sint-MOD:*\\*)
```

```
Rule :
→ (VPinf-ATS:* ADV tree)
← (VPinf-ATS:* ADV:*/* SENT:*)
Rule :
→ (AP-MOD:* ADJ VPinf)
← (AP-MOD:* ADJ:* VPinf:*\\*)
Rule :
→ (AP-ATS:* tree ADJ)
← (AP-ATS:* "AP:*/(n\\n)" ADJ:n\\n)
Rule :
→ (AP-ATO:* ADV ADJ PP)
← (AP-ATO:* ADV:*/* (:* ADJ:*/pp PP:pp))
Rule :
→ (Ssub:* CS tree (COORD CC (Ssub CS tree)))
← (Ssub:* (:* CS:*/s SENT:s)
     (COORD:*\\* "CC:(*\\*)/*" (Ssub:* CS:*/s SENT:s)))
Rule :
→ (Ssub-MOD:* P CS tree)
← (Ssub-MOD:* P:*/* (:* CS:*/s SENT:s))
Rule :
→ (COORD:pp\\pp CC tree)
← (COORD:pp\\pp "CC:(pp\\pp)/pp" PP:pp)
```

```
Rule :
→ ("COORD:(n\\n)\\(n\\n)" CC PP)
← ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/(n\\n)" PP:n\\n)
```

```
Rule :
→ (COORD:* CC Sint)
← (COORD:* CC:*/s Sint:s)

Rule :
→ (PP-SUJ:* P+D tree)
← (PP-SUJ:* P+D:*/n NP:n)

Rule :
→ (SENT:* NP-P_OBJ VN)
← (SENT:* NP-P_OBJ:np VN:np\\*)
```

```
Rule : use-ponct
→ (SENT ADV PONCT VN AP-ATS VPinf-OBJ PONCT)
← (TEXT:s (SENT:s ADV:s/s (:s PONCT:s/s (:s
          (":s/(np\\s_i)" "VN:(s/(np\\s_i))/(n\\n)"
            AP-ATS:n\\n)
          VPinf-OBJ:np\\s_i)))
    PONCT:s\\s)
Rule :
→ (SENT:* ADVWH VPinf)
← (SENT:* "ADVWH:*/(np\\s_i)" VPinf:np\\s_i)
```

```
Rule :
→ (SENT:* (VN tree VPP) NP-OBJ VPinf-DE_OBJ)
← (SENT:* (VN:* "VN:*/(np\\s_p)"
      (VPP:np\\s_p ("VPP:(np\\s_p)/(np\\s_i)" "VPP:((np\\s_p)/(np\\s_i))/np" NP-OBJ:np) VPinf-DE_OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP VPP) PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
        (VPP:np\\s_p "VPP:(np\\s_p)/pp" PP-OBJ:pp))))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "PP-MOD:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
        VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "NP-MOD:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
        VPinf-OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/cs" "VPP:(np\\s_p)/cs" "ADV:((np\\s_p)/cs)\\((np\\s_p)/cs)") Ssub-OBJ:cs)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "ADV:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
        VPinf-OBJ:np\\s_i)))
```

```
Rule :
→ (SENT tree VPinf-MOD)
← (VPinf-OBJ:np\\s_i SENT:s VPinf-MOD:s\\s)
```

```
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "NP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_de" "VN:(np\\s)/pp_de" "NP-MOD:((np\\s)/pp_de)\\((np\\*)/pp_de)") PP-DE_OBJ:pp_de))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD PP-A_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "AdP-MOD:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
Rule :
→ (SENT:* NP-SUJ VN ADV NP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
        "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
      NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN ADV ADV Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
        "ADV:((np\\*)/cs)\\((np\\*)/cs)")
      Ssub-OBJ:cs))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "PP-MOD:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "PP-MOD:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "AdP-MOD:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
        "ADV:((np\\*)/np)\\((np\\*)/np)")
      NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN ADV Ssub-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)") Ssub-ATS:cs))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD AP-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "AdP-MOD:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD AP-ATS)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "NP-MOD:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV AP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/(n\\n)"
        (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "ADV:((np\\*)/(n\\n))\\((np\\*)/(n\\n))")
        "ADV:((np\\*)/(n\\n))\\((np\\*)/(n\\n))")
      AP-ATS:n\\n))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD VPpart-ATS)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "PP-MOD:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
      VPpart-ATS:np\\s_p))
```

```
Rule :
→ (SENT:* NP-SUJ VN NP-OBJ PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/pp" "VN:((np\\*)/pp)/np" NP-OBJ:np)
      PP-OBJ:pp))
```

```
Rule :
→ (SENT:* NP-SUJ VN VPinf-OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np"
        "VN:((np\\*)/np)/(np\\s_i)" VPinf-OBJ:np\\s_i)
      NP-OBJ:np))
```

```
Rule :
→ (SENT:* NP-SUJ VN PP-A_OBJ Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs"
        "VN:((np\\*)/cs)/pp_a" PP-A_OBJ:pp_a)
      Ssub-OBJ:cs))
```

```
Rule :
→ (SENT:* VPinf-SUJ VN AP-ATS)
← (SENT:* VPinf-SUJ:np\\s_i (":(np\\s_i)\\*"
      "VN:((np\\s_i)\\*)/(n\\n)" AP-ATS:n\\n))
```

```
Rule :
→ (SENT:* NP-SUJ NP-MOD VN PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "NP-MOD:((np\\*)/pp)/((np\\*)/pp)" "VN:(np\\*)/pp") PP-P_OBJ:pp))
Rule : use-ponct
→ (SENT:* VN VPinf-OBJ PONCT VPinf-OBJ)
← (SENT:* "VN:*/(np\\s_i)"
    (:np\\s_i VPinf-OBJ:np\\s_i (":(np\\s_i)\\(np\\s_i)" "PONCT:((np\\s_i)\\(np\\s_i))/(np\\s_i)" VPinf-OBJ:np\\s_i)))
```

```
Rule : use-ponct
→ (SENT:* tree (NC-MOD PONCT NC PONCT))
← (SENT:* SENT:* (NC-MOD:*\\*
      (":(*\\*)/pf" "PONCT:((*\\*)/pf)/n" NC:n)
      PONCT:pf))
Rule :
→ (SENT:* tree NP-OBJ (COORD CC PP NP))
← (SENT:* SENT:*/np (:np NP-OBJ:np (COORD:np\\np
       "CC:(np\\np)/np" (:np PP:np/np NP:np))))
Rule :
→ (SENT:* tree NP-ATS (COORD CC NP))
← (SENT:* SENT:*/np (:np NP-ATS:np
      (COORD:np\\np "CC:(np\\np)/np" NP:np)))
Rule :
→ (SENT:* (CLO en) VN)
← (SENT:* (CLO:*/* en:*/*) VN:*)
```

```
Rule :
→ (SENT:* I NP-SUJ VN)
← (SENT:* I:*/* (:* NP-SUJ:np VN:np\\*))
Rule :
→ (SENT:* I VN)
← (SENT:* I:*/* VN:*)
Rule :
→ (SENT:* ADJ NP-SUJ tree)
← (SENT:* (:np ADJ:np/np NP-SUJ:np) SENT:np\\*)
Rule :
→ (SENT:* NP ADV VPinf)
← (SENT:* NP:np
    (:np\\* "ADV:(np\\*)/(np\\*)" VPinf:np\\*))
Rule :
→ (SENT:* NP AP)
← (SENT:* NP:np AP:np\\*)
```

```
Rule :
→ (SENT:* tree (COORD CC CC VPinf))
← (SENT:* SENT:*
    (COORD:*\\* (":(*\\*)/(np\\s_i)" "CC:((*\\*)/(np\\s_i))/((*\\*)/(np\\s_i))" "CC:(*\\*)/(np\\s_i)") VPinf:np\\s_i))
Rule :
→ (SENT:* VN ADV VN)
← (SENT:* (":*/(np\\s)" "VN:*/(np\\s)" "ADV:(*/(np\\s))\\(*/(np\\s))") VN:np\\s)
```

```
Rule :
→ (SENT:* (COORD CC NP) (COORD CC NP) VN)
← (SENT:* CC:*/* (:* (COORD:np NP:np
       (COORD:np\\np "CC:(np\\np)/np" NP:np))
      VN:np\\*))
```

```
Rule :
→ (SENT:* (COORD CC ADV) tree)
← (SENT:* (COORD:*/* "CC:(*/*)/(*/*)" ADV:*/*) SENT:*)
Rule :
→ (SENT:* tree NP-OBJ PP)
← (SENT:* SENT:*/np (:np NP-OBJ:np PP:np\\np))
```

```
Rule :
→ (SENT:* tree VPinf PONCT VPinf)
← (SENT:* "SENT:*/(np\\s_i)"
    (:np\\s_i VPinf:np\\s_i (":(np\\s_i)\\(np\\s_i)" "PONCT:((np\\s_i)\\(np\\s_i))/(np\\s_i)" VPinf:np\\s_i)))
```

```
Rule :
→ (SENT:* (COORD CC Ssub) tree)
← (SENT:* (COORD:*/* "CC:(*/*)/cs" Ssub:cs) SENT:*)

Rule :
→ (SENT:* tree PP-SUJ)
← (SENT:* SENT:*/pp PP-SUJ:pp)

Rule :
→ (SENT:* VN NP-OBJ VPpart)
← (SENT:*
    (":*/(np\\s_p)" "VN:(*/(np\\s_p))/np" NP-OBJ:np)
    VPpart:np\\s_p)

Rule :
→ (SENT:* VPP VPpart)
← (SENT:* "VPP:*/(np\\s_p)" VPpart:np\\s_p)

Rule :
→ (SENT:* PP VPpart)
← (SENT:* PP:pp VPpart:pp\\*)
```

```
Rule :
→ (SENT:* VPR VPP)
← (SENT:* "VPR:*/(np\\s_p)" VPP:np\\s_p)

Rule : use-ponct
→ (SENT:* tree (COORD CC PONCT ADV PONCT Sint))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* PONCT:*/* (:* ADV:*/* (:* PONCT:*/* Sint:*)))))

Rule :
→ (SENT:* tree (COORD CC tree NP-OBJ NP-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/*"
      (:* (:* SENT:*/np NP-OBJ:np) NP-MOD:*\\*)))

Rule :
→ (SENT:* tree (COORD CC CC tree PP-DE_OBJ))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/*"
        "CC:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      (:* SENT:*/pp_de PP-DE_OBJ:pp_de)))
```

Rule :
→ (SENT:* tree (COORD CC VN AdP-MOD))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* VN:* AdP-MOD:**)))

Rule :
→ (SENT:* tree (COORD CC tree PP-OBJ))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (:* SENT:*/pp PP-OBJ:pp)))

Rule :
→ (SENT:* VN ADV AP-OBJ)
← (SENT:* (":*/(n\\n)" "VN:*/(n\\n)" "ADV:(*/(n\\n))\\(*/(n\\n))") AP-OBJ:n\\n)

Rule :
→ (PP-DE_OBJ:* P AP)
← (PP-DE_OBJ:* "P:*/(n\\n)" AP:n\\n)

Rule :
→ (PP:* P ADV Ssub)
← (PP:* (:*/cs P:*/cs "ADV:(*/cs)\\(*/cs)") Ssub:cs)

Rule :
→ (PP:* P ADVWH)
← (PP:* "P:*/(s/s)" ADVWH:s/s)

Rule :
→ (PP:* tree Sint)
← (PP:* PP:*/s Sint:s)

Rule :
→ (PP:* P+D tree VPpart)
← (PP:* P+D:*/n (:n NP:n VPpart:n\\n))

Rule : use-ponct
→ (PP:* tree PONCT COORD PONCT)
← (PP:* PP:* (:** (":(**)/pf" "PONCT:((**)/pf)/(pp\\pp)" COORD:pp\\pp) PONCT:pf))

Rule :
→ (PP:* P NPP)
← (PP:* P:*/np NPP:np)

Rule :
→ (PP:* P ADJ (COORD CC tree) NP)
← (PP:* P:*/np (:np (:np/np ADJ:np/np ("COORD:(np/np)\\(np/np)" "CC:((np/np)\\(np/np))/(np/np)" AP:np/np)) NP:np))

Rule :
→ (NP:* ADVWH P)
← (NP:* ADVWH:*/* P:*)

Rule : use-ponct
→ (NP:* ADJ PONCT)
← (NP:* ADJ:* PONCT:**)

Rule :
→ (NP:* tree (COORD CC NC))
← (NP:* NP:* (COORD:** "CC:(**)/n" NC:n))

Rule :
→ (NP:* tree (COORD CC NPP))
← (NP:* NP:* (COORD:** "CC:(**)/np" NPP:np))

Rule :
→ (NP:* PRO Srel)
← (NP:* PRO:np Srel:np*)

Rule :
→ (NP:* NC ADV)
← (NP:* NC:n ADV:n*)

Rule : use-ponct
→ (NP-SUJ:* NPP PONCT NP PONCT)
← (NP-SUJ:* NPP:np
(:np* "PONCT:(np*)/n" (:n NP:n PONCT:n\\n)))

Rule :
→ (NP-MOD:* tree VPinf)
← (PONCT:n\\n NP:n VPinf:n*)

Rule :
→ (NP-MOD:* tree Ssub)
← (NP-MOD:* NP:n Ssub:n*)

Rule :
→ (NP-ATS:* PREF tree)
← (NP-ATS:* PREF:*/n NP:n)

Rule :
→ (NP-ATS:* tree ADV)
← (NP-ATS:* NP:n ADV:n*)

Rule :
→ (NP-ATS:* tree (NP-MOD PONCT NP PONCT))
← (NP-ATS:* NP:* (NP-MOD:**
(":(**)/pf" "PONCT:((**)/pf)/np" NP:np)
PONCT:pf))

Rule :
→ (NP-ATO:* DET NC)
← (NP-ATO:* DET:*/n NC:n)

Rule :
→ (VN:* CLR-MOD tree)
← (VN:* CLR-MOD:cl_r VN:cl_r*)

Rule :
→ (VN:* CLR-P_OBJ tree)
← (VN:* CLR-P_OBJ:cl_r VN:cl_r*)

Rule :
→ (VN:* V (NP PRO) VPP)
← (VN:* "V:*/(np\\s_p)" (:np\\s_p "PRO:(np\\s_p)/((np\\s_p)/np)" "VPP:(np\\s_p)/np"))

Rule :
→ (VN:* CLS-A_OBJ tree)
← (VN:* CLS-A_OBJ:np VN:np*)

Rule :
→ (VPpart-ATO:* VPP PP)
← (VPpart-ATO:* VPP:*/pp PP:pp)

```
Rule : use-ponct
→ (VPpart:* VPP PONCT AP PONCT)
← (VPpart:* "VPP:*/(n\\n)" (:n\\n
       (":(n\\n)/gf" "PONCT:((n\\n)/gf)/(n\\n)" AP:n\\n)
       PONCT:gf))
Rule :
→ (VPpart-MOD:* tree Sint-MOD)
← (VPpart-MOD:* VPpart:* Sint-MOD:*\\*)
Rule :
→ (VPpart-MOD:* tree VPpart-MOD)
← (VPpart-MOD:* VPpart:* VPpart-MOD:*\\*)
Rule :
→ (VPpart-MOD:* tree AP-MOD)
← (VPpart-MOD:* VPpart:* AP-MOD:*\\*)
```

```
Rule :
→ (VPpart:* tree Ssub-MOD)
← (VPpart:* VPpart:* Ssub-MOD:*\\*)
Rule :
→ (VPinf-OBJ:* VN NP-MOD PP-MOD)
← (VPinf-OBJ:* (:* VN:* NP-MOD:*\\*) PP-MOD:*\\*)
Rule :
→ (VPinf-OBJ:* CLR VINF)
← (VPinf-OBJ:* CLR:np VINF:np\\*)
Rule :
→ (VPinf-ATS:* VN PP-DE_OBJ)
← (VPinf-ATS:* VN:*/pp_de PP-DE_OBJ:pp_de)
Rule :
→ (VPinf:* VN AP-MOD)
← (VPinf:* VN:* AP-MOD:*\\*)
```

```
Rule :
→ (VPinf-OBJ:* P VN NP-MOD)
← (VPinf-OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "NP-MOD:(np\\s_i)\\(np\\s_i)"))
```

```
Rule :
→ (VPinf-MOD:* tree PP-A_OBJ)
← (VPinf-MOD:* VPinf:*/pp_a PP-A_OBJ:pp_a)
Rule :
→ (VPinf-MOD:* tree VPinf-OBJ)
← (VPinf-MOD:* "VPinf:*/(np\\s_i)" VPinf-OBJ:np\\s_i)
Rule :
→ (VPinf:* tree PP-ATO)
← (VPinf:* VPinf:*/pp PP-ATO:pp)
Rule :
→ (VPinf-OBJ:* AdP-MOD tree)
← (VPinf-OBJ:* AdP-MOD:*/* SENT:*)
Rule :
→ (VPinf-ATS:* tree Ssub-MOD)
← (VPinf-ATS:* VPinf:* Ssub-MOD:*\\*)
Rule :
→ (AP-MOD:* ADJ Ssub)
← (AP-MOD:* ADJ:*/cs Ssub:cs)
Rule :
→ (AP-MOD:* tree (COORD CC AP))
← (AP-MOD:* AP:* (COORD:*\\* "CC:(*\\*)/*" AP:*))
Rule :
→ (AP-MOD:* tree ADV)
← (AP-MOD:* AP:* ADV:*\\*)
Rule :
→ (AP-ATS:* tree NP)
← (AP-ATS:* AP:*/np NP:np)
Rule :
→ (AP:* tree PP-MOD)
← (AP:* AP:* PP-MOD:*\\*)
Rule :
→ (AP-ATO:* tree (COORD CC AP))
← (AP-ATO:* AP:* (COORD:*\\* "CC:(*\\*)/*" AP:*))
Rule :
→ (Ssub:* ADVWH tree)
← (Ssub:* ADVWH:*/s SENT:s)
Rule :
→ (Ssub-OBJ:* ADV tree)
← (Ssub-OBJ:* ADV:*/s SENT:s)
```

```
Rule : use-ponct
→ (Ssub-MOD:* PONCT Ssub-MOD PONCT)
← (Ssub-MOD:* (:*/pf "PONCT:(*/pf)/*" Ssub-MOD:*)
     PONCT:pf)
Rule :
→ (AdP-MOD:* ADV ADV ADV Ssub)
← (AdP-MOD:* ADV:*/* (:* ADV:*/* (:* ADV:* Ssub:*\\*)))
Rule :
→ (AdP-ATS:* ADV Ssub)
← (Ssub:*\\* ADV:*/cs Ssub:cs)
Rule : use-ponct
→ (AdP-MOD:* PONCT AdP-MOD PONCT)
← (AdP-MOD:* (:*/pf "PONCT:(*/pf)/*" AdP-MOD:*) PONCT:pf)
Rule :
→ (AdP-ATS:* tree)
← (AdP-ATS:* AdP:*)
Rule :
→ (COORD-MOD:* CC NP)
← (COORD-MOD:* CC:*/np NP:np)
Rule :
→ (PONCT-MOD:* PONCT PONCT PONCT)
← (PONCT-MOD:* (:*/pf "PONCT:(*/pf)/*" PONCT:*) PONCT:pf)
Rule :
→ (COORD:* CC VPpart)
← (COORD:* "CC:*/(np\\s_p)" VPpart:np\\s_p)
Rule :
→ (SENT:* VPinf PP)
← (SENT:* VPinf:*/pp PP:pp)
Rule :
→ (COORD:* CC VN NP-ATS Srel-MOD)
← (COORD:* CC:*/s
     (:s (:s VN:s/np NP-ATS:np) Srel-MOD:s\\s))
Rule : use-ponct
→ (NC-MOD:* PONCT NC PONCT)
← (NC-MOD:* (:*/pf "PONCT:(*/pf)/n" NC:n) PONCT:pf)
Rule : use-ponct
→ (SENT (PP P NP) PONCT (PP P NP) PONCT)
← (SENT:s (:pp (PP:pp P:pp/np NP:np) (:pp\\pp
         "PONCT:(pp\\pp)/pp" (PP:pp P:pp/np NP:np)))
     PONCT:pp\\s)
```

```
Rule :
→ (SENT:* (VN CLS-SUJ V NP-MOD VPP) NP-OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "V:(np\\*)/(np\\s_p)"
       (:np\\s_p "NP-MOD:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p "VPP:(np\\s_p)/np" NP-OBJ:np))))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP ADV VPP) PP-P_OBJ)
← (SENT:* CLS-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
           "ADV:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p
             "VPP:(np\\s_p)/pp" PP-P_OBJ:pp)))))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree VPP) PP-DE_OBJ NP-OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p ("VPP:(np\\s_p)/np"
           "VPP:((np\\s_p)/np)/pp_de" PP-DE_OBJ:pp_de)
         NP-OBJ:np)))
```

```
Rule :
→ (SENT:* NP-SUJ (VN V NP-MOD VPP VPP))
← (SENT:* NP-SUJ:np (VN:np\\* "V:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p "NP-MOD:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" VPP:np\\s_p))))
Rule :
→ (SENT:* NP-SUJ (VN V NP-MOD VPP))
← (SENT:* NP-SUJ:np (VN:np\\* "V:(np\\*)/(np\\s_p)" (VPP:np\\s_p "NP-MOD:(np\\s_p)/(np\\s_p)" VPP:np\\s_p)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p (":(np\\s_p)/cs" "VPP:(np\\s_p)/cs" "NP-MOD:((np\\s_p)/cs)\\((np\\s_p)/cs)") Ssub-OBJ:cs)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
         (":(np\\s_p)/pp_de" "VPP:(np\\s_p)/pp_de" "NP-MOD:((np\\s_p)/pp_de)\\((np\\s_p)/pp_de)") PP-DE_OBJ:pp_de)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD VPinf-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
         ("VPP:(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "PP-MOD:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
         VPinf-DE_OBJ:np\\s_i)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
       (VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "ADV:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-ATS:pp)))
```

```
Rule :
→ (SENT tree Ssub-MOD)
← (PP-ATS:pp SENT:s Ssub-MOD:s\\s)

Rule :
→ (SENT tree COORD-MOD)
← (SENT:s SENT:s COORD-MOD:s\\s)

Rule :
→ (SENT tree PONCT-MOD)
← (TEXT:txt SENT:s PONCT-MOD:s\\txt)
```

```
Rule : use-ponct
→ (SENT:* tree (PONCT-MOD PONCT PONCT PONCT))
← (SENT:* SENT:* (PONCT-MOD:*\\* (":(*\\*)/pf"
         "PONCT:((*\\*)/pf)/(*\\*)" PONCT:*\\*)
       PONCT:pf))
Rule :
→ (SENT:* NP VN NP-ATS)
← (SENT:* NP:np (:np\\* "VN:(np\\*)/np" NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-ATO)
← (SENT:* CLS-SUJ:np (:np\\* "VN:(np\\*)/np" NP-ATO:np))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) VPpart-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "VPpart-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np
     (:np\\* (":(np\\*)/pp_de" "VN:(np\\s)/pp_de" "AdP-MOD:((np\\s)/pp_de)\\((np\\*)/pp_de)") PP-DE_OBJ:pp_de))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "AdP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) AdP-MOD PP-P_OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "AdP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-P_OBJ:pp))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "AdP-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) AdP-MOD VPinf-OBJ)
← (SENT:* CLS-SUJ:np (:np\\*
       (":(np\\*)/(np\\s_i)" "VN:(np\\s)/(np\\s_i)" "AdP-MOD:((np\\s)/(np\\s_i))\\((np\\*)/(np\\s_i))")
       VPinf-OBJ:np\\s_i))
```

```
Rule :
→ (SENT:* NP-SUJ VN ADV ADV PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" (":(np\\*)/pp" "VN:(np\\*)/pp" "ADV:((np\\*)/pp)\\((np\\*)/pp)")
        "ADV:((np\\*)/pp)\\((np\\*)/pp)")
      PP-OBJ:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV VPinf-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/(np\\s_i)"
        (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
        "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) AdP-MOD NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "AdP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "NP-MOD:((np\\s)/np)\\((np\\*)/np)") NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN ADV PP-MOD NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
        "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
      NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV Ssub-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
        "ADV:((np\\*)/cs)\\((np\\*)/cs)")
      Ssub-ATS:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD AP-ATS)
← (SENT:* CLS-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "PP-MOD:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "PP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-ATS:pp))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV Srel)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/s" "VN:(np\\*)/s" "ADV:((np\\*)/s)\\((np\\*)/s)") Srel:s))

Rule :
→ (SENT:* NP-SUJ VN PP-DE_OBJ Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs"
        "VN:((np\\*)/cs)/pp_de" PP-DE_OBJ:pp_de)
      Ssub-OBJ:cs))

Rule :
→ (SENT:* NP-SUJ VN PP-A_OBJ PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp"
        "VN:((np\\*)/pp)/pp_a" PP-A_OBJ:pp_a)
      PP-P_OBJ:pp))

Rule :
→ (SENT:* NP-SUJ VN ADV NP)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "ADV:((np\\s)/np)\\((np\\*)/np)") NP:np))

Rule :
→ (SENT:* NP-SUJ VN PP)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(n\\n)" PP:n\\n))
Rule :
→ (SENT NP-MOD PP)
← (SENT:np NP-MOD:np PP:np\\np)

Rule :
→ (SENT NP VPinf)
← (SENT:np NP:np VPinf:np\\np)
Rule :
→ (SENT NP-SUJ VN)
← (SENT:s NP-SUJ:np VN:np\\s)

Rule :
→ (SENT:* NP-SUJ ADV VN ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "ADV:(np\\*)/(np\\*)"
      (:np\\* (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") NP-OBJ:np)))
Rule : use-ponct
→ (SENT PONCT NP-MOD)
← (SENT:np PONCT:np/np NP-MOD:np)
Rule :
→ (SENT:* tree AP-A_OBJ)
← (SENT:* "SENT:*/(n\\n)" AP-A_OBJ:n\\n)

Rule :
→ (SENT:* tree (COORD-DE_OBJ CC PP))
← (SENT:* SENT:*/pp_de
    (COORD-DE_OBJ:pp_de CC:pp_de/pp_de PP:pp_de))
```

Rule : use-ponct
→ (SENT:* tree (NP-ATS-MOD PONCT NP-ATS PONCT))
← (SENT:* SENT:* (NP-ATS-MOD:**
(":(**)/pf" "PONCT:((**)/pf)/np" NP-ATS:np)
PONCT:pf))

Rule :
→ (SENT:* VPinf (COORD CC VPinf))
← (SENT:* VPinf:np\\s_i ("COORD:(np\\s_i)*" "CC:((np\\s_i)*)/(np\\s_i)" VPinf:np\\s_i))

Rule :
→ (SENT:* NC-MOD tree)
← (SENT:* NC-MOD:*/* SENT:*)

Rule :
→ (SENT:* PP (COORD CC Ssub))
← (SENT:* PP:n\\n
("COORD:(n\\n)*" "CC:((n\\n)*)/cs" Ssub:cs))

Rule :
→ (SENT:* VN NP-OBJ (Srel tree))
← (SENT:* (:*/s "VN:(*/s)/np" NP-OBJ:np) (Srel:s SENT:s))

Rule :
→ (SENT:* tree (COORD CC AdP VPinf))
← (SENT:* SENT:* (COORD:** "CC:(**)/(np\\s_i)"
(:np\\s_i "AdP:(np\\s_i)/(np\\s_i)"
VPinf:np\\s_i)))

Rule :
→ (SENT:* VN ADV I)
← (SENT:* (:* VN:* ADV:**) I:**)

Rule :
→ (SENT:* VINF NP VPpart)
← (SENT:* VINF:*/s_p (:s_p NP:np VPpart:np\\s_p))

Rule :
→ (SENT:* tree VPinf-SUJ)
← (SENT:* "SENT:*/(np\\s_i)" VPinf-SUJ:np\\s_i)

Rule :
→ (SENT:* CLO tree VPP)
← (SENT:* (:* "CLO:*/(*/np)" (:*/np
"VN:(*/np)/((np\\s_p)/np)" "VPP:(np\\s_p)/np")))

Rule :
→ (SENT:* NP ADV PP)
← (SENT:* (:np NP:np ADV:np\\np) PP:np*)

Rule :
→ (SENT:* NP NP-MOD PP)
← (SENT:* (:np NP:np NP-MOD:np\\np) PP:np*)

Rule :
→ (SENT:* NP VPinf tree PP)
← (SENT:* NP:np
(:np* "VPinf:(np*)/pp" (:pp PP:pp PP:pp\\pp)))

Rule :
→ (SENT:* VPP PP VPpart)
← (SENT:* (":*/(np\\s_p)" "VPP:(*/(np\\s_p))/pp" PP:pp)
VPpart:np\\s_p)

Rule :
→ (SENT:* VPP ADV VPpart)
← (SENT:* (":*/(np\\s_p)" "VPP:*/(np\\s_p)" "ADV:(*/(np\\s_p))\\(*/(np\\s_p))") VPpart:np\\s_p)

Rule :
→ (SENT:* PP NP VPpart)
← (SENT:* PP:*/s_p (:s_p NP:np VPpart:np\\s_p))

Rule :
→ (SENT:* VPR PP PP)
← (SENT:* (:*/pp "VPR:(*/pp)/pp" PP:pp) PP:pp)

Rule :
→ (SENT:* tree (COORD (Sint CC tree)))
← (SENT:* SENT:*
(COORD:** "CC:(**)/*" (Sint:* Sint:*)))

Rule : use-ponct
→ (SENT:* tree (COORD CC tree Ssub-MOD PONCT VPinf-MOD))
← (SENT:* SENT:* (COORD:** "CC:(**)/*" (:*
(:* (:* SENT:* Ssub-MOD:**) PONCT:**)
VPinf-MOD:**)))

Rule :
→ (SENT:* VN Ssub-P_OBJ)
← (VPinf-MOD:** VN:*/cs Ssub-P_OBJ:cs)

Rule :
→ (SENT:* (COORD CC ADJ) tree)
← (SENT:* (COORD:n\\n "CC:(n\\n)/(n\\n)" ADJ:n\\n)
"SENT:(n\\n)*")

Rule :
→ (SENT:* (COORD CC VPpart) tree)
← (SENT:* (COORD:np\\s_p "CC:(np\\s_p)/(np\\s_p)"
VPpart:np\\s_p)
"SENT:(np\\s_p)*")

Rule :
→ (SENT:* tree (COORD CC CC tree PP-MOD))
← (SENT:* SENT:* (COORD:** (":(**)/*"
"CC:((**)/*)/((**)/*)" "CC:(**)/*")
(:* SENT:* PP-MOD:**)))

Rule :
→ (SENT:* tree (COORD CC CC tree AP-ATS))
← (SENT:* SENT:* (COORD:** (":(**)/*"
"CC:((**)/*)/((**)/*)" "CC:(**)/*")
(:* "SENT:*/(n\\n)" AP-ATS:n\\n)))

Rule :
→ (SENT:* VPP ET)
← (AP-ATS:n\\n VPP:*/np ET:np)

Rule :
→ (SENT:* NP AdP)
← (SENT:* NP:np AdP:np*)

Rule :
→ (SENT:* PROREL VN)
← (SENT:* PROREL:np VN:np*)

Rule :
→ (SENT:* VN P)
← (SENT:* "VN:*/(pp/np)" P:pp/np)

Rule :
→ (SENT:* VN (COORD CC VPP))
← (SENT:* VN:np\\s ("COORD:(np\\s)*"
"CC:((np\\s)*)/(np\\s)" VPP:np\\s))

Rule :
→ (SENT:* VN (COORD CC CC NP))
← (SENT:* VN:*/np (COORD:np
(:np/np "CC:(np/np)/(np/np)" CC:np/np) NP:np))

Rule :
→ (SENT:* VN ADV VPP)
← (SENT:* (":*/(np\\s_p)" "VN:*/(np\\s_p)" "ADV:(*/(np\\s_p))\\(*/(np\\s_p))") VPP:np\\s_p)

```
Rule : use-ponct
→ (PP:* P PONCT (NP NPP PONCT))
← (PP:* P:*/n
     (:n (:n/gf "PONCT:(n/gf)/n" (NP:n NPP:n)) PONCT:gf))

Rule :
→ (PP-MOD:* P PROWH)
← (PP-MOD:* P:*/np PROWH:np)

Rule :
→ (PP-P_OBJ:* P AP)
← (PP-P_OBJ:* "P:*/(n\\n)" AP:n\\n)

Rule :
→ (PP-DE_OBJ:* P ADV)
← (PP-DE_OBJ:* "P:*/(n\\n)" ADV:n\\n)

Rule :
→ (PP:* NP P VPinf)
← (PP:* "NP:*/(n\\n)"
     (:n\\n "P:(n\\n)/(np\\s_i)" VPinf:np\\s_i))
Rule :
→ (PP:* NP P ADV)
← (PP:* NP:np (:np\\* P:np\\* "ADV:(np\\*)\\(np\\*)"))
Rule :
→ (PP:* P ADV ADV)
← (PP:* P:* (:*\\* ADV:*\\* "ADV:(*\\*)\\(*\\*)"))
Rule :
→ (PP:* P+D NP AP)
← (PP:* P+D:*/n (:n NP:n AP:n\\n))
Rule :
→ (PP:* P NP-SUJ tree)
← (PP:* P:*/s (:s NP-SUJ:np SENT:np\\s))

Rule :
→ (PP:* CLS CLO V NP)
← (PP:* CLS:np (:np\\* (":(np\\*)/np" "CLO:((np\\*)/np)/((np\\*)/np)" "V:(np\\*)/np") NP:np))

Rule :
→ (PP:* P ADJ (COORD CC tree))
← (PP:* P:*/np (:np ADJ:n\\n
       ("COORD:(n\\n)\\np" "CC:((n\\n)\\np)/np" NP:np)))
Rule : use-ponct
→ (NP-OBJ:* tree PONCT AP (COORD CC NP))
← (NP-OBJ:* NP:* (:*\\* "PONCT:(*\\*)/np" (:np AP:n\\n
         ("COORD:(n\\n)\\np" "CC:((n\\n)\\np)/np"
           NP:np))))
Rule :
→ (NP:* tree AdP-MOD)
← (NP:* NP:* AdP-MOD:*\\*)

Rule : use-ponct
→ (NP:* tree (ADJ-MOD PONCT ADJ PONCT))
← (NP:* NP:n (NC-MOD:n\\*
       (":(n\\*)/pf" "PONCT:((n\\*)/pf)/(n\\n)" ADJ:n\\n)
       PONCT:pf))

Rule :
→ (NP:* NC PP DET)
← (NP:* (:n NC:n PP:n\\n) DET:n\\*)

Rule :
→ (NP:* CLO tree)
← (NP:* CLO:*/n NP:n)

Rule :
→ (NP:* tree PP (COORD CC ADV))
← (NP:* NP:n (:n\\* PP:n\\n ("COORD:(n\\n)\\(n\\*)" "CC:((n\\n)\\(n\\*))/(n\\n)" ADV:n\\n)))

Rule :
→ (NP:* tree (COORD CC AP NP))
← (NP:* NP:n
     (COORD:n\\* "CC:(n\\*)/np" (:np AP:np/np NP:np)))
Rule : use-ponct
→ (NP:* NC ADV PP PONCT)
← (NP:* (:n NC:n (:n\\n "ADV:(n\\n)/(n\\n)" PP:n\\n))
     PONCT:n\\*)
Rule :
→ (NP:* tree (COORD CC AdP))
← (NP:* NP:* (COORD:*\\* "CC:(*\\*)/(n\\n)" AdP:n\\n))
Rule :
→ (NP:* PRO PP)
← (NP:* PRO:np PP:np\\*)
Rule : use-ponct
→ (NP:* tree (Sint-MOD PONCT Sint-MOD PONCT))
← (NP:* NP:* (:*\\* (":(*\\*)/pf"
         "PONCT:((*\\*)/pf)/(*\\*)" Sint-MOD:*\\*)
       PONCT:pf))
Rule :
→ (NP-OBJ:* tree VPinf-MOD)
← (NP-OBJ:* NP:n VPinf-MOD:n\\*)
Rule :
→ (NP-MOD:* NC NPP NPP)
← (NP-MOD:* NC:*/np (:np NPP:np/np NPP:np))

Rule :
→ (NP-MOD:* tree (COORD CC NP PP))
← (NP-MOD:* (:* NP:*
       (COORD:*\\* "CC:(*\\*)/np" (:np NP:np PP:np\\np))))
Rule :
→ (NP-ATS:* ADV DET tree)
← (PP:np\\np ADV:*/* (:* DET:*/n NP:n))
Rule :
→ (NP-ATS:* P tree)
← (NP-ATS:* P:*/n NP:n)
Rule :
→ (NP:* CLS tree)
← (NP:* CLS:np VN:np\\*)
Rule :
→ (NP:* NC Srel Sint-MOD)
← (NP:* (:* NC:n Srel:n\\*) Sint-MOD:*\\*)
Rule :
→ (VN:* CLO-SUJ tree)
← (VN:* CLO-SUJ:np VN:np\\*)
Rule :
→ (VN:* CLO-P_OBJ tree)
← (VN:* "CLO-P_OBJ:*/(*/pp)" VN:*/pp)
Rule :
→ (VN:* NP-OBJ tree)
← (VN:* NP-OBJ:np VN:np\\*)

Rule :
→ (VN:* V (NP-OBJ PRO) VPP)
← (VN:* "V:*/(np\\s_p)" (:np\\s_p "PRO:(np\\s_p)/((np\\s_p)/np)" "VPP:(np\\s_p)/np"))
```

Rule :
→ (VN:* V P)
← (VN:* V:*/pp P:pp)

Rule :
→ (VN:* tree PROREL)
← (VN:* VN:*/pp PROREL:pp)

Rule :
→ (VN:* tree (COORD CC ADV VN))
← (VN:* VN:*
(COORD:** "CC:(**)/*" (:* ADV:*/* VN:*)))

Rule :
→ (VN:* tree (COORD CC VN ADV))
← (VN:* VN:*
(COORD:** "CC:(**)/*" (:* VN:* ADV:**)))

Rule :
→ (VN:* V (Sint tree))
← (VN:* V:*/s (Sint:s SENT:s))

Rule :
→ (VN:* DET V)
← (VN:* DET:np/n "V:(np/n)*")

Rule :
→ (VN:* V VPP (COORD CC tree VPP))
← (VN:* "V:*/(np\\s_p)" (:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)"
(:np\\s_p "ADV:(np\\s_p)/(np\\s_p)" VPP:np\\s_p))))

Rule : use-ponct
→ (VN:* tree PONCT ADV VPP PONCT)
← (VN:* "VN:*/(np\\s_p)" (:np\\s_p
(":(np\\s_p)/gf" "PONCT:((np\\s_p)/gf)/(np\\s_p)" (:np\\s_p "ADV:(np\\s_p)/(np\\s_p)" VPP:np\\s_p)) PONCT:gf))

Rule :
→ (VN:* VPR ADV VPR)
← (VN:* "VPR:*/(np\\s)"
(:np\\s "ADV:(np\\s)/(np\\s)" VPR:np\\s))

Rule :
→ (VN:* tree CLS-MOD)
← (VN:* VN:* CLS-MOD:**)

Rule :
→ (Srel:* (PP (NP-MOD P+PRO) tree))
← (Srel:* P+PRO:*/s Srel:s)

Rule :
→ (Srel:* (PP P NP-MOD tree))
← (Srel:* P:pp/np
(":(pp/np)*" "NP-MOD:((pp/np)*)/s" Srel:s))

Rule :
→ (Srel:* (NP-OBJ CS) tree)
← (Srel:* (NP-OBJ:*/s CS:*/s) SENT:s)

Rule :
→ (Srel:* NP-SUJ VN)
← (Srel:* NP-SUJ:np VN:np*)

Rule :
→ (Srel:* NP-SUJ VN ADV)
← (Srel:* (:* NP-SUJ:np VN:np*) ADV:**)

Rule :
→ (Srel:* ADVWH tree)
← (Srel:* ADVWH:*/* SENT:*)

Rule :
→ (VPpart-ATS:* VPP ADV)
← (VPpart-ATS:* VPP:* ADV:**)

Rule :
→ (VPpart-ATO:* VPP VPinf)
← (VPpart-ATO:* VPP:np\\s_p "VPinf:(np\\s_p)*")

Rule :
→ (VPpart:* tree AP-MOD)
← (VPpart:* VPpart:* AP-MOD:**)

Rule :
→ (VPinf-OBJ:* VN AdP-MOD PP-MOD)
← (VPinf-OBJ:* (:* VN:* AdP-MOD:**) PP-MOD:**)

Rule :
→ (VPinf-DE_OBJ:* P VN NP-OBJ VPinf-ATO)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)"
(:np\\s_i (":(np\\s_i)/(np\\s_i)" "VN:((np\\s_i)/(np\\s_i))/np" NP-OBJ:np) VPinf-ATO:np\\s_i))

Rule :
→ (VPinf-SUJ:* VN NP-OBJ PP-P_OBJ)
← (VPinf-SUJ:* (:*/pp "VN:(*/pp)/np" NP-OBJ:np)
PP-P_OBJ:pp)

Rule :
→ (VPinf-ATS:* VN PP-A_OBJ)
← (VPinf-ATS:* VN:*/pp_a PP-A_OBJ:pp_a)

Rule :
→ (VPinf-ATS:* VN Ssub-OBJ)
← (VPinf-ATS:* VN:*/cs Ssub-OBJ:cs)

Rule :
→ (VPinf-ATS:* VN AP-ATS)
← (VPinf-ATS:* "VN:*/(n\\n)" AP-ATS:n\\n)

Rule :
→ (VPinf:* VINF ADV)
← (VPinf:* VINF:* ADV:**)

Rule :
→ (VPinf-ATS:* ADV VN VPinf-OBJ)
← (VPinf-ATS:* ADV:*/*
(:* "VN:*/(np\\s_i)" VPinf-OBJ:np\\s_i))

Rule :
→ (VPinf-MOD:* P VN NP-MOD)
← (VPinf-MOD:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "NP-MOD:(np\\s_i)\\(np\\s_i)"))

Rule :
→ (VPinf-OBJ:* VINF NP PP)
← (VPinf-OBJ:* (:*/pp "VINF:(*/pp)/np" NP:np) PP:pp)

Rule :
→ (VPinf-OBJ:* P VN VPinf-A_OBJ)
← (VPinf-OBJ:* "P:*/(np\\s_i)" (:np\\s_i
"VN:(np\\s_i)/(np\\s_i)" VPinf-A_OBJ:np\\s_i))

Rule :
→ (VPinf-ATS:* VINF Ssub)
← (VPinf-ATS:* VINF:*/cs Ssub:cs)

Rule :
→ (VPinf-MOD:* tree VPinf-MOD)
← (VPinf-MOD:* VPinf:* VPinf-MOD:**)

Rule :
→ (VPinf:* Ssub-MOD tree)
← (VPinf:* Ssub-MOD:*/* VPinf:*)

```
Rule :
→ (VPinf:* tree PP-ATS)
← (VPinf:* VPinf:*/pp PP-ATS:pp)
Rule :
→ (AP:* ADJ NPP)
← (AP:* ADJ:*/np NPP:np)
Rule :
→ (AP:* tree PP-A_OBJ)
← (AP:* AP:*/pp_a PP-A_OBJ:pp_a)
Rule :
→ (AP-MOD:* ADJ NP)
← (AP-MOD:* ADJ:*/np NP:np)
Rule :
→ (AP-MOD:* tree VPinf)
← (AP-MOD:* AP:* VPinf:*\\*)
Rule :
→ (AP:* tree VPinf-MOD)
← (AP:* AP:* VPinf-MOD:*\\*)
Rule :
→ (AP:* tree VPP)
← (AP:* "AP:*/(n\\n)" VPP:n\\n)
Rule :
→ (AP:* tree NC PONCT NC)
← (AP:* AP:*/n (:n NC:n (:n\\n "PONCT:(n\\n)/n" NC:n)))
Rule :
→ (AP-ATS:* AdP tree)
← (NC:n AdP:*/* AP:*)
Rule :
→ (AP:* NP (COORD CC PP))
← (AP:* NP:np (COORD:np\\* "CC:(np\\*)/*" PP:*))
Rule :
→ (AP:* tree (COORD CC VPinf))
← (AP:* AP:* (COORD:*\\* "CC:(*\\*)/*" VPinf:*))
Rule :
→ (AP:* (COORD CC AP) (COORD CC tree))
← (AP:* (COORD:*/* "CC:(*/*)/(n\\n)" AP:n\\n)
     (COORD:* "CC:*/(n\\n)" AP:n\\n))
Rule :
→ (Ssub-OBJ:* ADV CS tree)
← (Ssub-OBJ:* ADV:*/* (:* CS:*/s SENT:s))
```

```
Rule :
→ (AdP-MOD:* P NP)
← (AdP-MOD:* P:*/np NP:np)
Rule :
→ (AdP-MOD:* ADV ADJ)
← (AdP-MOD:* "ADV:*/(n/n)" ADJ:n/n)
Rule :
→ (AdP-MOD:* P ADV)
← (AdP-MOD:* P:*/* ADV:*)
Rule :
→ (AdP-MOD:* tree AP)
← (AdP-MOD:* "AdP:*/(n\\n)" AP:n\\n)
Rule :
→ (AdP:* CC ADV)
← (AdP:* CC:*/* ADV:*)
Rule : use-ponct
→ (AdP:* PP PONCT PP)
← (AdP:* PP:* (:*\\* "PONCT:(*\\*)/*" PP:*))
Rule : use-ponct
→ (AdP:* PONCT ADV)
← (AdP:* PONCT:*/* ADV:*)
Rule :
→ (COORD-MOD:* CC Ssub)
← (COORD-MOD:* CC:*/cs Ssub:cs)
Rule :
→ (COORD-MOD:* CC VPpart)
← (COORD-MOD:* "CC:*/(np\\s_p)" VPpart:np\\s_p)
Rule :
→ (COORD:* CC CC PP)
← (COORD:* CC:*/* (:* CC:*/pp PP:pp))
Rule :
→ (COORD:* CC NP PP)
← (COORD:* CC:*/np (:np NP:np PP:np\\np))
Rule :
→ (SENT:* VN (COORD CC ADV VPpart))
← (SENT:* VN:np\\s ("COORD:(np\\s)\\*"
       "CC:((np\\s)\\*)/(np\\s_p)" (:np\\s_p
         "ADV:(np\\s_p)/(np\\s_p)" VPpart:np\\s_p)))
Rule :
→ (SENT:* VN (COORD CC ADV NP))
← (SENT:* VN:*
     (COORD:*\\* "CC:(*\\*)/np" (:np ADV:np/np NP:np)))
```

```
Rule :
→ (SENT:* (VN tree VPP) (COORD ADV VPpart))
← (SENT:* (VN:* "VN:*/(np\\s_p)"
       (:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" "ADV:((np\\s_p)\\(np\\s_p))/(np\\s_p)" VPpart:np\\s_p))))
```

```
Rule :
→ (SENT:* VN CLO)
← (SENT:* VN:*/np CLO:np)
```

```
Rule : use-ponct
→ (VPinf-A_OBJ:* P VN PP-MOD PONCT PP-MOD PP-MOD (COORD CC PP PP))
← (VPinf-A_OBJ:* P:*/* (:* VN:* (:*\\* PP-MOD:*\\* (":(*\\*)\\(*\\*)" "PONCT:((*\\*)\\(*\\*))/(*\\*)" (:*\\*
             (:*\\* PP-MOD:*\\* "PP-MOD:(*\\*)\\(*\\*)")
             ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" (:*\\* PP:*\\* "PP:(*\\*)\\(*\\*)")))))))
```

```
Rule :
→ (VPpart:* VPP VPP PP)
← (VPpart:* VPP:* (:*\\* "VPP:(*\\*)/pp" PP:pp))
```

```
Rule :
→ (SENT:* VN NP-OBJ (COORD CC NP NP) (COORD CC NP NP))
← (SENT:* VN:*/np (:np (:np NP-OBJ:np (COORD:np\\np
          (":(np\\np)/np" "CC:((np\\np)/np)/np" NP:np)
          NP:np))
       (COORD:np\\np
         (":(np\\np)/np" "CC:((np\\np)/np)/np" NP:np)
         NP:np)))
```

Rule : use-ponct
```
→ (SENT (NP-SUJ DET NC (PP P+D (NP NC PONCT COORD))) PONCT VN VPinf-A_OBJ PONCT)
← (TEXT:txt (SENT:s (NP-SUJ:np DET:np/n (:n NC:n (PP:n\\n "P+D:(n\\n)/n" NC:n))) (:np\\s (":(np\\s)/(np\\s)"
          (":(np\\s)/(np\\s)" (":(np\\s)/(np\\s)" "PONCT:((np\\s)/(np\\s))/((np\\s)/(np\\s))" "COORD:(np\\s)/(np\\s)")
            "PONCT:((np\\s)/(np\\s))\\((np\\s)/(np\\s))"))
        (:np\\s "VN:(np\\s)/(np\\s_i)" VPinf-A_OBJ:np\\s_i)))
    PONCT:s\\txt)
```
Rule : use-ponct
```
→ (NP-OBJ:* DET NC (Ssub NP NP) PONCT VN ADV Ssub-OBJ)
← (NP-OBJ:* (NP:np DET:np/n NC:n) (Ssub:np\\* "NP:(np\\*)/s" (:s NP:np (:np\\s "PONCT:(np\\s)/(np\\s)"
          (:np\\s (":(np\\s)/cs" "VN:(np\\s)/cs" "ADV:((np\\s)/cs)\\((np\\s)/cs)") Ssub-OBJ:cs)))))
```
Rule : use-ponct
```
→ (Ssub-DE_OBJ:* P PRO CS PONCT NP-SUJ VN NP-OBJ PONCT
    (COORD CC PONCT VN VPinf-OBJ PONCT))
← (Ssub-DE_OBJ:* (:*/s "P:(*/s)/(*/s)"
      (:*/s "PRO:(*/s)/(*/s)" CS:*/s))
    (:s (:s (:s/gf "PONCT:(s/gf)/s" (:s NP-SUJ:np
            (:np\\s "VN:(np\\s)/np" NP-OBJ:np)))
        PONCT:gf)
      (COORD:s\\s "CC:(s\\s)/s" (:s (:s/gf
            "PONCT:(s/gf)/s"
            (:s "VN:s/(np\\s_i)" VPinf-OBJ:np\\s_i))
          PONCT:gf))))
```
Rule : use-ponct
```
→ (Sint:* NP-SUJ VN (Ssub-OBJ CS NP) PONCT VN NP-OBJ)
← (Sint:* NP-SUJ:np (:np\\* "VN:(np\\*)/cs" (Ssub-OBJ:cs
        CS:cs/s (:s (:np NP:np PONCT:np\\np)
          (:np\\s "VN:(np\\s)/np" NP-OBJ:np)))))
```
Rule : use-ponct
```
→ (Sint:* PP-A_OBJ VN VPinf-OBJ PONCT (Ssub-MOD CS NP) PONCT VN ADV NP-OBJ)
← (Sint:* (:* (:* (:* PP-A_OBJ:pp_a (:pp_a\\* "VN:(pp_a\\*)/(np\\s_i)" VPinf-OBJ:np\\s_i)) PONCT:*\\*) (Ssub-MOD:*\\*
        "CS:(*\\*)/s" (:s (:np NP:np PONCT:np\\np)
          (:np\\s (":(np\\s)/np" "VN:(np\\s)/np" "ADV:((np\\s)/np)\\((np\\s)/np)") NP-OBJ:np)))))
```
Rule : use-ponct
```
→ (SENT
    (COORD CC NP-SUJ VN ADV PONCT NP-SUJ PONCT VN AP-ATS)
    PONCT)
← (TEXT:s (SENT:s (COORD:s CC:s/s (:s (:s NP-SUJ:np
            (:np\\s VN:np\\s "ADV:(np\\s)\\(np\\s)"))
          (:s\\s "PONCT:(s\\s)/s" (:s
              (:np NP-SUJ:np PONCT:np\\np) (:np\\s
                "VN:(np\\s)/(n\\n)" AP-ATS:n\\n))))))
    PONCT:s\\s)
```
Rule :
```
→ (NP-OBJ:* NC VPinf-A_OBJ)
← (NP-OBJ:* NC:n VPinf-A_OBJ:n\\*)
```
Rule : use-ponct
```
→ (SENT VN PP-MOD PONCT (COORD-MOD CC ADJ PONCT PP) PONCT PONCT COORD-MOD PONCT)
← (TEXT:s (SENT:s (:s (:s VN:s PP-MOD:s\\s) PONCT:s\\s) (:s\\s (COORD-MOD:s\\s "CC:(s\\s)/(n\\n)" (:n\\n ADJ:n\\n
            (":(n\\n)\\(n\\n)" (":((n\\n)\\(n\\n))/pf" "PONCT:(((n\\n)\\(n\\n))/pf)/(n\\n)" PP:n\\n) PONCT:pf)))
        (":(s\\s)\\(s\\s)" "PONCT:((s\\s)\\(s\\s))/(s\\s)" COORD-MOD:s\\s)))
    PONCT:s\\s)
```
Rule : use-ponct
```
→ (COORD-MOD:* CC AP PP (PP P (NP DET NC (COORD CC NP) AP PONCT tree)) PONCT)
← (COORD-MOD:* "CC:*/(n\\n)" (:n\\n (:n\\n AP:n\\n "PP:(n\\n)\\(n\\n)") ("PP:(n\\n)\\(n\\n)" "P:((n\\n)\\(n\\n))/np"
        (NP:np DET:np/n (:n (:n NC:n (COORD:n\\n "CC:(n\\n)/n" NP:n)) (:n\\n AP:n\\n (":(n\\n)\\(n\\n)"
                (":((n\\n)\\(n\\n))/pf" "PONCT:(((n\\n)\\(n\\n))/pf)/(n\\n)" NP-MOD:n\\n) PONCT:pf)))))))
```
Rule :
```
→ (PP:* P NP (COORD ADV ADV PP-MOD))
← (PP:* (:* P:*/np NP:np) (COORD:*\\* (":(*\\*)/*" "ADV:(*\\*)/*" "ADV:((*\\*)/*)\\((*\\*)/*)") PP-MOD:*))
```
Rule :
```
→ (NP:* DET NC ADV Sint-MOD (COORD CC (NP DET NC AP))
    Sint-MOD)
← (NP:*
    (:* DET:*/n (:n (:n NC:n ADV:n\\n) Sint-MOD:n\\n))
    (COORD:*\\* "CC:(*\\*)/*" (NP:* DET:*/n
        (:n (:n NC:n AP:n\\n) Sint-MOD:n\\n))))
```
Rule : use-ponct
```
→ (SENT NP-MOD PONCT NP-SUJ VN PP-P_OBJ PONCT
    (PP-MOD P NP) P PONCT)
← (TEXT:s (SENT:s NP-MOD:s/s (:s PONCT:s/s (:s NP-SUJ:np
          (:np\\s (:np\\s
              (:np\\s "VN:(np\\s)/pp" PP-P_OBJ:pp)
              "PONCT:(np\\s)\\(np\\s)")
            ("PP-MOD:(np\\s)\\(np\\s)"
              "P:((np\\s)\\(np\\s))/np"
              (:np NP:np P:np\\np))))))
    PONCT:s\\s)
```

```
Rule :
→ (PP:* P ADJ ADV (VPpart VPP ADV (COORD CC ADV PP)))
← (PP:* P:*/n (:n ADJ:n (:n\\n "ADV:(n\\n)/(n\\n)" (:n\\n "VPP:(n\\n)/pp"
          (:pp (:pp/pp ADV:pp/pp ("COORD:(pp/pp)\\(pp/pp)" "CC:((pp/pp)\\(pp/pp))/(pp/pp)" ADV:pp/pp)) PP:pp)))))
```

```
Rule :
→ (SENT:* VN AP-ATS (COORD CC NP))
← (SENT:* "VN:*/(n\\n)" (:n\\n AP-ATS:n\\n
      ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/np"
        NP:np)))
Rule :
→ (Ssub-P_OBJ:* P NP)
← (NP:np P:*/np NP:np)
Rule :
→ (VPpart-ATO:* ADV VPP VPinf)
← (VPpart-ATO:* ADV:*/*
    (:* "VPP:*/(np\\s_i)" VPinf:np\\s_i))
Rule : use-ponct
→ (SENT ADV PONCT VN NP-SUJ PONCT VPpart PONCT)
← (TEXT:s (SENT:s (:s/s ADV:s/s "PONCT:(s/s)\\(s/s)") (:s
        (":s/(np\\s_p)" "VN:(s/(np\\s_p))/np"
          (:np NP-SUJ:np PONCT:np\\np))
        VPpart:np\\s_p))
    PONCT:s\\s)
```

```
Rule : use-ponct
→ (SENT NP-SUJ VN PP-MOD PP-MOD (COORD ADV VN NP-OBJ)
    PONCT)
← (TEXT:s (SENT:s (:s
        (:s (:s NP-SUJ:np VN:np\\s) PP-MOD:s\\s)
        PP-MOD:s\\s)
      (COORD:s\\s "ADV:(s\\s)/s" (:s VN:s/np NP-OBJ:np)))
    PONCT:s\\s)
Rule : use-ponct
→ (VPpart:* VPP PP PONCT PP PONCT VPinf)
← (VPpart:* (":*/(np\\s_i)" "VPP:(*/(np\\s_i))/pp" (:pp
        PP:pp (:pp\\pp "PONCT:(pp\\pp)/pp"
          (:pp PP:pp PONCT:pp\\pp))))
    VPinf:np\\s_i)
```

```
Rule : use-ponct
→ (VPpart-MOD:* (VN P VPR) PONCT NP-MOD PONCT PP-MOD PONCT Ssub-OBJ)
← (VPpart-MOD:* (VN:* "P:*/(np\\s_p)" (:np\\s_p (":(np\\s_p)/cs" (":(np\\s_p)/cs" (":(np\\s_p)/cs" (":(np\\s_p)/cs"
                (":(np\\s_p)/cs" "VPR:(np\\s_p)/cs" "PONCT:((np\\s_p)/cs)\\((np\\s_p)/cs)")
                "NP-MOD:((np\\s_p)/cs)\\((np\\s_p)/cs)")
              "PONCT:((np\\s_p)/cs)\\((np\\s_p)/cs)")
            "PP-MOD:((np\\s_p)/cs)\\((np\\s_p)/cs)")
          "PONCT:((np\\s_p)/cs)\\((np\\s_p)/cs)")
        Ssub-OBJ:cs)))
```

```
Rule : use-ponct
→ (VPinf-OBJ:* P VN NP-OBJ Sint-MOD CS (Ssub-MOD PP))
← (VPinf-OBJ:* P:*/* (:*
      (:* (:* VN:*/np NP-OBJ:np) Sint-MOD:*\\*)
      (Ssub-MOD:*\\* "CS:(*\\*)/pp" PP:pp)))
Rule :
→ (NP:* DET NC P NP VPP PP)
← (NP:* DET:*/n (:n
      (:n (:n NC:n (:n\\n "P:(n\\n)/np" NP:np)) VPP:n\\n)
      PP:n\\n))
```

```
Rule : use-ponct
→ (SENT (PP P NP PONCT (COORD CC PP)) PONCT
    (PP P NP (COORD CC PP)) PONCT)
← (SENT:s (:pp (PP:pp (:pp P:pp/np NP:np) (:pp\\pp
          "PONCT:(pp\\pp)/(pp\\pp)"
          (COORD:pp\\pp "CC:(pp\\pp)/pp" PP:pp)))
      (:pp\\pp "PONCT:(pp\\pp)/pp" (PP:pp
          (:pp P:pp/np NP:np)
          (COORD:pp\\pp "CC:(pp\\pp)/pp" PP:pp))))
    PONCT:pp\\s)
Rule :
→ (PP:* PP NP)
← (PP:* PP:*/np NP:np)
```

```
Rule : use-ponct
→ (SENT:* (VN V VPP VPP) NP-ATS (COORD CC PONCT PP PONCT VPpart))
← (SENT:* (VN:* "V:*/(np\\s_p)" (:np\\s_p (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)" (:np\\s_p "VPP:(np\\s_p)/np" NP-ATS:np))
        ("COORD:(np\\s_p)\\(np\\s_p)" (":((np\\s_p)\\(np\\s_p))/(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)"
            (":(((np\\s_p)\\(np\\s_p))/(np\\s_p))\\(((np\\s_p)\\(np\\s_p))/(np\\s_p))"
              "PONCT:((((np\\s_p)\\(np\\s_p))/(np\\s_p))\\(((np\\s_p)\\(np\\s_p))/(np\\s_p)))/pp"
              (:pp PP:pp PONCT:pp\\pp)))
          VPpart:np\\s_p))))
Rule :
→ (VN:* V VPP PP-MOD (COORD CC (VN VPP) PP-MOD))
← (VN:* "V:*/(np\\s_p)" (:np\\s_p (:np\\s_p VPP:np\\s_p "PP-MOD:(np\\s_p)\\(np\\s_p)") ("COORD:(np\\s_p)\\(np\\s_p)"
      "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)" (:np\\s_p (VN:np\\s_p VPP:np\\s_p) "PP-MOD:(np\\s_p)\\(np\\s_p)"))))
Rule :
→ (VN:* V ADV (COORD CC PP))
← (VN:* V:* (:*\\* ADV:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" PP:*\\*)))
Rule :
→ (VN:* V VPP (COORD CC VPP ADV))
← (VN:* "V:*/(np\\s_p)" (:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)"
      (:np\\s_p VPP:np\\s_p "ADV:(np\\s_p)\\(np\\s_p)"))))
```

Rule : use-ponct
```
→ (VN:* V VPP PONCT (COORD CC VPP))
← (VN:* "V:*/(np\\s_p)" (:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" (":((np\\s_p)\\(np\\s_p))/np\\s_p"
          "PONCT:(((np\\s_p)\\(np\\s_p))/np\\s_p)/(((np\\s_p)\\(np\\s_p))/np\\s_p)"
          "CC:((np\\s_p)\\(np\\s_p))/np\\s_p")
        VPP:np\\s_p)))
```

Rule :
```
→ (VN:* P VPR)
← (VPP:np\\s_p "P:*/(np\\s_p)" VPR:np\\s_p)
```

Rule :
```
→ (VN:* V ADJWH)
← (VN:* V:*/np ADJWH:np)
```

Rule :
```
→ (SENT:* (VN CLS-SUJ ADV V (NP-OBJ PRO) VPP))
← (SENT:* CLS-SUJ:np (VN:np\\* "ADV:(np\\*)/(np\\*)"
      (VN:np\\* "V:(np\\*)/(np\\s_p)" (VPP:np\\s_p "PRO:(np\\s_p)/((np\\s_p)/np)" "VPP:(np\\s_p)/np"))))
```

Rule :
```
→ (SENT:* (VN CLS-SUJ tree VPP) PP-A_OBJ NP-OBJ)
← (SENT:* CLS-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p ("VPP:(np\\s_p)/np"
          "VPP:((np\\s_p)/np)/pp_a" PP-A_OBJ:pp_a)
        NP-OBJ:np)))
```

Rule : use-ponct
```
→ (SENT:* NP-SUJ PONCT (VN tree VPP) ADV NP-OBJ VPinf-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "PONCT:(np\\*)/(np\\*)" (VN:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
          ("VPP:(np\\s_p)/(np\\s_i)" ("VPP:((np\\s_p)/(np\\s_i))/np" "VPP:((np\\s_p)/(np\\s_i))/np"
              "ADV:(((np\\s_p)/(np\\s_i))/np)\\(((np\\s_p)/(np\\s_i))/np)")
            NP-OBJ:np)
          VPinf-A_OBJ:np\\s_i))))
```

Rule :
```
→ (SENT (NP-MOD NC))
← (SENT:n (NP-MOD:n NC:n))
```
Rule : use-ponct
```
→ (SENT PP NP PONCT)
← (TEXT:txt (SENT:np PP:np/np NP:np) PONCT:np\\txt)
```
Rule : use-ponct
```
→ (SENT ADVWH NP PONCT)
← (TEXT:txt (SENT:np ADVWH:np/np NP:np) PONCT:np\\txt)
```

Rule :
```
→ (SENT tree VPinf-MOD)
← (SENT:s SENT:s VPinf-MOD:s\\s)
```
Rule :
```
→ (SENT VPpart)
← (SENT:np\\s_p VPpart:np\\s_p)
```

Rule :
```
→ (SENT:* NP-SUJ (VN V NP-MOD VPP) Ssub-OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "V:(np\\*)/(np\\s_p)"
      (:np\\s_p "NP-MOD:(np\\s_p)/(np\\s_p)" (VPP:np\\s_p "VPP:(np\\s_p)/cs" Ssub-OBJ:cs))))
```

Rule :
```
→ (SENT:* NP-SUJ (VN tree VPP VPP) NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (:np\\s_p "VPP:(np\\s_p)/(np\\s_p)"
        (VPP:np\\s_p "VPP:(np\\s_p)/np" NP-ATS:np))))
```

Rule :
```
→ (SENT:* NP-SUJ (VN tree VPP) PP-MOD PP-OBJ)
← (SENT:* NP-SUJ:np (VN:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "PP-MOD:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-OBJ:pp)))
```
Rule :
```
→ (SENT:* NP-SUJ (VN tree VPP) ADV VPinf-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        ("VPP:(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "ADV:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
        VPinf-A_OBJ:np\\s_i)))
```
Rule :
```
→ (SENT:* NP-SUJ (VN tree VPP) VPinf-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "VPinf-MOD:((np\\s_p)/np)\\((np\\s_p)/np)") NP-OBJ:np)))
```
Rule :
```
→ (SENT:* NP-SUJ (VN tree VPP) AdP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "AdP-MOD:((np\\s_p)/np)\\((np\\s_p)/np)") NP-OBJ:np)))
```

```
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) AdP-MOD PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/pp_de" "VPP:(np\\s_p)/pp_de" "AdP-MOD:((np\\s_p)/pp_de)\\((np\\s_p)/pp_de)") PP-DE_OBJ:pp_de)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/cs" "VPP:(np\\s_p)/cs" "ADV:((np\\s_p)/cs)\\((np\\s_p)/cs)") Ssub-OBJ:cs)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "ADV:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-OBJ:pp)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) NP-MOD VPpart-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(n\\n)" "VPP:(np\\s_p)/(n\\n)" "NP-MOD:((np\\s_p)/(n\\n))\\((np\\s_p)/(n\\n))")
        VPpart-ATS:n\\n)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) VPinf-MOD PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/pp" "VPP:(np\\s_p)/pp" "VPinf-MOD:((np\\s_p)/pp)\\((np\\s_p)/pp)") PP-ATS:pp)))
```

```
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) PP-P_OBJ PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p ("VPP:(np\\s_p)/(n\\n)"
          "VPP:((np\\s_p)/(n\\n))/pp" PP-P_OBJ:pp)
        PP-ATS:n\\n)))
```

```
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)"
      (VPP:np\\s_p (":(np\\s_p)/np" "VPP:(np\\s_p)/np" "ADV:((np\\s_p)/np)\\((np\\s_p)/np)") NP-ATS:np)))
Rule :
→ (SENT:* NP-SUJ (VN tree VPP) ADV VPinf-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(np\\s_p)" (VPP:np\\s_p
        (":(np\\s_p)/(np\\s_i)" "VPP:(np\\s_p)/(np\\s_i)" "ADV:((np\\s_p)/(np\\s_i))\\((np\\s_p)/(np\\s_i))")
        VPinf-ATS:np\\s_i)))
```

```
Rule :
→ (SENT PP)
← (SENT:pp PP:pp)
```

```
Rule : use-ponct
→ (SENT:* (PONCT-MOD PONCT PONCT PONCT) tree)
← (SENT:* (PONCT-MOD:*/*
      (":(*/*)/pf" "PONCT:((*/*)/pf)/(*/*)" PONCT:*/*)
      PONCT:pf)
    SENT:*)
```

```
Rule :
→ (SENT:* (VN CLS-A_OBJ tree) NP-OBJ)
← (SENT:* CLS-A_OBJ:np
    (:np\\* "VN:(np\\*)/np" NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN VPpart-OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(np\\s_p)" VPpart-OBJ:np\\s_p))
Rule :
→ (SENT:* NP-SUJ VN AP-OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* "VN:(np\\*)/(n\\n)" AP-OBJ:n\\n))
```

```
Rule :
→ (SENT:* NP-SUJ VN AP-ATS PONCT AP-ATS PONCT AP-ATS)
← (SENT:* NP-SUJ:np (:np\\* "VN:(np\\*)/(n\\n)" (AP-ATS:n\\n AP-ATS:n\\n (":(n\\n)\\(n\\n)"
          "PONCT:((n\\n)\\(n\\n))/(n\\n)"
          (:n\\n AP-ATS:n\\n (":(n\\n)\\(n\\n)" "PONCT:((n\\n)\\(n\\n))/(n\\n)" AP-ATS:n\\n))))))
Rule :
→ (SENT:* NP-SUJ VN VPinf-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "VPinf-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN VPpart-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" "VN:(np\\s)/np" "VPpart-MOD:((np\\s)/np)\\((np\\*)/np)") NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "PP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-OBJ:pp))
```

```
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-MOD PP-A_OBJ)
← (SENT:* CLS-SUJ:np
     (:np\\* (":(np\\*)/pp_a" "VN:(np\\s)/pp_a" "NP-MOD:((np\\s)/pp_a)\\((np\\*)/pp_a)") PP-A_OBJ:pp_a))
Rule :
→ (SENT:* NP-SUJ VN VPinf-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "VPinf-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-MOD Ssub-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\s)/cs" "NP-MOD:((np\\s)/cs)\\((np\\*)/cs)") Ssub-OBJ:cs))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD VPpart-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
       (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "PP-MOD:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
       VPpart-OBJ:np\\s_p))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-MOD NP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
         "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN ADV AdP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
         "AdP-MOD:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
         "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
         "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD NP-MOD NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
         "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "AdP-MOD:((np\\*)/np)\\((np\\*)/np)")
         "ADV:((np\\*)/np)\\((np\\*)/np)")
       NP-OBJ:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV PP-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" (":(np\\*)/pp" "VN:(np\\*)/pp" "ADV:((np\\*)/pp)\\((np\\*)/pp)")
         "ADV:((np\\*)/pp)\\((np\\*)/pp)")
       PP-OBJ:pp))
Rule :
→ (SENT:* NP-SUJ VN ADV NP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
         "NP-MOD:((np\\*)/cs)\\((np\\*)/cs)")
       Ssub-OBJ:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV NP-MOD Ssub-OBJ)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
         "NP-MOD:((np\\*)/cs)\\((np\\*)/cs)")
       Ssub-OBJ:cs))
Rule :
→ (SENT:* NP-SUJ VN ADV PP-MOD Ssub-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/cs" (":(np\\*)/cs" "VN:(np\\*)/cs" "ADV:((np\\*)/cs)\\((np\\*)/cs)")
         "PP-MOD:((np\\*)/cs)\\((np\\*)/cs)")
       Ssub-OBJ:cs))
```

```
Rule :
→ (SENT:* NP-SUJ VN ADV PP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/(np\\s_i)"
        (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
        "PP-MOD:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* NP-SUJ VN PP-MOD PP-MOD VPinf-OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/(np\\s_i)"
        (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "PP-MOD:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
        "PP-MOD:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-OBJ:np\\s_i))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV PP-MOD NP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)")
        "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
      NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-MOD NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
        "PP-MOD:((np\\*)/np)\\((np\\*)/np)")
      NP-ATS:np))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD ADV NP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/np" (":(np\\*)/np" "VN:(np\\*)/np" "NP-MOD:((np\\*)/np)\\((np\\*)/np)")
        "ADV:((np\\*)/np)\\((np\\*)/np)")
      NP-ATS:np))
Rule :
→ (SENT:* (VN CLS-SUJ tree) NP-MOD Ssub-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/cs" "VN:(np\\*)/cs" "NP-MOD:((np\\*)/cs)\\((np\\*)/cs)") Ssub-ATS:cs))
Rule :
→ (SENT:* (VN CLS-SUJ tree) AdP-MOD AP-ATS)
← (SENT:* CLS-SUJ:np
    (:np\\* (":(np\\*)/(n\\n)" "VN:(np\\*)/(n\\n)" "AdP-MOD:((np\\*)/(n\\n))\\((np\\*)/(n\\n))") AP-ATS:n\\n))
Rule :
→ (SENT:* (VN CLS-SUJ tree) PP-MOD PP-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "PP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-ATS:pp))
Rule :
→ (SENT:* NP-SUJ VN AdP-MOD PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "AdP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-ATS:pp))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" "VN:(np\\s)/pp" "NP-MOD:((np\\s)/pp)\\((np\\*)/pp)") PP-ATS:pp))
Rule :
→ (SENT:* NP-SUJ VN ADV ADV PP-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp" (":(np\\*)/pp" "VN:(np\\*)/pp" "ADV:((np\\*)/pp)\\((np\\*)/pp)")
        "ADV:((np\\*)/pp)\\((np\\*)/pp)")
      PP-ATS:pp))
Rule :
→ (SENT:* NP-SUJ VN ADV ADV VPinf-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/(np\\s_i)"
        (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
        "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-ATS:np\\s_i))
Rule :
→ (SENT:* (VN CLS-SUJ tree) ADV ADV VPinf-ATS)
← (SENT:* CLS-SUJ:np (:np\\* (":(np\\*)/(np\\s_i)"
        (":(np\\*)/(np\\s_i)" "VN:(np\\*)/(np\\s_i)" "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
        "ADV:((np\\*)/(np\\s_i))\\((np\\*)/(np\\s_i))")
      VPinf-ATS:np\\s_i))
Rule :
→ (SENT:* NP-SUJ VN NP-MOD VPpart-ATS)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "NP-MOD:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
      VPpart-ATS:np\\s_p))
```

Rule :
```
→ (SENT:* NP-SUJ VN ADV ADV VPpart-ATS)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/(np\\s_p)"
        (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "ADV:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
        "ADV:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))")
      VPpart-ATS:np\\s_p))
```
Rule :
```
→ (SENT:* (VN CLS-SUJ tree) ADV PP-MOD Srel)
← (SENT:* CLS-SUJ:np (:np\\*
      (":(np\\*)/s" (":(np\\*)/s" "VN:(np\\*)/s" "ADV:((np\\*)/s)\\((np\\*)/s)") "PP-MOD:((np\\*)/s)\\((np\\*)/s)")
      Srel:s))
```

Rule :
```
→ (SENT:* NP-SUJ VN NP-OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/np" "VN:((np\\*)/np)/np" NP-OBJ:np)
      NP-OBJ:np))
```
Rule :
```
→ (SENT:* NP-SUJ VN PP-OBJ NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\*
      (":(np\\*)/np" "VN:((np\\*)/np)/pp" PP-OBJ:pp)
      NP-OBJ:np))
```
Rule :
```
→ (SENT:* NP-SUJ VN PP-DE_OBJ PP-A_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp_a"
        "VN:((np\\*)/pp_a)/pp_de" PP-DE_OBJ:pp_de)
      PP-A_OBJ:pp_a))
```

Rule :
```
→ (SENT:* NP-SUJ VN PP-A_OBJ PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp_de"
        "VN:((np\\*)/pp_de)/pp_a" PP-A_OBJ:pp_a)
      PP-DE_OBJ:pp_de))
```
Rule :
```
→ (SENT:* NP-SUJ VN PP-P_OBJ PP-DE_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (":(np\\*)/pp_de"
        "VN:((np\\*)/pp_de)/pp" PP-P_OBJ:pp)
      PP-DE_OBJ:pp_de))
```

Rule :
```
→ (SENT:* NP-SUJ NP-MOD VN PP-DE_OBJ)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/pp_de" "NP-MOD:((np\\*)/pp_de)/((np\\*)/pp_de)" "VN:(np\\*)/pp_de") PP-DE_OBJ:pp_de))
```
Rule :
```
→ (SENT NP PP (COORD CC PP))
← (SENT:np NP:np (:np\\np PP:np\\np (":(np\\np)\\(np\\np)" "CC:((np\\np)\\(np\\np))/(np\\np)" PP:np\\np)))
```

Rule :
```
→ (SENT NP VPpart)
← (PP:np\\np NP:np VPpart:np\\np)
```

Rule :
```
→ (SENT:* NP-SUJ ADV VN ADV ADV NP-OBJ)
← (SENT:* NP-SUJ:np (:np\\* "ADV:(np\\*)/(np\\*)" (:np\\* (":(np\\*)/np"
          (":(np\\*)/np" "VN:(np\\*)/np" "ADV:((np\\*)/np)\\((np\\*)/np)") "ADV:((np\\*)/np)\\((np\\*)/np)")
        NP-OBJ:np)))
```

Rule : use-ponct
```
→ (SENT:* tree PP-DE_OBJ PONCT PP PONCT PP)
← (SENT:* SENT:*/pp (:pp PP-DE_OBJ:pp_de (:pp_de\\pp
        "PONCT:(pp_de\\pp)/pp" (:pp PP:pp
          (:pp\\pp "PONCT:(pp\\pp)/pp" PP:pp)))))
```

Rule :
```
→ (SENT:* VN ADVWH (Sint-ATS tree))
← (PP:pp (:*/s VN:*/s "ADVWH:(*/s)\\(*/s)")
     (Sint-ATS:s SENT:s))
```

Rule :
```
→ (SENT:* NP-SUJ VN ADV VPP)
← (SENT:* NP-SUJ:np
    (:np\\* (":(np\\*)/(np\\s_p)" "VN:(np\\*)/(np\\s_p)" "ADV:((np\\*)/(np\\s_p))\\((np\\*)/(np\\s_p))") VPP:np\\s_p))
```
Rule : use-ponct
```
→ (SENT:* VPinf-SUJ VN PONCT ADV PONCT NP-ATS)
← (SENT:* VPinf-SUJ:np\\s_i (":(np\\s_i)\\*" (":((np\\s_i)\\*)/np" "VN:((np\\s_i)\\*)/np"
        (":(((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np)"
          "PONCT:((((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np))/((((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np))"
          (":(((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np)" "ADV:(((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np)"
            "PONCT:((((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np))\\((((np\\s_i)\\*)/np)\\(((np\\s_i)\\*)/np))")))
      NP-ATS:np))
```
Rule : use-ponct
```
→ (SENT:* NP-SUJ VN NP-OBJ PONCT VN PP-P_OBJ)
← (SENT:* NP-SUJ:np (:np\\* (:np\\* "VN:(np\\*)/np" NP-OBJ:np)
      (":(np\\*)\\(np\\*)" "PONCT:((np\\*)\\(np\\*))/(np\\*)" (:np\\* "VN:(np\\*)/pp" PP-P_OBJ:pp))))
```

```
Rule : use-ponct
→ (SENT PONCT NP PP)
← (SENT:np PONCT:np/np (:np NP:np PP:np\\np))
```

```
Rule :
→ (SENT:* NP Sint)
← (SENT:* NP:np Sint:np\\*)
```

```
Rule :
→ (SENT:* tree Srel-OBJ)
← (SENT:* SENT:*/cs Srel-OBJ:cs)
```

```
Rule :
→ (SENT:* tree NP-A_OBJ)
← (SENT:* SENT:*/np NP-A_OBJ:np)
```

```
Rule :
→ (SENT:* tree (Ssub-A_OBJ CS PP))
← (SENT:* SENT:*/cs
     (Ssub-A_OBJ:cs "CS:cs/(n\\n)" PP:n\\n))
```

```
Rule :
→ (SENT:* tree Sint-ATS)
← (SENT:* SENT:*/s Sint-ATS:s)
```

```
Rule :
→ (SENT:* tree VPinf-ATS (COORD CC VPinf))
← (SENT:* "SENT:*/(np\\s_i)"
    (:np\\s_i VPinf-ATS:np\\s_i ("COORD:(np\\s_i)\\(np\\s_i)" "CC:((np\\s_i)\\(np\\s_i))/(np\\s_i)" VPinf:np\\s_i)))
```

```
Rule :
→ (SENT:* tree VPinf-A_OBJ (COORD CC VPinf))
← (SENT:* "SENT:*/(np\\s_i)"
    (:np\\s_i VPinf-A_OBJ:np\\s_i ("COORD:(np\\s_i)\\(np\\s_i)" "CC:((np\\s_i)\\(np\\s_i))/(np\\s_i)" VPinf:np\\s_i)))
```

```
Rule :
→ (SENT:* NP-A_OBJ-MOD tree)
← (VPinf:np\\s_i NP-A_OBJ-MOD:*/* SENT:*)
```

```
Rule :
→ (SENT:* (CLO-MOD en) VN)
← (SENT:* (CLO-MOD:*/* en:*/*) VN:*)
```

```
Rule :
→ (SENT:* CLS tree)
← (SENT:* CLS:np SENT:np\\*)
```

```
Rule :
→ (SENT:* (ADV-MOD PONCT ADV PONCT) tree)
← (SENT:* (ADV-MOD:*/*
      (":(*/*)/pf" "PONCT:((*/*)/pf)/(*/*)" ADV:*/*)
      PONCT:pf)
    SENT:*)
```

```
Rule :
→ (SENT:* I NP)
← (SENT:* I:*/np NP:np)
```

```
Rule :
→ (SENT:* NP (COORD CC Ssub))
← (SENT:* NP:np (COORD:np\\* "CC:(np\\*)/cs" Ssub:cs))
```

```
Rule :
→ (SENT:* VN ADV VPinf)
← (SENT:* (":*/(np\\s_i)" "VN:*/(np\\s_i)" "ADV:(*/(np\\s_i))\\(*/(np\\s_i))") VPinf:np\\s_i)
```

```
Rule :
→ (SENT:* tree NP-ATS PP-MOD (COORD CC NP PP))
← (SENT:* SENT:*/np (:np (:np NP-ATS:np PP-MOD:np\\np)
      (COORD:np\\np "CC:(np\\np)/np"
        (:np NP:np PP:np\\np))))
```

```
Rule :
→ (SENT:* tree NP-ATS NP-MOD PP-MOD (COORD CC NP PP))
← (SENT:* SENT:*/np (:np
      (:np (:np NP-ATS:np NP-MOD:np\\np) PP-MOD:np\\np)
      (COORD:np\\np "CC:(np\\np)/np"
        (:np NP:np PP:np\\np))))
```

```
Rule :
→ (SENT:* VN NP-SUJ (COORD CC tree))
← (SENT:* (:* VN:*/np NP-SUJ:np)
     (COORD:*\\* "CC:(*\\*)/*" SENT:*))
```

```
Rule :
→ (SENT:* NP VN PP-P_OBJ PP)
← (SENT:* NP:np
     (:np\\* "VN:(np\\*)/pp" (:pp PP-P_OBJ:pp PP:pp\\pp)))
```

```
Rule :
→ (SENT:* VPR VPpart)
← (PP:pp\\pp "VPR:*/(np\\s_p)" VPpart:np\\s_p)
```

```
Rule :
→ (SENT:* tree (PP-A_OBJ P NP) (COORD CC CC NP-ATS))
← (SENT:* SENT:*/pp_a (PP-A_OBJ:pp_a P:pp_a/np
      (:np NP:np (COORD:np\\np (":(np\\np)/np" "CC:((np\\np)/np)/((np\\np)/np)" "CC:(np\\np)/np") NP-ATS:np))))
```

```
Rule :
→ (SENT:* tree (COORD CC PP PP-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/pp"
      (:pp PP:pp PP-MOD:pp\\pp)))
```

```
Rule :
→ (SENT:* tree (COORD CC CC NP-MOD PP-MOD))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/(*\\*)" "CC:((*\\*)/(*\\*))/((*\\*)/(*\\*))" "CC:(*\\*)/(*\\*)")
      (:*\\* NP-MOD:*\\* "PP-MOD:(*\\*)\\(*\\*)")))
```

```
Rule :
→ (SENT:* tree (COORD CC NP PP-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/np"
      (:np NP:np PP-MOD:np\\np)))
```

```
Rule :
→ (SENT:* tree (COORD CC NP ADV))
← (SENT:* SENT:*
     (COORD:*\\* "CC:(*\\*)/np" (:np NP:np ADV:np\\np)))
```

Rule :
```
→ (SENT:* tree (COORD CC CC tree VPinf-OBJ))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/*"
        "CC:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      (:* "SENT:*/(np\\s_i)" VPinf-OBJ:np\\s_i)))
```
Rule :
```
→ (SENT:* tree (COORD CC CC tree VPinf-A_OBJ))
← (SENT:* SENT:* (COORD:*\\* (":(*\\*)/*"
        "CC:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      (:* "SENT:*/(np\\s_i)" VPinf-A_OBJ:np\\s_i)))
```

Rule :
```
→ (SENT:* VN ADV AdP-MOD (COORD CC CC VPinf NP-MOD))
← (SENT:* (:* (:* VN:* ADV:*\\*) AdP-MOD:*\\*) (COORD:*\\*
      (":(*\\*)/(*\\*)" "CC:((*\\*)/(*\\*))/((*\\*)/(*\\*))" "CC:(*\\*)/(*\\*)")
      (:*\\* VPinf:*\\* "NP-MOD:(*\\*)\\(*\\*)")))
```

Rule :
```
→ (SENT:* VN AP-ATS (COORD CC NP AP NP-MOD))
← (SENT:* (:* "VN:*/(n\\n)" AP-ATS:n\\n) (COORD:*\\*
      "CC:(*\\*)/np" (:np NP:np
        (:np\\np AP:n\\n "NP-MOD:(n\\n)\\(np\\np)"))))
```

Rule : use-ponct
```
→ (SENT:* tree (COORD CC PONCT PP-MOD PONCT NP-MOD))
← (SENT:* SENT:* (COORD:*\\* "CC:(*\\*)/(*\\*)" (:*\\* "PONCT:(*\\*)/(*\\*)"
        (:*\\* PP-MOD:*\\* (":(*\\*)\\(*\\*)" "PONCT:((*\\*)\\(*\\*))/(*\\*)" NP-MOD:*\\*)))))
```
Rule : use-ponct
```
→ (SENT:* (VN VINF ADV VPP) PP-MOD PONCT NP-MOD PP-MOD PONCT (COORD CC ADV VPP PP NP-MOD))
← (SENT:* (VN:* "VINF:*/(np\\s_p)" (:np\\s_p (:np\\s_p "ADV:(np\\s_p)/(np\\s_p)" (:np\\s_p
            (:np\\s_p VPP:np\\s_p "PP-MOD:(np\\s_p)\\(np\\s_p)") (":(np\\s_p)\\(np\\s_p)" (":((np\\s_p)\\(np\\s_p))/pf"
                "PONCT:(((np\\s_p)\\(np\\s_p))/pf)/((np\\s_p)\\(np\\s_p))" (":(np\\s_p)\\(np\\s_p)"
                  "NP-MOD:(np\\s_p)\\(np\\s_p)" "PP-MOD:((np\\s_p)\\(np\\s_p))\\((np\\s_p)\\(np\\s_p))"))
              PONCT:pf)))
        ("COORD:(np\\s_p)\\(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)" (:np\\s_p "ADV:(np\\s_p)/(np\\s_p)"
            (:np\\s_p (:np\\s_p VPP:np\\s_p "PP:(np\\s_p)\\(np\\s_p)") "NP-MOD:(np\\s_p)\\(np\\s_p)"))))))
```

Rule :
```
→ (SENT:* VPpart PP)
← (SENT:* VPpart:np\\s_p "PP:(np\\s_p)\\*")
```
Rule :
```
→ (SENT:* AP-OBJ VN)
← (SENT:* AP-OBJ:n\\n "VN:(n\\n)\\*")
```
Rule :
```
→ (PP-SUJ:* DET NP)
← (PP-SUJ:* DET:*/np NP:np)
```
Rule :
```
→ (PP-OBJ:* P AdP)
← (PP-OBJ:* "P:*/(n\\n)" AdP:n\\n)
```
Rule :
```
→ (PP-A_OBJ:* P AdP)
← (PP-A_OBJ:* "P:*/(n\\n)" AdP:n\\n)
```
Rule :
```
→ (PP-A_OBJ:* P PRO)
← (PP-A_OBJ:* P:*/np PRO:np)
```
Rule :
```
→ (PP-DE_OBJ:* P AdP)
← (PP-DE_OBJ:* "P:*/(n\\n)" AdP:n\\n)
```
Rule :
```
→ (PP:* P ADJ NC)
← (PP:* P:*/np (:np ADJ:np/n NC:n))
```
Rule :
```
→ (PP:* tree (COORD ADV CC PP))
← (PP:* PP:* (COORD:*\\* (":(*\\*)/*"
        "ADV:((*\\*)/*)/((*\\*)/*)" "CC:(*\\*)/*")
      PP:*))
```

Rule :
```
→ (PP:* tree (COORD PP))
← (PP:* PP:* (COORD:*\\* PP:*\\*))
```

Rule :
```
→ (PP:* P+D NC)
← (PP:* P+D:*/n NC:n)
```

Rule :
```
→ (PP:* P tree PP-DE_OBJ)
← (PP:* P:*/s (:s SENT:s/pp_de PP-DE_OBJ:pp_de))
```

Rule :
```
→ (PP:* P NP tree NP-SUJ)
← (PP:* P:*/s (:s NP:s/s (:s SENT:s/np NP-SUJ:np)))
```

Rule :
```
→ (PP:* P NP tree AP-ATS)
← (PP:* P:*/s
    (:s NP:s/s (:s "SENT:s/(n\\n)" AP-ATS:n\\n)))
```

Rule : use-ponct
```
→ (NP:* tree (NP-SUJ-MOD PONCT NP-SUJ PONCT))
← (NP:* NP:n (NC-MOD:n\\*
      (":(n\\*)/pf" "PONCT:((n\\*)/pf)/n" NP-SUJ:n)
      PONCT:pf))
```

Rule : use-ponct
```
→ (NP:* tree (PONCT-MOD PONCT PONCT PONCT))
← (NP:* NP:* (NC-MOD:*\\* (":(*\\*)/pf"
        "PONCT:((*\\*)/pf)/(*\\*)" PONCT:*\\*)
      PONCT:pf))
```

```
Rule : use-ponct
→ (NP:* tree (PRO-MOD PONCT PRO PONCT))
← (NP:* NP:* (Ssub-MOD:*\\*
       (":(*\\*)/pf" "PONCT:((*\\*)/pf)/np" PRO:np)
       PONCT:pf))
```

```
Rule :
→ (NP:* NC NPP)
← (NP:* NC:*/n NPP:n)
Rule :
→ (NP:* NC NP PP)
← (NP:* (:n NC:n NP:n\\n) PP:n\\*)
Rule :
→ (NP:* NC NPP NPP)
← (NP:* NC:*/np (:np NPP:np/np NPP:np))
```

```
Rule :
→ (NP:* tree ADJ (COORD CC ADJ))
← (NP:* NP:* (:*\\* ADJ:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" ADJ:*\\*)))
```

```
Rule :
→ (NP:* NPP PONCT NPP DET)
← (NP:* (:n NPP:n (:n\\n "PONCT:(n\\n)/n" NPP:n))
    DET:n\\*)
Rule :
→ (NP:* tree (COORD NP))
← (NP:* NP:n (COORD:n\\* NP:n\\*))
Rule :
→ (NP:* tree NP-OBJ-MOD)
← (NP:* NP:* NP-OBJ-MOD:*\\*)
```

```
Rule : use-ponct
→ (NP:* tree (PONCT -LBR-) PONCT Sint PONCT
    (PONCT -RBR-))
← (NP:* NP:np (:np\\* (":(np\\*)/pf"
        ("PONCT:((np\\*)/pf)/s" "-LBR-:((np\\*)/pf)/s")
        (:s (:s/gf "PONCT:(s/gf)/s" Sint:s) PONCT:gf))
      (PONCT:pf -RBR-:pf)))
```

```
Rule : use-ponct
→ (NP:* tree (PONCT -LBR-) AP (COORD CC ADV) (PONCT -RBR-))
← (NP:* NP:np (:np\\* (":(np\\*)/pf" ("PONCT:((np\\*)/pf)/(n\\n)" "-LBR-:((np\\*)/pf)/(n\\n)")
        (:n\\n AP:n\\n ("COORD:(n\\n)\\(n\\n)" "CC:((n\\n)\\(n\\n))/(n\\n)" ADV:n\\n)))
      (PONCT:pf -RBR-:pf)))
```

```
Rule :
→ (NP:* tree VPpart (COORD CC VPpart tree))
← (NP:* NP:*
    (:*\\* (":(*\\*)/*" VPpart:*\\* ("COORD:(*\\*)\\((*\\*)/*)" "CC:((*\\*)\\((*\\*)/*))/(*\\*)" VPpart:*\\*)) NP:*))
```

```
Rule : use-ponct
→ (NP:* tree (ET-MOD PONCT ET PONCT))
← (NP:* NP:* (:*\\*
       (":(*\\*)/pf" "PONCT:((*\\*)/pf)/np" ET:np)
       PONCT:pf))
```

```
Rule : use-ponct
→ (NP:* tree (NPP-MOD PONCT NPP PONCT))
← (NP:* NP:* (:*\\*
       (":(*\\*)/pf" "PONCT:((*\\*)/pf)/np" NPP:np)
       PONCT:pf))
```

```
Rule :
→ (NP-SUJ:* AP tree)
← (NP-SUJ:* AP:*/n NP:n)
Rule :
→ (NP-SUJ:* tree (COORD CC NP PP))
← (NP-SUJ:* (:* NP:*
       (COORD:*\\* "CC:(*\\*)/np" (:np NP:np PP:np\\np))))
Rule : use-ponct
→ (NP-SUJ:* PRO PONCT PP PONCT)
← (NP-SUJ:* PRO:* (:*\\* "PONCT:(*\\*)/(*\\*)"
       (:*\\* PP:*\\* "PONCT:(*\\*)\\(*\\*)")))
```

```
Rule : use-ponct
→ (NP-SUJ:* PRO PONCT VPpart PONCT)
← (NP-SUJ:* PRO:np (:np\\* "PONCT:(np\\*)/(np\\s_p)" (:np\\s_p VPpart:np\\s_p "PONCT:(np\\s_p)\\(np\\s_p)")))
```

```
Rule :
→ (NP-SUJ:* DETWH ADJ NC)
← (NP-SUJ:* DETWH:*/n (:n ADJ:n/n NC:n))
Rule :
→ (NP-SUJ:* ADVWH P NC)
← (NP-SUJ:* ADVWH:*/pp (:pp P:pp/n NC:n))
Rule :
→ (NP-OBJ:* ADV ADV P DET NC PP)
← (NP-OBJ:* ADV:*/cs (:cs ADV:cs/np
       (:np P:np/np (:np DET:np/n (:n NC:n PP:n\\n)))))
Rule :
→ (NP-OBJ:* AP tree)
← (NP-OBJ:* AP:*/n NP:n)
Rule :
→ (NP-OBJ:* tree (COORD CC PP NP))
← (NP-OBJ:* (:* NP:*
       (COORD:*\\* "CC:(*\\*)/np" (:np PP:np/np NP:np))))
```

```
Rule :
→ (NP-MOD:* ADV ADV P DET NC PP)
← (NP-MOD:* ADV:*/cs (:cs ADV:cs/np
       (:np P:np/np (:np DET:np/n (:n NC:n PP:n\\n)))))
Rule :
→ (NP-MOD:* AP tree)
← (NP-MOD:* AP:*/n NP:n)
Rule :
→ (NP-ATS:* ADV ADV P DET NC PP)
← (NP-ATS:* ADV:*/cs (:cs ADV:cs/np
       (:np P:np/np (:np DET:np/n (:n NC:n PP:n\\n)))))
Rule :
→ (NP-ATS:* NC ADJ NC)
← (NP-ATS:* NC:*/n (:n ADJ:n/n NC:n))
Rule :
→ (NP-A_OBJ:* tree)
← (NC:n NP:*)
```

```
Rule :
→ (VN:* VIMP (COORD CC VN))
← (VN:* VIMP:* (COORD:*\\* "CC:(*\\*)/*" VN:*))
Rule :
→ (VN:* CLS-OBJ tree)
← (VN:* CLS-OBJ:np VN:np\\*)
Rule :
→ (VN:* NP tree)
← (VN:* NP:np VN:np\\*)
Rule :
→ (VN:* tree ADV AP)
← (VN:* "VN:*/(n\\n)"
     (:n\\n "ADV:(n\\n)/(n\\n)" AP:n\\n))
Rule :
→ (VN:* tree AP)
← (VN:* "VN:*/(n\\n)" AP:n\\n)
Rule :
→ (VN:* V NP Ssub)
← (VN:* V:*/np (:np NP:np Ssub:np\\np))
Rule :
→ (VN:* V VPP NP-OBJ)
← (VN:* "V:*/(np\\s_p)"
     (:np\\s_p "VPP:(np\\s_p)/np" NP-OBJ:np))
Rule :
→ (VN:* V VPpart)
← (VN:* "V:*/(np\\s_p)" VPpart:np\\s_p)
Rule :
→ (VN:* VPR tree)
← (VN:* VPR:np\\s_p "VN:(np\\s_p)\\*")
Rule :
→ (Srel:* (PP P (NP PROREL tree)))
← (Srel:* P:pp/np
     (":(pp/np)\\*" "PROREL:((pp/np)\\*)/s" Srel:s))
Rule :
→ (Srel:* (PP-P_OBJ PROREL) tree)
← (Srel:* (PP-P_OBJ:*/s PROREL:*/s) SENT:s)
```

```
Rule :
→ (Srel:* CS VN NP-SUJ)
← (Srel:* CS:*/* (SENT:* VN:*/np NP-SUJ:np))
Rule :
→ (Srel:* PP-DE_OBJ VN)
← (Srel:* PP-DE_OBJ:pp_de VN:pp_de\\*)
Rule :
→ (Srel:* NP-SUJ VN NP-ATS)
← (Srel:* NP-SUJ:np (:np\\* "VN:(np\\*)/np" NP-ATS:np))
Rule :
→ (Srel:* tree AdP-MOD)
← (Srel:* SENT:* AdP-MOD:*\\*)
Rule :
→ (Srel-OBJ:* CS tree)
← (Srel-OBJ:* CS:*/* SENT:*)
Rule :
→ (VPpart-MOD:* P VN PP)
← (VPpart-MOD:* "P:*/(np\\s_p)"
     (:np\\s_p "VN:(np\\s_p)/pp" PP:pp))
Rule :
→ (VPpart-ATO:* VPP PP-P_OBJ)
← (VPpart-ATO:* VPP:*/pp PP-P_OBJ:pp)
Rule :
→ (VPpart-ATO:* ADV VPP)
← (VPpart-ATO:* ADV:*/* VPP:*)
Rule :
→ (VPpart-MOD:* VPP ADV)
← (VPpart-MOD:* VPP:* ADV:*\\*)
Rule :
→ (VPpart-MOD:* ADJ tree)
← (VPpart-MOD:* ADJ:n/n "VPpart:(n/n)\\*")
Rule :
→ (VPpart:* AP tree)
← (VPpart:* AP:*/* VPpart:*)
```

```
Rule :
→ (VPpart:* tree VPP (COORD CC VPpart))
← (VPpart:* "VPpart:*/(np\\s_p)"
     (:np\\s_p VPP:np\\s_p ("COORD:(np\\s_p)\\(np\\s_p)" "CC:((np\\s_p)\\(np\\s_p))/(np\\s_p)" VPpart:np\\s_p)))
```

```
Rule :
→ (VPinf-OBJ:* VN PP-DE_OBJ NP-MOD)
← (VPinf-OBJ:* (:* VN:*/pp_de PP-DE_OBJ:pp_de)
     NP-MOD:*\\*)
Rule :
→ (VPinf-OBJ:* VN ADV ADV)
← (VPinf-OBJ:* (:* VN:* ADV:*\\*) ADV:*\\*)
Rule :
→ (VPinf-OBJ:* VN NP-MOD ADV)
← (VPinf-OBJ:* (:* VN:* NP-MOD:*\\*) ADV:*\\*)
Rule :
→ (VPinf-OBJ:* VN AdP-MOD ADV)
← (VPinf-OBJ:* (:* VN:* AdP-MOD:*\\*) ADV:*\\*)
Rule :
→ (VPinf-SUJ:* VN NP-OBJ ADV)
← (VPinf-SUJ:* (:* VN:*/np NP-OBJ:np) ADV:*\\*)
Rule :
→ (VPinf-SUJ:* VN Ssub-OBJ ADV)
← (VPinf-SUJ:* (:* VN:*/cs Ssub-OBJ:cs) ADV:*\\*)
```

```
Rule :
→ (VPinf-OBJ:* VINF AP)
← (VPinf-OBJ:* "VINF:*/(n\\n)" AP:n\\n)
Rule :
→ (VPinf-DE_OBJ:* VN NP-OBJ)
← (VPinf-DE_OBJ:* VN:*/np NP-OBJ:np)
Rule :
→ (VPinf-ATS:* CLR VINF)
← (VPinf-ATS:* CLR:np VINF:np\\*)
Rule :
→ (VPinf-ATS:* VN ADV)
← (VPinf-ATS:* VN:* ADV:*\\*)
Rule :
→ (VPinf-OBJ:* VN VPinf-ATO ADV NP-OBJ)
← (VPinf-OBJ:* (:*/np
       (:*/np "VN:(*/np)/(np\\s_i)" VPinf-ATO:np\\s_i)
       "ADV:(*/np)\\(*/np)")
     NP-OBJ:np)
Rule :
→ (VPinf-OBJ:* VINF PP PP)
← (Vinf:* (:*/pp "VINF:(*/pp)/pp" PP:pp) PP:pp)
```

```
Rule :
→ (VPinf-ATS:* P VN AdP-MOD)
← (VPinf-ATS:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "AdP-MOD:(np\\s_i)\\(np\\s_i)"))
```

```
Rule :
→ (VPinf-ATS:* P VN VPinf-A_OBJ)
← (VPinf-ATS:* "P:*/(np\\s_i)" (:np\\s_i
      "VN:(np\\s_i)/(np\\s_i)" VPinf-A_OBJ:np\\s_i))

Rule :
→ (VPinf-A_OBJ:* P VN AdP-MOD)
← (VPinf-A_OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "AdP-MOD:(np\\s_i)\\(np\\s_i)"))
Rule :
→ (VPinf-DE_OBJ:* P VN AdP-MOD)
← (VPinf-DE_OBJ:* "P:*/(np\\s_i)" (:np\\s_i VN:np\\s_i "AdP-MOD:(np\\s_i)\\(np\\s_i)"))

Rule :
→ (VPinf-MOD:* tree AdP-MOD)
← (VPinf-MOD:* VPinf:* AdP-MOD:*\\*)

Rule :
→ (VPinf-MOD:* tree Ssub-OBJ)
← (VPinf-MOD:* VPinf:*/cs Ssub-OBJ:cs)

Rule :
→ (VPinf-MOD:* tree NP-ATS)
← (VPinf-MOD:* VPinf:*/np NP-ATS:np)

Rule : use-ponct
→ (VPinf:* tree PONCT-MOD)
← (VPinf:* VPinf:* PONCT-MOD:*\\*)

Rule :
→ (VPinf:* tree VPinf-ATS)
← (VPinf:* "VPinf:*/(np\\s_i)" VPinf-ATS:np\\s_i)

Rule :
→ (VPinf-OBJ:* tree Srel-MOD)
← (VPinf-OBJ:* SENT:* Srel-MOD:*\\*)
Rule :
→ (VPinf-ATS:* AdP-MOD tree)
← (VPinf-ATS:* AdP-MOD:*/* SENT:*)
Rule :
→ (AP-ATS:* ADJ ADJ)
← (AP-ATS:* ADJ:*/* ADJ:*)
Rule :
→ (AP-MOD:* ADJ (COORD CC ADV))
← (AP-MOD:* ADJ:* (COORD:*\\* "CC:(*\\*)/*" ADV:*))
Rule :
→ (AP-MOD:* ADJ (COORD CC tree))
← (AP-MOD:* ADJ:* (COORD:*\\* "CC:(*\\*)/*" AP:*))
Rule :
→ (AP-MOD:* ADV ADV ADJ)
← (AP-MOD:* ADV:*/* (:* "ADV:*/(n/n)" ADJ:n/n))

Rule :
→ (AP-ATS:* tree PP (COORD CC PP))
← (AP-ATS:* AP:* (:*\\* PP:*\\* ("COORD:(*\\*)\\(*\\*)" "CC:((*\\*)\\(*\\*))/(*\\*)" PP:*\\*)))

Rule :
→ (AP:* tree P NP)
← (AP:* AP:* (:*\\* "P:(*\\*)/np" NP:np))
Rule :
→ (AP-ATS:* tree VPinf-MOD)
← (AP-ATS:* AP:* VPinf-MOD:*\\*)
Rule :
→ (AP-ATS:* tree Ssub-MOD)
← (AP-ATS:* AP:* Ssub-MOD:*\\*)
Rule :
→ (AP:* tree CS)
← (AP:* "AP:*/(cs/(n/n))" "CS:cs/(n/n)")
Rule :
→ (AP-ATS:* ADJ PP (COORD CC (PP P NP) PP))
← (AP-ATS:* ADJ:*/pp (:pp PP:pp (COORD:pp\\pp
        "CC:(pp\\pp)/pp"
        (PP:pp P:pp/np (:np NP:np PP:np\\np)))))
Rule :
→ (AP:* NC AP)
← (AP:* NC:n AP:n\\*)
Rule :
→ (Ssub-ATS:* ADVWH tree)
← (Ssub-ATS:* ADVWH:*/s SENT:s)
Rule :
→ (Ssub-MOD:* ADV P CS tree)
← (Ssub-MOD:* ADV:*/* (:* P:*/* (:* CS:*/s SENT:s)))
Rule :
→ (Ssub-ATS:* CS PP)
← (SENT:s CS:*/pp PP:pp)
Rule :
→ (AdP:s CS tree)
← (AdP:s CS:s/s SENT:s)
Rule :
→ (AdP:s ADVWH tree)
← (AdP:s ADVWH:s/s SENT:s)

Rule :
→ (AdP:s ADV tree)
← (AdP:s ADV:s/s SENT:s)
Rule :
→ (AdP-MOD:* tree (COORD CC AdP))
← (AdP-MOD:* AdP:* (COORD:*\\* "CC:(*\\*)/*" AdP:*))
Rule :
→ (AdP-MOD:* PONCT AdP PONCT)
← (AdP-MOD:* (:*/pf "PONCT:(*/pf)/*" AdP:*) PONCT:pf)
Rule :
→ (COORD-MOD:* CC Srel)
← (COORD-MOD:* CC:*/s Srel:s)
Rule :
→ (SENT:* ET ET)
← (SENT:* ET:*/np ET:np)
Rule :
→ (SENT:* (AdP-SUJ DET ADV VPpart) VN)
← (SENT:* (AdP-SUJ:np DET:np/n (:n ADV:n/n VPpart:n))
    VN:np\\*)
Rule :
→ (SENT:* VPpart-SUJ VN)
← (SENT:* VPpart-SUJ:np\\s "VN:(np\\s)\\*")
Rule :
→ (SENT:* P+PRO VN)
← (SENT:* P+PRO:np VN:np\\*)
Rule :
→ (SENT:* VPinf VN)
← (SENT:* VPinf:np\\s "VN:(np\\s)\\*")
Rule :
→ (SENT:* VPP VPR)
← (SENT:* "VPP:*/(np\\s_p)" VPR:np\\s_p)
Rule :
→ (SENT:* VPP P)
← (SENT:* VPP:np\\s_p "P:(np\\s_p)\\*")
```

Rule :
→ (SENT:* VPP (COORD CC NP))
← (SENT:* VPP:np (COORD:np* "CC:(np*)/np" NP:np))
Rule :
→ (SENT:* VN VPpart-A_OBJ)
← (SENT:* "VN:*/(np\\s_p)" VPpart-A_OBJ:np\\s_p)
Rule :
→ (SENT:* VN VINF)
← (SENT:* "VN:*/(*/np)" VINF:*/np)
Rule :
→ (SENT:* VN Ssub-A_OBJ)
← (SENT:* VN:*/cs Ssub-A_OBJ:cs)
Rule :
→ (SENT:* VN PP-OBJ1)
← (SENT:* VN:*/pp PP-OBJ1:pp)
Rule :
→ (SENT:* VN NC)
← (SENT:* VN:*/n NC:n)

Rule :
→ (SENT:* VN (COORD-OBJ CC VPinf))
← (SENT:* "VN:*/(np\\s_i)" (COORD-OBJ:np\\s_i
"CC:(np\\s_i)/(np\\s_i)" VPinf:np\\s_i))
Rule :
→ (SENT:* VPP V)
← (SENT:* VPP:np\\s_p "V:(np\\s_p)*")
Rule : use-ponct
→ (SENT:* VN (COORD CC PP PONCT PP))
← (SENT:* VN:*/pp (COORD:pp CC:pp/pp
(:pp PP:pp (:pp\\pp "PONCT:(pp\\pp)/pp" PP:pp))))
Rule : use-ponct
→ (SENT:* VN (COORD CC NP PONCT NP))
← (SENT:* VN:*/np (COORD:np CC:np/np
(:np NP:np (:np\\np "PONCT:(np\\np)/np" NP:np))))
Rule :
→ (SENT:* VN (COORD CC (COORD CC VN)))
← (SENT:* VN:*
(COORD:** "CC:(**)/*" (COORD:* CC:*/* VN:*)))

Rule :
→ (SENT:* VN (COORD ADV VPinf))
← (SENT:* VN:np\\s ("COORD:(np\\s)*" "ADV:((np\\s)*)/(np\\s_i)" VPinf:np\\s_i))

Rule :
→ (SENT:* VN (COORD CC VPP AP PP))
← (SENT:* VN:* (COORD:** "CC:(**)/*" (:*
"VPP:*/(n\\n)"
(:n\\n AP:n\\n "PP:(n\\n)\\(n\\n)"))))
Rule :
→ (SENT:* VN (COORD CC VN PP-A_OBJ PP-MOD
(COORD CC PP-A_OBJ PP-MOD)))
← (SENT:* VN:* (COORD:** "CC:(**)/*" (:* VN:*/pp_a
(:pp_a (:pp_a PP-A_OBJ:pp_a PP-MOD:pp_a\\pp_a)
(COORD:pp_a\\pp_a "CC:(pp_a\\pp_a)/pp_a"
(:pp_a PP-A_OBJ:pp_a PP-MOD:pp_a\\pp_a))))))
Rule :
→ (SENT:* VN (COORD CC NP PP))
← (PP-MOD:pp_a\\pp_a VN:* (COORD:** "CC:(**)/(**)"
(:** NP:np "PP:np\\(**)")))
Rule :
→ (SENT:* VN (COORD CC AdP))
← (SENT:* VN:* (COORD:** "CC:(**)/(**)" AdP:**))
Rule :
→ (SENT:* VN (COORD CC ADV NP-SUJ VN))
← (SENT:* VN:* (COORD:** "CC:(**)/*"
(:* ADV:*/* (:* NP-SUJ:np VN:np*))))

Rule :
→ (SENT:* VN (COORD CC ADV VN))
← (SENT:* VN:*
(COORD:** "CC:(**)/*" (:* ADV:*/* VN:*)))
Rule :
→ (SENT:* VN ADJ)
← (VN:* VN:* ADJ:**)
Rule :
→ (SENT:* VINF VPP)
← (SENT:* "VINF:*/(np\\s_p)" VPP:np\\s_p)
Rule :
→ (SENT:* V VN)
← (SENT:* "V:*/(np\\s)" VN:np\\s)
Rule :
→ (SENT:* Srel VN)
← (SENT:* Srel:s VN:s*)
Rule :
→ (SENT:* PP AP)
← (SENT:* PP:pp AP:pp*)
Rule :
→ (SENT:* NC VPP)
← (SENT:* NC:n VPP:n*)

Rule :
→ (SENT:* (COORD CC CC NP) (COORD CC CC NP))
← (SENT:* (COORD:* (:*/np "CC:(*/np)/(*/np)" CC:*/np) NP:np)
(COORD:** (":(**)/np" "CC:((**)/np)/((**)/np)" "CC:(**)/np") NP:np))

Rule : use-ponct
→ (SENT:* (COORD CC ADV PONCT NP-SUJ) VN)
← (SENT:* (COORD:*/* "CC:(*/*)/(*/*)"
(:*/* ADV:*/* "PONCT:(*/*)\\(*/*)"))
(:* NP-SUJ:np VN:np*))
Rule :
→ (SENT:* (COORD CC AP) VN)
← (SENT:* (COORD:*/* "CC:(*/*)/(n\\n)" AP:n\\n) VN:*)
Rule :
→ (SENT:* (COORD CC AP PP) VN)
← (SENT:* (COORD:*/* "CC:(*/*)/(n\\n)"
(:n\\n AP:n\\n "PP:(n\\n)\\(n\\n)"))
VN:*)

Rule :
→ (SENT:* (COORD CC AdP) VN)
← (SENT:* (COORD:*/* "CC:(*/*)/(*/*)" AdP:*/*) VN:*)
Rule :
→ (SENT:* (COORD CC PP Ssub) VN)
← (SENT:* (COORD:*/* CC:*/*)
(:* PP:pp (:pp* Ssub:cs "VN:cs\\(pp*)")))
Rule :
→ (SENT:* (COORD CC PREF ADJ) VN)
← (SENT:* (COORD:n\\n "CC:(n\\n)/(n\\n)"
(:n\\n "PREF:(n\\n)/(n\\n)" ADJ:n\\n))
"VN:(n\\n)*")