Représentations sémantiques pour les Grammaires Minimalistes

Maxime Amblard

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Représentations sémantiques pour les Grammaires Minimalistes

Maxime Amblard

N° 5360

Novembre 2004

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5360--FR

Représentations sémantiques pour les Grammaires Minimalistes

Maxime Amblard *

Thème SYM — Systèmes symboliques
Projet Signes

Rapport de recherche n° 5360 — Novembre 2004 — 53 pages

Résumé : Les Grammaires Minimalistes répondent à un formalisme inspirés des travaux de la théorie générative et formalisées par Ed Stabler dans les années 90. Elles se proposent d'analyser un énoncé d'une langue naturel à partir de ses composantes lexicales, en s'appuyant sur les relations syntaxiques qui existent entre elles. La notion de minimalisme provient de la simplicité et du nombre de règles mises en oeuvre pour aboutir à l'analyse.

A partir de ces analyses, nous avons étendu ce formalisme pour obtenir une représentation logique d'ordre supérieur représentant le sens de la phrase. Pour cela, nous nous rapprochons des grammaires de Lambeck et utilisons le λ-calcul, pour donner un nouveau formalisme des Grammaires Minimalistes.

En particulier, nous utilisons une représentation arborescente dans laquelle nous introduisons des hypothèses. Une fois ces places existantes dans la dérivation, nous les substituons par leur contenu réel en temps voulu. Dans la contre partie sémantique, les hypothèses seront des λ-termes avec contextes, qui nous permettrons une nouvelle abstraction plus tard dans la dérivation afin de nous donner une représentation sémantique.

Mots-clés : sémantique, grammaires minimalistes, linguistique computationnelle, λ-calcul, théorie générative.

* doctorant bourse MENRT, Laboratoire Bordelais de Recherche en Informatique (LaBRI) & INRIA-Futurs

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,

Minimalist Grammars and Semantic

Abstract: Minimalists Grammars are a formalism inspired by work of the generative theory and formalized by ED Stabler in the Nineties. They propose to analyze a statement of a natural language, starting from its lexical components, while being based on the syntactic relations which exist between them. The concept of minimalism comes from the simplicity and the number of rules implemented to lead to the analysis.

From these analyses, we extended this formalism to obtain a logical representation of a higher level representing the semantic of the sentence. For that, we approach Lambeck Grammars and we use lambda-calcus, to give a new formalism of Grammars Minimalists.

In particular, we use an arborescent representation into which we introduce assumptions. Once these places existing in the derivation, we substitute them by their real contents at the appropriate time. In against semantic part, the assumptions will be lambda-terms with contexts, which we will allow a new abstraction later in derivation to give us a semantic representation.

Key-words: semantic, Minimalist Grammars, linguistic, λ-calcul, generative theory.

1 Introduction

Au milieu des années 90, Chomsky propose son programme minimaliste, étendant l'idée de *grammaire universelle*. Il met en avant le fait que toutes les langues ont une partie commune et ne diffèrent l'une de l'autre que par un certains nombres de principes et paramètres.

Inspiré du programme minimaliste de Chomsky, Ed Stabler propose un formalisme algébrique de grammaires pour représenter la langue, en prenant en compte l'une des principales particularités que l'on retrouve dans toutes les langues, le déplacement de certains composants. Et ce, contrairement aux autres formalismes où ils sont traités d'une autre manière.

Les grammaires minimalistes ont donc la capacité et la particularité de pouvoir analyser certaines phrases dont la construction est complexe - et l'utilisation fréquente dans la pratique de la langue. En effet, il arrive souvent qu'un énoncé donne plusieurs fonctions à un unique constituant. On le comprend aisément avec un exemple tel que "Le livre que Pierre a acheté est sur la table" où "le livre" est le sujet du verbe *être* et dans le même temps l'objet du verbe acheté. Or, le français est une langue SVO - Sujet Verbe Objet - et dans ce cas l'objet est passé devant le verbe. Il faut donc rendre compte de ce déplacement dans la structure de la phrase.

Dans un premier temps, nous présenterons de la façon la plus explicite le formalisme proposé par Stabler dans ses différents articles depuis 1997 au sujet des grammaires minimalistes. Nous consacrerons donc une première partie à exposer les définitions utiles à la présentation et à l'utilisation de ces grammaires (arbres finis ordonnés avec projection ainsi que les caractéristiques intrinsèques de cette structure).

Une fois ces définitions données, nous présenterons des exemples d'analyse syntaxique. Dans ce but nous définirons les types syntaxiques utilisés et la coindexation nécessaire, ainsi que le nouveau formalisme que nous utilisons. Ce dernier est un hybride entre les grammaires de Lambeck et les grammaires minimaliste de Stabler. Puis nous exposerons la formation du lexique qui est la base des grammaires minimalistes puisque la dérivation est entièrement dirigée par les traits des entrées lexicales.

L'objectif de ce mémoire est de fournir un formalisme pour la sémantique dans ces grammaires et c'est ce que nous proposerons dans un troisième temps. Cette couche sémantique reflète la majeure partie de mon travail. Le but est d'associer à une phrase une formule logique d'ordre supérieur à partir du champ sémantique de chacune des entrées lexicales. Afin de la mettre en évidence, nous nous appuierons sur les exemples de l'anlyse syntaxique pour exposer l'analyse sémantique.

En ce qui concerne les grammaires minimalistes, un parser a été écrit en Prolog par l'équipe d'Edward Stabler, et une version en Ocaml est en cours d'élaboration par l'un de ses étudiants John Hale. Ocaml étant le langage choisi par l'équipe signe, je me suis donc intéressé de plus près à ce dernier pour voir comment y ajouter une couche sémantique. Nous montrerons le fonctionnement général de ce parser et les divers programmes que j'ai élaboré pour faciliter le fonctionnement. Enfin, on trouvera en annexe de ce mémoire les différents programmes en Ocaml des fonctions exposées dans la dernière partie.

2 Définition des Grammaires Minimalistes

Pour définir les grammaires minimalistes, nous devons d'abord donner la structure de représentation que nous utilisons pour expliciter les relations qui éxistent entre nos différents constituants. Puis nous donner les définitions des grammaires minimalistes à proprement parler, c'est-à-dire les différents ensembles mis en jeu et les règles de composition de nos constituants.

2.1 Arbres finis ordonnés avec projection

Les arbres finis avec projection sont des arbres ayant certaines propriétés que nous utilisons pour représenter les grammaires minimalistes. Nous donnons ici le formalisme de cette structure et ses propriétés intrinsèques.

2.1.1 Arbres finis

Un arbre fini τ peut-être vu comme un ensemble de noeuds N et une relation de dominance, de telle sorte que $\tau = (N_\tau, \lhd_\tau^*)$.

Définition 1 $x \lhd y$ *signifie que "x est parent de y".*

Exemple 1 *Un petit exemple pour illustrer la relation de dominance :*

Définition 2 *La relation de dominance est raffinée par :*

$x \lhd^+ y$ *signifiant x domine strictement y,*
$x \lhd^* y$ *signifiant x domine y au sens large.*

Remarque 1 $\lhd^*$ *est la clôture transitive de* $\lhd$

Par convention, la racine d'un arbre τ est considérée comme le plus petit élément de N et les feuilles comme les plus grands. Nous pouvons alors définir L_τ l'ensemble des feuilles de τ, un sous-ensemble de N.

Définition 3 *L'ensemble de feuilles d'un arbre* τ *est l'ensemble des éléments de* N *pour lesquels il n'existe aucun autre élément tel qu'il le domine :* $L_\tau = \{x | \neg \exists y \; x \lhd y\}$

Les constituants sont choisis avec la relation de dominance.

Remarque 2 *Un sous-arbre ayant x pour racine est constitué de la partie de la structure dominée par x.*

On définit alors l'ensemble des éléments dominés par x.

Définition 4 *L'ensemble des éléments dominés par x est donné par :*

$$\uparrow x = \{y \in N | x \lhd^* y\}.$$

Remarque 3 *Une feuille se domine et ne domine qu'elle, soit* $\uparrow x = \{x\}$.

Notation 1 τ_x *est le sous-arbre ayant x pour racine :* $\tau_x = \{\uparrow x, (\lhd^* \restriction \uparrow x))\}$

2.1.2 Arbres finis ordonnés

Un arbre fini ordonné est un arbre fini auquel on ajoute une relation de précédence. Dans notre cas, cette relation est immédiate puisque nos arbres sont linéairement ordonnés de gauche à droite. On note $\prec$ cette relation et on obtient $\tau = (N_\tau, \lhd^*_\tau, \prec^*_\tau)$.

Définition 5 *De la même manière que pour les arbres finis, on obtient les raffinements :*

$x \prec y$ *signifiant x précède immédiatement y.*
$x \prec^+ y$ *signifiant x précède strictement y.*
$x \prec^* y$ *signifiant x précède y.*

Cette relation de précédence cohabite avec notre notion de domination. On obtient donc les propriétés :

Propriété 1 *Pour deux noeuds distincts, l'un domine l'autre ou l'un précède l'autre (mais jamais les deux en même temps).*

Propriété 2 *Nous assumons la précédence comme héritée au travers de la relation de dominance :*

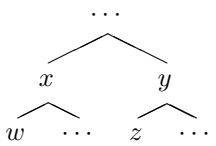
$$(\forall w, x, y, z) \quad (x \prec^* y \wedge x \lhd^* w \wedge y \lhd^* z) \rightarrow (w \prec^* z)$$

Preuve 1 *$x \prec^* y$ peut être illustré par :*

$x \lhd^* w$ peut être illustré par :

$y \lhd^* z$ peut être illustré par :

on obtient donc :



où $w \prec^* z$

2.1.3 Arbres finis ordonnés avec projection

Une dernière relation est ajoutée à notre structure d'arbre pour obtenir des objets adaptés aux grammaires minimalistes. On considère que lorsque deux constituants sont combinés, l'un d'eux se *projette* sur l'autre. Cette approche est directement inspirée du programme minimaliste de Chomsky.

Définition 6 *Le fait qu'un élément se projette sur un autre nous donne l'information de quel constituant utilise l'autre pour former le nœud résultant.*

Notation 2 *Nous adoptons la notation $<$ pour représenter le sens de la projection. Comme pour les deux premières relations, nous adoptons les définitions et les notations suivantes :*

$x < y$ *signifiant x est immédiatement projeté sur y.*
$x <^+ y$ *signifiant x est strictement projeté sur y.*
$x <^* y$ *signifiant x se projette sur y.*

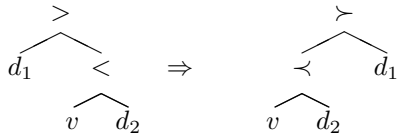
Dans la description linguistique standard des arbres étiquetés et ordonnés, l'élément minimal est au dessus des autres, avec une domination immédiate mise en évidence par des arcs et la précédence linéaire gauche-droite. Nos arbres ont une troisième relation qu'il faut marquer. Comme nous ne considérons, pour le moment, que des arbres binaires, il existe une idée simple : comme seules les feuilles sont étiquetées, il suffit d'indiquer au noeud interne quelle est le fils qui se projette sur l'autre.

Exemple 2 *Pour représenter qu'un déterminant d se projette sur un nom n pour former un groupe nominal DP, nous notons $d < n$ cette relation. Ce qui sur un arbre se présente comme suit*

Proposition 1 *Pour tout noeud x, il existe un unique fils se projetant sur les autres. Ce qui se traduit par la propriété :*

$$(\forall x)((\exists y)(x \lhd y)) \rightarrow ((\forall z)(x \lhd z \rightarrow y < z))$$

Remarque 4 *Noter que sur notre exemple, $<$ et $\prec$ coïncident. Ce qui ne sera pas le cas sur d'autres arbres comme le suivant - qui représente ce que les linguistes appellent "VP-Shell". Pour montrer la différence entre la relation de précédence et de projection, nous pouvons faire le même arbre en laissant la projection explicite et en marquant sur les nœuds la précédence :*



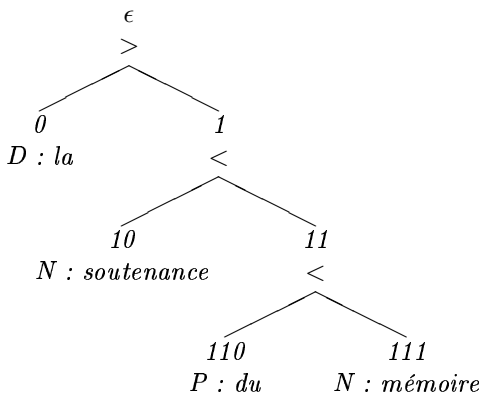
Dans ce genre de représentation, le nœud projeté est toujours sur la gauche. Nous utiliserons par convention, la représentation où la projection est marquée sur les nœuds.

En ajoutant $<^*$ dans notre structure, nous obtenons $\tau = (N_\tau, \lhd^*_\tau, \prec^*_\tau, <^*_\tau)$ qui est la structure de base utilisée pour les grammaires minimalistes.

2.2 Règles structurelles

Dans tout arbre fini ordonné avec projection, on peut définir plusieurs notions structurelles qui servent à spécifier la grammaire, que nous utiliserons fréquemment dans le formalisme des grammaires minimalistes.

Remarque 5 *Nous utiliserons l'exemple suivant pour mettre en avant les différentes définitions des règles structurelles : "la soutenance du mémoire".*



Dans ce groupe nominal :

le nœud 1 se projette par dessus le nœud 0,
le nœud 10 se projette par dessus le nœud 11,
le nœud 110 se projette par dessus le nœud 111 .

2.2.1 Tête

La tête nous est donnée par la relation de projection.

Définition 7 *D'après un arbre donné quelconque* $\tau = (\lhd_\tau^*, \prec_\tau^*, <_\tau^*)$ *et* $\forall x,\ x \in N$, *x est la tête de y ssi*

1. *y est une feuille et* $x = y$
2. *ou* $(\exists z)(y \lhd z \wedge (\forall w)(y \lhd w \rightarrow z <^* w) \wedge x$ *est la tête de* $z)$, *i.e. il existe un noeud z de parent y qui se projette par-dessus tous les enfants de y et x est tête de z.*

D'après cette définition, on obtient les propriétés intrinsèques :

Propriété 3

1. *Toute tête est une feuille, et toute feuille est sa propre tête.*
2. *Si x est tête de y alors* $y \lhd^* x$.
3. *Tout noeud a une unique tête.*

Il suffit de suivre le sens de la projection pour trouver la tête.

Exemple 3 *Si on cherche la tête du nœud ϵ, le premier nœud nous indique que la tête est vers la droite, le nœud suivant qu'elle est vers la gauche et nous arrivons sur une feuille qui est alors la tête de ϵ. De plus, 10 est aussi la tête de 1 puisque nous l'avons traversé pour trouver la tête.*
Pour le nœud 11, c'est la feuille 110 qui se projette sur 111 donc c'est la tête de 11.

2.2.2 Projection maximale

Nous définissons la projection maximale d'une tête x comme étant le plus petit noeud y ayant pour tête x.

Définition 8 *La projection maximale y de x est le noeud y tel que $\{\forall z \in N \mid$ la tête de z est $x\}$, de ce fait, il n'y a pas de nœud w dans cet ensemble, ayant la propriété : $w \lhd^+ y$.*

Notation 3 *On notera t^* la projection maximale de t.*

Exemple 4 *Le nœud 10 est la tête des nœuds 1 et ϵ. On cherche sa projection maximale, or $\epsilon \lhd 1$, ainsi ϵ est la projection maximale de la tête 10.*
De la même manière, la projection maximale de la tête 110 est le nœud 11.

2.2.3 Spécifieur

Spécifieur décrit la position d'un nœud et de sa tête par la relation de précédence qui les relie.

Définition 9 *Un noeud x un spécifieur de la tête y ssi*

1. *x est une projection maximale,*
2. *les parents de x ont pour tête y,*
3. *$x \prec^+ y$.*

Exemple 5 *D'après la définition, le nœud 0 est projection maximale de lui-même. La tête de ϵ, père de 0, est 10. Or le nœud 0 précède linéairement le nœud 1. Par transitivité, il précède tous les fils du nœud 1, donc 0 est en position de spécifieur par rapport à 10.*

2.2.4 Complément

De la même manière que *spécifieur*, nous utilisons la notion de *Complément.*

Définition 10 *Et de manière similaire, x est complément de la tête y ssi*

1. *x est une projection maximale,*
2. *les parents de x ont pour tête y,*
3. *$y \prec^+ x$.*

Exemple 6 *De la même manière, d'après la définition, le nœud 11 est une projection maximale. Ce dernier a pour parent 1 de tête le nœud 10. De plus 10 précède 11 donc le nœud 11 est en position de complément par rapport à la tête 10.*

Exemple 7 *Pour finir l'analyse des rapports entre les constituants de ce groupe nominal, nous pouvons dire que : (D : la) est un spécifieur de la tête (N : soutenance) et ((P : du)(N : mémoire)) est un complément de cette tête. ((D : la)((N : soutenance)((P : du)(N : mémoire)) est la projection maximale de la tête (N : soutenance)*

2.3 Les traits

Dans les grammaires minimalistes, il y a trois grands types de traits :

1. les traits phonétiques - la partie du mot qui se prononce.
2. les traits sémantiques - la partie qui représente le sens du mot.
3. les traits syntaxiques - utilisés pour construire la dérivation syntaxique.

Définition 11 *L'union des ensembles des traits phonétiques et sémantiques constitue un ensemble* V *représentant les traits non-syntaxiques.*

Définition 12 *Les traits syntaxiques forment un ensemble de catégories, noté* Cat *divisé en quatre sous-ensembles : base, sélecteurs, assignateurs, assignés.*

Il nous faut à présent donner les définitions de chacun de ces sous-ensembles de catégorie et leurs spécificités.

Définition 13 *Base est le sous-ensemble des traits représentant les catégories syntaxiques connues, comme v pour un verbe, n pour un nom, p pour une préposition, d pour un déterminant, c le trait acceptant d'une phrase, t pour un élément de temps.*

Définition 14 *Les sélecteurs sont les traits qui expriment une demande par rapport à une expression possédant un certain trait de base. Si α est un trait de base d'un élément, $=\alpha$ est un sélecteur : il exprime la demande d'une expression possédant le trait α.*

Remarque 6 *Dans l'ensemble des sélecteurs, il existe pour chaque trait de base x, des traits $=X$ et $X=$ dont la fonction sera explicitée plus loin.*

Définition 15 *Les assignateurs sont les traits qui assignent une propriété à certaines expressions qui sont dans une certaine relation (relation spécifieur-tête) par rapport à celles qui les portent. Si une expression porte le trait +cas, elle est prête à assigner un cas (nominatif, accusatif, obliqueÉ) à une autre expression, qui viendrait occuper une place de spécifieur par rapport à elle, prise comme tête.*

Il faudra aussi que l'expression recevant le trait assigné soit appropriée, autrement dit qu'elle demande à recevoir ce trait, ce qui se traduira par le fait qu'elle possède le trait complémentaire de +f : -f. (En l'occurrence, pour le trait +cas, son correspondant sera -cas, qui ne pourra être possédé que par des syntagmes nominaux, d'après le principe selon lequel "les syntagmes nominaux doivent nécessairement recevoir un cas" pour que la phrase soit valide).

Définition 16 *Les assignés, opposés des assignateurs, sont les traits notés -f.*

Définition 17 *Une entrée lexicale est définit comme une séquence ordonnée :*
SELECTEUR (ASSIGNATEUR) SELECTEUR* BASE ASSIGNÉ P* I**
où

P est la forme phonétique.
I est la forme sémantique.
BASE est l'ensemble des traits syntaxiques de base.
$SELECTEUR = \{= b, = B, B = | b \in BASE\}$
FUN est l'ensemble des traits de cas.
$ASSIGNÉ = \{-x | x \in FUN\}$
$ASSIGNATEUR = \{+x, +X | x \in FUN\}$

Dans certaines versions des grammaires minimalistes (Stabler, 1997), on introduit une différence entre des assignateurs forts (notés +X) et des assignateurs faibles (notés +x).

Définition 18 *Les traits fort notés avec une majuscule attirent l'intégralité des traits du sous-arbre maximal déplacé. Nous verrons par la suite ce que sont les déplacements.*

Définition 19 *Au contraire les traits faibles, notés en minuscule n'attirent que les traits non-phonétiques.*

C'est en faisant varier sur certains traits la valeur "forts/faibles" que l'on obtient des variations dans l'ordre des mots. C'est notamment ce qui permet d'analyser des langues SOV - Sujet-Objet-Verbe pour le Japonnais - ou SVO - Sujet-Verbe-Objet pour le Français. On donne alors de cette manière une réalisation de la notion de paramètre qui correspond aux particularités d'une langue naturelle.

Proposition 2 *Le langage défini par une telle grammaire est la clôture du lexique sous les fonctions génératrices (L(G) = CL(Lex, Φ)).*

Une expression sera par définition un arbre binaire ordonné fini avec projection, dont les feuilles sont des entrées lexicales étiquetées, et ce, au moyen de suites de traits.

2.4 Fonctions génératrices

Dans les grammaires minimalistes, comme nous l'avons précédemment dit, ce sont les traits qui déclenchent les opérations de construction. Pour cela, nous avons besoin de définir Φ l'ensemble des fonctions génératrices utilisées. Nous en avons deux, la fusion et le déplacement.

2.4.1 La fusion

La fusion est une opération qui réunit deux morceaux en les combinant ensemble pour en donner un troisième. Cette opération est déclenchée par la présence d'un assignateur et d'un assigné.

Définition 20 *Soit t une expression, dont la tête est T(t). Soit* t[f] *l'expression obtenue en préfixant le trait f à la suite de traits qui étiquettent T(t), soit* t(phon(α)) *l'expression t dans laquelle la concaténation des traits phonétiques est α, alors pour toutes expressions t_1, t_2 et tout $c \in Base$:*

$fusion(t_1[= c], t_2[c]) = [< t_1, t_2] si\ t_1 \in Lex.$
$fusion(t_1[= c], t_2[c]) = [> t_2, t_1] sinon.$

Nous pouvons représenter sous forme de diagramme d'arbre ces règles :

$$fusion(t_1[= c], t_2[c]) = \begin{cases} \begin{array}{c} < \\ t_1 \quad t_2 \end{array} & si\ t_1 \in Lex \\ \begin{array}{c} > \\ t_2 \quad t_1 \end{array} & \text{sinon} \end{cases}$$

Pour formaliser les règles précédentes, nous les interprétons comme des règles de déduction naturelle.

$$\frac{t_1[= c] \quad t_2[c]}{[< t_1, t_2]}\, fusion[t_1 \in Lex]$$
$$\frac{t_1[= c] \quad t_2[c]}{[< t_2, t_1]}\, fusion[t_1 \notin Lex]$$

Nous avons vu qu'il existe un autre type de trait, les traits forts. Pour une fusion avec un trait fort :

Définition 21 *Une fusion avec un trait fort se comporte comme une fusion avec un trait faible et elle fusionne les parties phonétiques en les concaténant avec l'opérateur &.*
Si $t_1 \in Lex$:

$fusion(t_1[= C], t_2[c]) = [< t_1(phon(t_1)\&phon(t_2)), t_2(phon(\epsilon))].$
$fusion(t_1[C =], t_2[c]) = [< t_1(phon(t_2)\&phon(t_1)), t_2(phon(\epsilon))].$

Sinon :

$fusion(t_1[= C], t_2[c]) = [< t_2(phon(\epsilon)), t_1(phon(t_1)\&phon(t_2))].$
$fusion(t_1[C =], t_2[c]) = [< t_2(phon(\epsilon)), t_1(phon(t_2)\&phon(t_1))].$

Nous interprétons ces règles dans la déduction naturelle :

$$\frac{t_1[=C] \quad t_2[c]}{[< t_1(phon(t_1)\&phon(t_2)), t_2(phon(\epsilon))]}\, fusion[t_1 \in Lex]$$

$$\frac{t_1[C=] \quad t_2[c]}{[< t_1(phon(t_2)\&phon(t_1)), t_2(phon(\epsilon))]}\, fusion[t_1 \in Lex]$$

Nous appellerons donc l'ensemble de ces règles la *fusion*.

Dans tous les cas de fusion, le nœud résultant reçoit le sens de la projection et les traits utilisés sont enlevés de la liste les contenant.

2.4.2 Le déplacement

Une seconde opération est utilisée. Elle est directement inspirée du programme minimaliste et met en avant l'idée de déplacement des composants à l'intérieur de la phrase.

Définition 22 *Soit t^* une expression qui est projection maximale d'une tête t. Soit, pour tout triplet d'expression t, t_1, t_2, où t contient strictement t_1 mais ne contient pas t_2, $t\{t_1/t_2\}$ le résultat obtenu en remplaçant t_1 par t_2 dans t, ainsi pour toute expression $t_1[+f]$ contenant un seul arbre maximal $t_2[-f]^*$:*

$$déplace(t_1[+f]) = [> t_2^*, t_1\{t_2[-f]^*/\lambda\}]$$

où λ est l'arbre réduit à un seul nœud, vide.

L'opération de déplacement est appliquée à la projection maximale du sous-arbre ayant le trait -f. Après l'extraction du sous-arbre de l'arbre principal le sous-arbre est déplacé en position de spécifieur de la tête de l'arbre. Les deux traits ayant déclenché le déplacement sont alors abandonnés.

Définition 23 *Si on travaille dans une version des GM qui possède la distinction entre assignateurs forts et assignateurs faibles, alors on doit raffiner cette définition par déplacement fort et déplacement faible :*

$$déplace(t_1[+F]) = [> t_2^*, t_1\{t_2[-f]^*/\lambda\}]$$
$$déplace(t_1[+f]) = [> t_2(phon(\epsilon))^*, t_1\{t_2[-f]^*/t_2'^*\}]$$
où t_2' est l'arbre t_2* avec les traits non-phonétiques supprimés.

Nous exprimons également ces règles comme des règles de déduction naturelle, également appelées règles d'éliminations de traits. La différence réside dans le lieu où la phonétique se trouve, ce qui permet de véritablement faire passer la phonologie de certains éléments devant d'autres.

$$\frac{\dfrac{t_1[+F](t_2[-f]^*)}{[< t_2^*, t_1\{t_2[-f]^*\lambda\}]}\, déplacement}{[< t_2(phon(\epsilon))^*, t_1\{t_2[-f]^* t_2'\}]}\, déplacement$$

Où la notation $t(t'[-f]^*)$ signifie "arbre t contenant un sous-arbre maximal ayant le trait -f". On ajoute une deuxième clause au déplacement pour le cas où il y a présence d'une deuxième demande de cas - ce qui se présentera en particulier pour les questions comme nous le verrons par la suite. Pour cela, nous introduirons un deuxième déplacement, ce qui ne change pas la forme de la règle appliquée. Un exemple d'utilisation sera donné plus loin.

Avec ces notations, on peut représenter le processus d'engendrement d'une phrase de manière arborescente avec des pas de fusion et des pas de déplacement.

2.5 Grammaire minimaliste

Comme les grammaires catégorielles, les grammaires minimalistes sont dirigées par les traits des entrées lexicales. Dans son article, *Derivational minimalism*, Stabler établit le formalisme des grammaires minimalistes qui réalisent le programme minimaliste de Chomsky.

Adaptée du formalisme de Stabler, une grammaire est définie comme la spécification d'un lexique et la définition des fonctions de composition des éléments qui construisent des expressions complexes à partir des éléments du lexique.

Définition 24 *Une* grammaire minimaliste *est définie par la donnée d'un quadruplet* $< V,$ *Cat, Lex,* $\Phi >$ *où :*
$V = \{P \cup I\}$ *l'ensemble fini des traits non-syntaxiques.*
$Cat = \{base \cup sélecteur \cup assignateur \cup assigné\}$ *l'ensemble fini des traits syntaxiques.*
Lex est l'ensemble d'expressions construites à partir de V *et de* Cat.
$\Phi = \{déplacement \cup fusion\}$ *l'ensemble des fonctions génératrices.*

A chaque étape de l'analyse, les traits déclenchant la fusion ou le mouvement sont alors enlevés des composants les portant, c'est pour cela que nos arbres seront progressivement débarrassés de leurs traits. Dans cette version, les expressions seront des arbres ordonnés, finis avec projection.

3 Analyse syntaxique

Pour l'analyse syntaxique, nous présenterons les types syntaxiques, et la coindexation sur ces types, puis les règles de la dérivation. Les grammaires minimalistes étant dirigées par les entrées lexicales, nous montrerons le lexique utilisé. Puis nous mettrons en avant l'analyse syntaxique sur des exemples.

3.1 Types syntaxiques

On associe à chaque entrée lexicale un type syntaxique où chaque trait correspond à un trait des grammaires minimalistes.

Définition 25 *Soit Sx l'ensemble des types syntaxiques de base associés à une entrée lexicale $Sx = \{d, k, n, v\}$ où*
- *d correspond à un groupe nominal.*
- *n correspond à un nom.*
- *v correspond à un verbe.*
- *k correspond à une attribution de cas.*

Le type d'une entrée lexicale est une composition de ces types de base.

Exemple 8 *Pour réaliser l'analyse syntaxique de la phrase suivante : "Un numéro correspond à tout étudiant", nous utiliserons le lexique avec les types syntaxiques :*

un	$: (k \times d)/n$
tout	$: (k \times d)/n$
numéro	$: n$
étudiant	$: n$
correspond_à	$: (k\backslash(d\backslash n))/d$

3.2 Dérivation syntaxique

En utilisant les définitions des grammaires minimalistes, on peut construire l'analyse syntaxique d'une phrase. La dérivation est une suite d'applications de règles déclenchées par l'apparition de certains traits sur les entrées lexicales

Définition 26 *règles dans la déduction naturelle sous forme de preuve :*

élimination de $/$: $\dfrac{\Gamma \vdash x : A/B \quad \Delta \vdash y : B}{\Gamma, \Delta \vdash xy : A}\,[/e]$

élimination de $\backslash$: $\dfrac{\Delta \vdash y : B \quad \Gamma \vdash x : B\backslash A}{\Delta, \Gamma \vdash xy : A}\,[\backslash e]$

élimination de $\times$: $\dfrac{\Gamma \vdash \alpha : A \times B \quad \Delta, x : A, y : B, \Delta' \vdash \gamma : C}{\Delta, \Gamma, \Delta' \vdash \gamma[\alpha/\{x, y\}] : C}\,[\times e]$

Dans ce nouveau formalisme de grammaires minimalistes, nous appellerons *fusion* l'élimination de / ou \, et *déplacement* l'élimination de $\times$.

Cette présentation se rapproche des grammaire de Lambeck. En effet, l'opération de fusion permet de grouper les éléments avec simplement l'élimination de ou \. Ce qui est plus complexe à simuler, c'est le déplacement. Ce dernier met en relation deux position dans l'arbre de dérivation et, en fonction des caractéristiques de l'opération, place différents éléments dans ces positions. L'idée est donc d'abandonner une construction récursive et de prévoir les positions dans l'arborescence pour leur donner leur contenu réel qu'au moment on on sait quelles parties vont être ou non déplacer.

Pour cela nous utiliserons donc des hypothèses que nous nous contraindrons de remplir à un moment de la dérivation.

Remarque 7 *Nous considererons $\times$ comme non commutatif.*

Cette remarque est importante car seuls les hypothèses trouvées dans un certain ordre lors de la dérivation pourront être mise en relation par le déplacement.

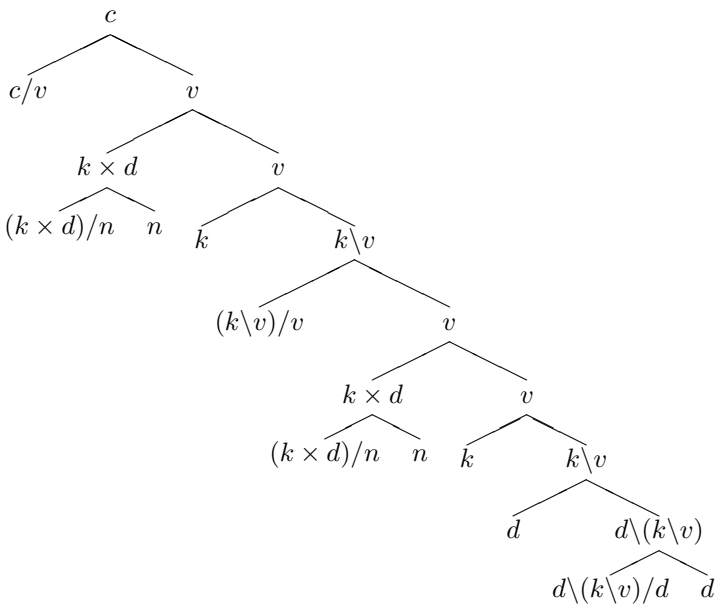
Exemple 9 *On obtient la suite de dérivations suivante :*

$$\dfrac{\dfrac{\dfrac{(k\times d)/n \quad n}{(k\times d)}[/e] \quad \dfrac{k_2 \qquad k_2\backslash v}{v}[\backslash e]}{v}[\times e] \qquad \dfrac{(k_2\backslash v)/v \qquad v}{v}[/e]}{c}[/e] \qquad \dfrac{\dfrac{(k\times d)/n \quad n}{(k\times d)}[/e] \quad \dfrac{k_1 \qquad k_1\backslash v}{v}[\backslash e]}{v}[\times e] \qquad \dfrac{\dfrac{(d_2\backslash(k_1\backslash v))/d_1 \quad d_1}{(d_2\backslash(k_1\backslash v))}[/e] \quad \dfrac{d_2 \qquad (d_2\backslash(k_1\backslash v))}{(k_1\backslash v)}[\backslash e]}{v}[/e]$$

Cette analyse peut être représentée sous la forme d'un arbre. Nous préfèrons cette représentation pour la suite de la présentation.

Définition 27 *Pour chaque utilisation des règles d'élimination on construit un arbre binaire ayant pour sommet la conclusion de la preuve, pour fils gauche la première hypothèse et pour fils droit la seconde.*

Ce qui nous donne sur notre exemple comme résultat l'arbre de type syntaxique :



Ce résultat est obtenu en plusieurs étapes comme l'arbre de preuve nous l'a montré. Nous présenterons dans une autre partie la dérivation qui nous donne cet arbre de type syntaxique.

3.3 La coindexation

Nous avons vu que l'élimination de $\times$ était une opération qui substituait deux variables à deux autres qui ont servis à les décharger de leur place dans le verbe. Dans ce cas, il faut utiliser deux variables qui agissent au même niveau.

Pour cela, nous utiliserons une coïndexation de ces deux variables. Comme le verbe et la première attribution de cas agissent sur la même constituant, nous les indexons avec le même indice. C'est pour cela que dans la suite de cette présentation, les types syntaxiques possèdent une coïndexation de certains de ses composants.

Dans ce cas, au lieu d'utiliser le type syntaxique $d\backslash(k\backslash v)/d$ pour le verbe, nous lui préfèrons $d_2\backslash(k_1\backslash v)/d_1$.

3.4 Le lexique

Le lexique est donc la base de la dérivation. Il correspond à l'ensemble des mots d'une langue avec la liste des traits associés.

un	$(k \times d)/n$	=n -case d
tout	$(k \times d)/n$	=n -case d
numéro	n	n
étudiant	n	n
correspond_à	$(k_1\backslash(d_2\backslash n))/d_1$	=d =d -case v

Dans notre lexique nous devons ajouter un certain nombre de phénomènes qui n'ont pas forcément de réalisation dans la phrase mais qui y jouent un rôle.

Notamment dans toute phrase, un verbe reçoit une inflexion. On considère que l'attribution de cas pour le sujet ne peut se faire que sur un verbe qui a reçu son inflexion pour construire une phrase bien formée. Nous utiliserons une entrée lexicale de type $(k \times v)/v$ pour représenter ce phénomène. La catégorie associée est alors =v +case v. Nous appellerons *infl* cette entrée lexicale.

Un autre cas est très fréquent et nous devons l'introduire systématiquement dans l'analyse car c'est lui qui validera la dérivation en donnant le type correspondant à une phrase : c. Le type associé est c/v, qui nous donnera lui aussi le type c acceptant pour une phrase. Nous appellerons *comp* cette entrée lexicale.

Pour l'analyse de notre phrase nous devons alors ajouter les deux entrées :

$infl$	$(k\backslash v)/v$	=v +case v
comp	c/v	=v c

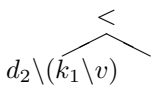
Ce genre de changement dans les types syntaxiques de la phrase est difficile à analyser. Pourtant, c'est certainement avec ce genre de trait non visible que l'on peut expliquer certains phénomènes que les linguistes mettent en avant. Par exemple, les travaux de Claire Beyssade montre que les groupes nominaux ont certains types que nous ne prenons pas en compte pour l'instant, ce qui tendrait à prouver que l'analyse même d'un groupe nominal n'est pas encore assez fine pour montrer toutes les interactions entre déterminant et nom.

3.5 L'analyse syntaxique de l'exemple

L'analyse syntaxique est présentée ci-dessous. Nous utiliserons les types syntaxiques plutôt que les traits sur des arbres similaires, c'est-à-dire des arbres finis ordonnés avec projection comme nous les avons précédemment exposés. Nous rappelons ici que les traits ou types déclenchant la fusion ou le mouvement sont enlevés des feuilles les portant.

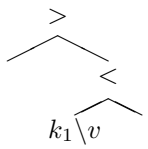
Pour commencer l'analyse syntaxique, nous devons décharger le verbe de son objet. Pour cela nous fusionnons le verbe $d_2\backslash(k_1\backslash v)/d_1$ avec un élément du type d.

étape 1 fusion :



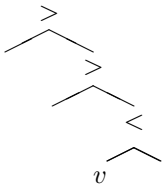
De la même manière, nous déchargeons le verbe à partir de son nouveau type $d_2\backslash(k_1\backslash v)$ en le fusionnant avec une autre variable de type d_2.

étape 2 fusion :



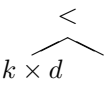
A présent, nous devons introduire une demande d'attribution de cas. Nous utilisons toujours la fusion du résultat de l'étape précédente avec un élément de type k_1.

étape 3 fusion :



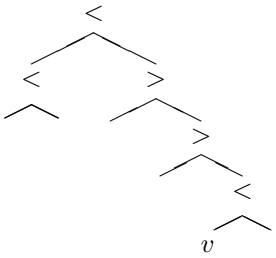
Nous devons maintenant construire l'élément qui va se substituer et qui sera l'objet de la phrase. Donc, à partir de notre dérivation, nous fusionnons le déterminant avec un nom.

étape 4 fusion :



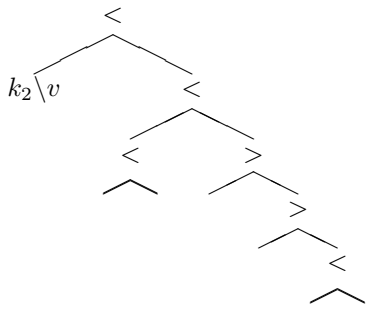
Cet objet maintenant construit doit être inséré dans la construction nous utilisons un déplacement qui relie les deux arbres.

étape 5 déplacement faible :



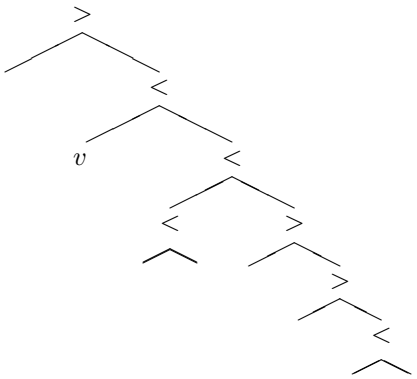
D'après la preuve de l'analyse, nous ne pouvons rien faire d'autre qu'introduire l'inflexion du verbe. C'est elle qui introduira le cas du sujet. Nous fusionnons donc notre arbre avec l'entrée lexicale $(k_2\backslash v)/v$.

étape 6 fusion :



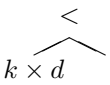
Maintenant, nous pouvons, comme pour l'objet, introduire une demande de cas mais cette fois pour le sujet de la phrase.

étape 7 fusion :



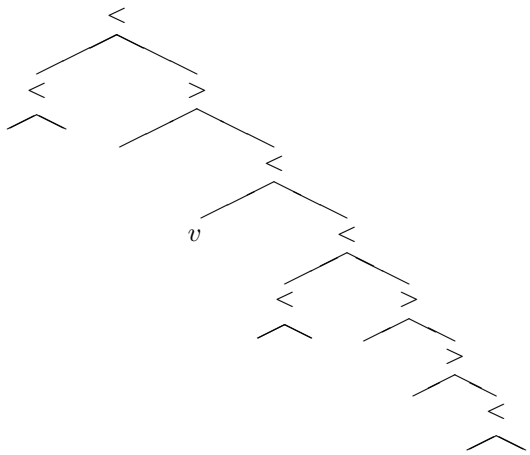
Comme pour l'objet, nous construisons à part l'élément qui forme le groupe nominal sujet en fusionnant le déterminant avec le nom, soit une fusion entre les entrées lexicales $(k \times d)/n$ et n.

étape 8 fusion :



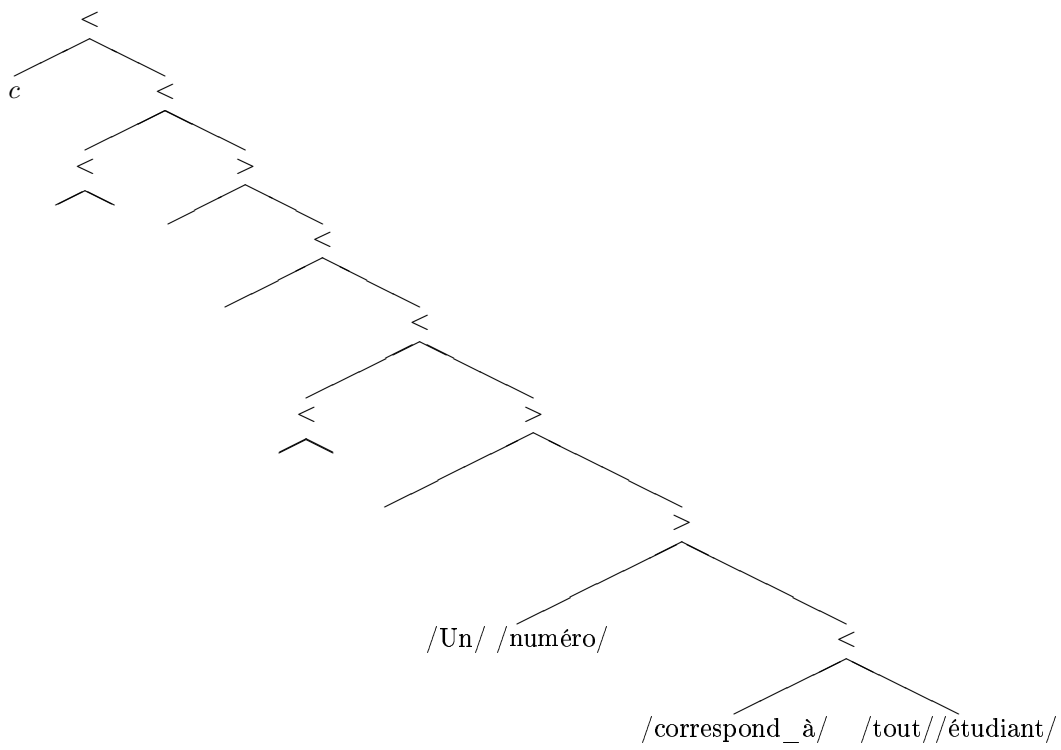
Enfin, nous substituons le sujet aux places qui ont servis à décharger les différentes composantes. Nous avons donc notre deuxième déplacement.

étape 9 déplacement :



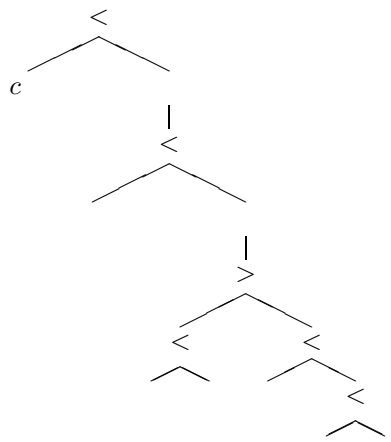
La dernière étape est une fusion avec l'entrée lexicale de *comp*, soit c/v et l'arbre précédent. Nous pouvons sur notre exemple nous pencher de plus près sur la position des mots dans l'arbre à la fin de l'analyse. Nous rappelons que le déplacement faible projette la forme phonologique sur la place d'origine. Ici, nous n'avons que des déplacements faibles, les formes phonétiques sont en position d'origine. Pour obtenir la phrase avec les mots ordonnés, on utilise l'ordre de précédence.

étape 10 fusion :



3.6 Une autre façon de présenter les choses, les arbres de dérivation

On représente une fusion par un sous-arbre binaire, un déplacement par une branche unaire et une entrée lexicale par une feuille. Cette fois on présentera l'analyse avec des arbres finis ordonnés avec projection.



Cet arbre s'obtient en commençant l'analyse par la construction de l'objet de la phrase puis en la fusionnant avec le verbe. On applique la même procédure pour le sujet. On a alors une liste de liste de traits : +case v, -case, -case. On déclenche alors un déplacement qui est le premier nœud unaire. De la même façon, on fusionne avec l'inflexion qui apporte le cas du sujet, ce qui nous permet un nouveau déplacement correspondant au sujet. Et enfin, on fusionne avec l'entrée de phrase *comp*.

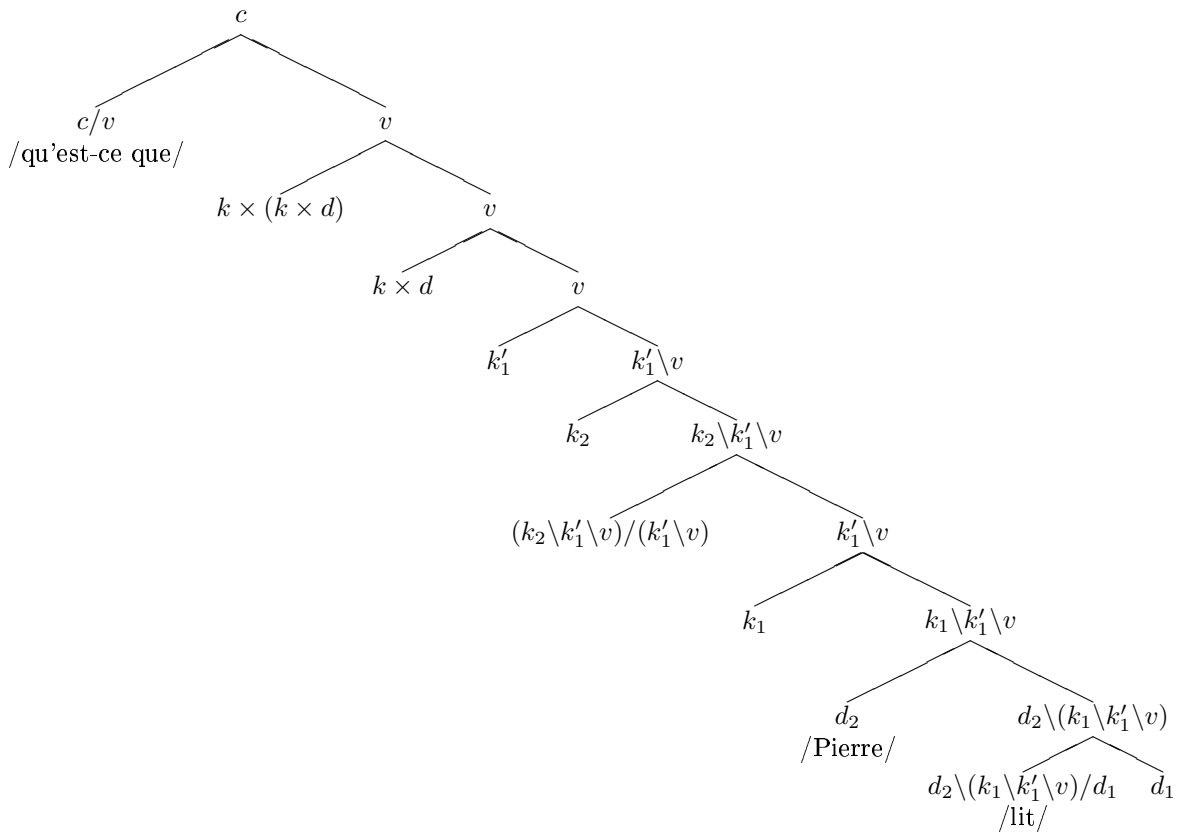
On suit donc la même procédure avec une représentation légèrement différente. En effet, en considérant que les nœuds unaires ne modifient pas le sens de la projection, ces arbres sont finis ordonnés avec projection.

3.7 Autres analyses et remarques

En effet, ce formalisme est particulièrement performant dans l'analyse d'une question où l'objet du verbe se place devant lui, et non plus après. Prenons la phrase "Qu'est-ce que Pierre lit ?". Pour l'analyser, nous utilisons le lexique :

qu'est-ce que	$(k \times (wh \times d))$	-case -wh d
Pierre	d	d
lire	$(k\backslash(d\backslash n))/d$	=d =d -case v
$infl$	$(k_2\backslash(k_1'\backslash v))/(k_1'\backslash v)$	=v +case v
comp	c/v	=v +Wh c

Ici, on est obligé de décharger le verbe deux fois pour l'attribution de cas de l'objet, et c'est la dernière opération qui permet l'introduction de ce dernier dans la phrase. De plus, le trait est fort dans *comp*, le déplacement sera donc fort. Ainsi, la forme phonétique de l'objet sera également déplacée vers le haut de l'arbre dans la position qui décharge la seconde attribution de cas.



On peut également faire l'analyse de phrase en d'autres langues. Pour le même genre de question en anglais, *" Which book does John like ?"*, l'analyse est la même. *"does"* est l'inflexion de la phrase et *"Which book"* l'objet.

Ce formalisme est également approprié pour les phrases qui contiennent des relatives, puisqu'il permet aisément de partager un élément de la phrase entre deux verbes pour lesquels il n'a pas le même rôle, par exemple pour les phrases du type : *"Le livre que Pierre a apporté est sur l'étagère du haut"*. On voit ici que *livre* est à la fois le sujet du verbe *être* et l'objet du verbe *apporter*.

On voit aussi sur cet exemple, qu'un certain nombre de raffinements peuvent être apportés à ce formalisme, en considérant que le verbe *lire* est une activité humaine, et qu'il ne peut s'appliquer que sur une famille d'objets particuliers. En effet, on lit un livre, une revue mais pas un arbre ou une main, bien qu'il soit possible de lire les lignes de la main. Cependant, comme un livre est composé de lignes, on pourrait étendre notre trait "lisible" aux lignes de la main. De la même manière que comme *Pierre* est un humain, on doit pouvoir lui associer un certain type d'activités et lui en enlever d'autres. Contrairement aux objets, *Pierre* mange, dort, joue ou danse, mais on ne met pas *Pierre* en vente, donc on ne peut pas

acheter *Pierre*, sauf bien sûr s'il s'agit d'une performance d'un artiste contemporain, mais nos analyses ne sont pas encore assez robustes pour être valides aux marges de la langue.

Une fois l'analyse syntaxique réalisée, nous voulons associer à cet arbre une analyse sémantique de la phrase. Nous nous intéresserons aux types mis en œuvre lors de la dérivation, puis ajouterons une *couche logique* pour la sémantique.

4 Sémantique dans les grammaires minimalistes

Le but est d'obtenir une forme logique d'ordre supérieur représentant le sens de la phrase. Nous avons vu qu'une dérivation minimaliste était dirigée par les traits des entrées lexicales et nous devons maintenant ajouter à chaque mot un champ représentant son sens. Pour cela nous verrons qu'il faut d'abord attribuer un type sémantique à partir des types syntaxiques, puis donner les règles de composition depuis l'analyse syntaxique qui nous donnera une représentation sémantique. Pour ajouter cette couche sémantique aux grammaires minimalistes, nous attribuerons un champ représentant cette sémantique pour chaque entrée lexicale de manière à conserver les bons types, et donner les règles utilisées en rapport avec la dérivation syntaxique.

4.1 Types sémantiques

Tout d'abord, nos entrées lexicales possèdent un type syntaxique que nous voulons transformer pour y associer un λ-terme représentant le sens logique de ce mot. Nous définirons un homomorphisme des types syntaxiques vers les types sémantiques. Définissons tout d'abord le type sémantique.

Notation 4 *Soit Se l'ensemble des types sémantiques tel que Se={e,t,$\emptyset$} où :*
- *e est un individu.*
- *t une valeur de vérité.*
- *$\emptyset$ un élément de type quelconque ou le type vide.*

Le type $\emptyset$ ne peut pas être utilisé tel quel dans la dérivation sémantique. Nous reviendrons plus en détail sur ce type plus tard. Nous pouvons donc expliciter notre homomorphisme.

Définition 28 *Soit l'*homomorphisme de type $H : Sx \longrightarrow Se$ *tel que*

$H(d) = e$, $H(n) = (e \rightarrow t)$, $H(v) = t$

$H(k) = \emptyset$, *où $\emptyset$ n'a pas de type s'il est appliqué et un type polymorphe s'il est composé avec $\times$ ou seul.*

$H(b \setminus a) = H(a / b) = (H(b) \rightarrow H(a))$

$H(a \times b) = (H(a) \times H(b))$

Exemple 10 *Grâce à notre homomorphisme, nous pouvons obtenir le type sémantique pour chaque élément de notre lexique :*

un	$: (e \rightarrow t) \rightarrow (\emptyset \times e)$
tout	$: (e \rightarrow t) \rightarrow (\emptyset \times e)$
numéro	$: e \rightarrow t$
étudiant	$: e \rightarrow t$
correspond_à	$: e \rightarrow e \rightarrow t$

4.2 Extension du lexique à la sémantique

Chaque entrée lexicale possède un type sémantique auquel nous devons associer un terme représentant son sens. Le problème est de faire coïncider ces termes avec les types sémantiques.

4.2.1 λ-termes pour la sémantique

Pour obtenir une représentation sémantique, nous utilisons des λ-termes pour leurs propriétés propositionnelles. A chacune des entrées lexicales nous associons un λ-terme. Cette procédure est classique pour la sémantique.

Notation 5

Les variables en minuscules représentent des individus. $x : e$

Les variables en majuscules représentent des prédicats. $P : (e \rightarrow t)$

Les variables en lettres grecques représentent des booléens. $\alpha : t$

Définition 29 *Nous obtenons donc pour les principaux éléments d'une phrase et leur analyse linguistique les représentations :*

- *un nom est un prédicat qui associe une valeur de vérité à une entité individuelle. On notera* $\lambda x.$*(/forme phonétique/* x*) où la forme phonétique est celle du nom, de type* $(e \rightarrow t)$
- *un verbe est aussi un prédicat auquel il faut associer deux éléments dont l'un est le sujet, l'autre l'objet de la phrase.*
  On notera $\lambda x \lambda y.$*(/forme phonétique/* y x*), de type :* $(e \rightarrow e \rightarrow t)$
- *un déterminant est un quantificateur qui associe deux prédicats pour donner une valeur de vérité. On notera* $\lambda P \lambda Q.$ *quantificateur x. (Px) relation (Qx), de type* $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$

Exemple 11 *On obtient sur notre lexique :*

un	$: \lambda P \lambda Q. \exists z(Pz) \wedge (Qz)$
tout	$: \lambda P \lambda Q. \forall z(Pz) \Rightarrow (Qz)$
numéro	$: \lambda n.(\textit{numéro}\ n)$
étudiant	$: \lambda n.(\textit{étudiant}\ n)$
correspond_à	$: \lambda x \lambda y.(\textit{correspond\_à}\ y\ x)$
inf	$: \lambda P.\textit{présent}(P)$ *pour l'inflexion.*
comp	$: \lambda \alpha.\alpha$ *pour la dernière étape de l'analyse qui correspond à* comp.

De façon évidente, nous observons que les types de nos λ-termes ne correspondent pas aux types sémantiques :

- les λ-termes ne construisent pas d'élément de type $k \times d$.
- nous n'avons rien défini pour les éléments de type k.

Nous allons maintenant voir comment modifier ces λ-termes afin d'obtenir des λ-termes pour la sémantique de grammaires minimalistes.

4.2.2 λ-termes contextués

Les λ-termes utilisés possèdent en général un type correspondant au type sémantique. Mais nous savons que l'analyse syntaxique décharge systématiquement l'objet et le sujet du verbe pour les remplacer par l'élimination de $\times$ dans la suite de la dérivation. Nous devrons donc décharger le verbe d'un de ses composants pour le substituer à un autre.

Pour cela, nous introduisons un contexte à nos λ-termes qui nous permettra de réintroduire une variable précédemment utilisée par une abstraction sur l'élément correspondant.

Définition 30 *Un* λ-terme contextué *est un λ-terme dont une partie des variables déclarées est placée à gauche du symbole* $\vdash$.

Propriété 4 *Les éléments dans le contexte d'un λ-terme contextué n'apparaissent pas dans le type sémantique du λ-terme.*

Les règles de composition que nous verrons plus loin conservent le contexte, il apparaît donc sur tous les noeuds de l'arbre de la dérivation sémantique. Cependant, il est généralement vide.

Pour notre dérivation, nous commençons par décharger le verbe de son objet et de son sujet. Pour réaliser cette opération nous avons besoin d'un type particulier de variables, les variables de contexte.

Définition 31 *Une variable de contexte est une variable pour un λ-terme doublée dans le contexte. On la note par exemple : $u \vdash u$ pour u de type e.*

Propriété 5 *Le type sémantique des variables dans le contexte n'apparaît pas dans le type sémantique de la formule. Ainsi le type sémantique de $u \vdash u$ est e*

Cependant, l'apparition du contexte nous contraint à ajouter une condition pour l'acceptation d'une formule logique représentant le sens d'une phrase. En effet, si à la fin d'une dérivation il existe encore une variable de contexte, la phrase est mal-formée et l'analyse de cette dernière doit être refusée.

Le contexte nous permet une nouvelle opération. Nous pouvons sortir une variable du contexte. Cette opération est l'équivalent d'une abstraction mais elle a la particularité d'abstraire la bonne variable dans la formule.

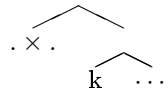
Propriété 6 *On peut sortir une variable du contexte par une abstraction dans le λ-terme. Ainsi $u \vdash u$ devint $\vdash \lambda u.u$.*

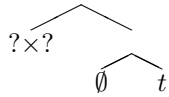
4.2.3 Rôle sémantique de l'attribution de cas

Ce qui correspond à une attribution de cas dans l'analyse syntaxique est une opération sans réalisation physique dans le corpus, comme l'inflexion, mais il y a là un échange d'informations d'un autre type que dans la simple construction d'un groupe nominal.

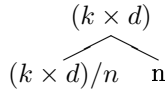
En effet, la position d'attribution de cas est le lieu de réception de la forme sémantique du groupe nominal précédemment déchargé par une variable de contexte, et qui n'a pas encore été construit (le sujet, l'objet, ...). Donc le nœud correspondant à k n'a pas à proprement parler de type sémantique prédéfini. Ce qui nous empêche de construire un homomorphisme entre type syntaxique et type sémantique. Nous considérons le type d'un tel nœud vide ($\emptyset$).

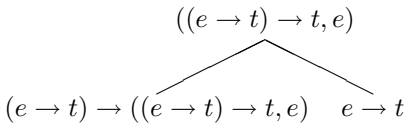
Par contre, rencontrer un nœud ayant le type vide nous apporte certaines informations. D'abord la dérivation doit être *gelée* jusqu'à que ce nœud reçoive un type. Pour qu'un tel nœud reçoive un type, il faut qu'un autre élément soit construit et se projette sur lui. C'est ce qu'il se passe avec la construction d'un élément de type $k \times d$ et son utilisation via la règle d'élimination de $\times$. En effet, chaque attribution de cas dans la dérivation syntaxique est immédiatement suivie d'une élimination de $\times$. On se retrouve dans la situation :

syntaxe 

$\Rightarrow$ sémantique 

construction d'un élément de type $\times$: nous avons vu qu'un élément composé de $\times$ était pour

la syntaxe 

et la sémantique 

projection d'un élément de type $\times$.

Définition 32 *En effet, $\times$ est un couple, nous définissons deux projections pour récupérer chacun des composants du couple.*

$$\pi_1(x_1 \times x_2) = x_1$$
$$\pi_2(x_1 \times x_2) = x_2$$

Nous verrons dans la formalisation des règles pour la sémantique comment s'utilise l'élimination de $\times$. Cependant, les déterminants représentés par des quantificateurs n'ont pas un type similaire entre syntaxe et sémantique. Au niveau de la syntaxe, ils sont représentés par un couple, ce qui n'est pas le cas pour l'instant. Nous rappelons le type d'un groupe nominal : $((e \to t) \to t, e)$. Nous considérons donc que le quantificateur n'a qu'un de ses prédicats en surface et qu'il donne un couple composé de la forme logique entière et d'une constante représentant un individu, ce qui, composé avec un prédicat, nous donne bien un couple avec une formule logique et un élément.

Exemple 12 *On obtient donc le nouveau lexique :*

un	$\vdash \lambda P(\lambda Q.\exists z(P\ z) \wedge (Q\ z), u)$	$(e \to t) \to ((e \to t) \to t, e)$
tout	$\vdash \lambda P(\lambda Q.\forall z(P\ z) \Rightarrow (Q\ z), u)$	$(e \to t) \to ((e \to t) \to t, e)$
numéro	$\vdash \lambda n.(\mathit{numéro}\ n)$	$e \to t$
étudiant	$\vdash \lambda n.(\mathit{étudiant}\ n)$	$e \to t$
correspond_à	$\vdash \lambda x \lambda y.(\mathit{correspond\_à}\ y\ x)$	$e \to e \to t$
inf	$\vdash \lambda P.\mathit{présent}(P)$	$t \to t$
comp	$\vdash \lambda \alpha.\alpha$	$t \to t$

A présent que nos λ-termes possèdent les bons types, nous allons voir comment les composer pour obtenir une représentation sémantique.

4.3 Règles pour la sémantique

Il existe deux règles différentes pour les grammaires minimalistes : la fusion et le déplacement. Nous allons exposer pour chacune d'elles les règles appliquées aux λ-termes à partir de la dérivation syntaxique.

4.3.1 La fusion

La fusion est l'étape qui revient le plus fréquemment dans la dérivation et sa contribution sémantique est naturelle dans la construction de la représentation.

Définition 33 *À une fusion est associée une application de λ-termes et une concaténation des contextes de ces λ-termes.*

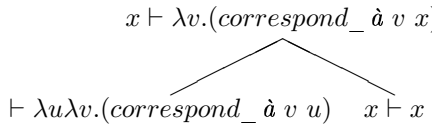
$$\frac{\Gamma \vdash x : A/B\ (F) \quad \Delta \vdash y : B\ (y)}{\Gamma, \Delta \vdash xy : A\ (F(y))}\ [/e]$$

$$\frac{\Delta \vdash y : B\ (y) \quad \Gamma \vdash x : B\backslash A\ (F)}{\Delta, \Gamma \vdash xy : A\ (F(y))}\ [\backslash e]$$

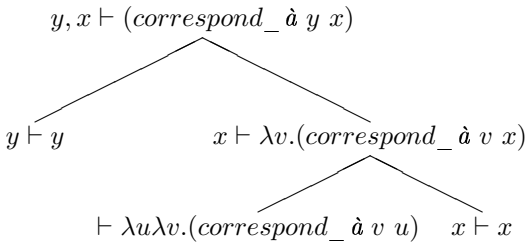
Proposition 3 *Le terme auquel on applique l'autre est toujours la tête du sous-arbre binaire.*

La fusion avec une constante de contexte se déroule suivant les mêmes règles que pour une fusion normale.

Exemple 13 *Sur notre exemple, le début de la dérivation se fait grâce à la fusion de deux termes - déchargement de l'objet du verbe par attribution d'une constante de contexte.*



Suite de la dérivation, déchargement du sujet.



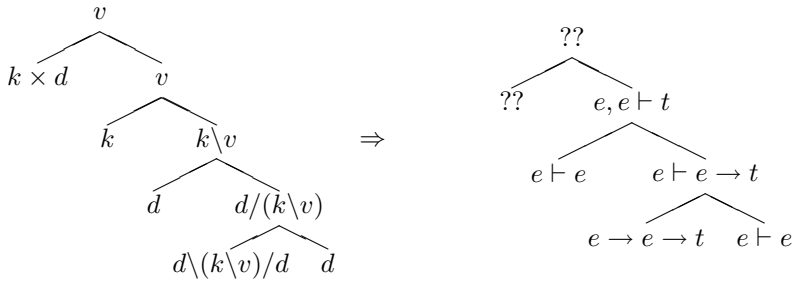
Ainsi, nous avons commencé la construction de la représentation sémantique. Le problème que nous rencontrons est qu'à l'étape suivante il faut fusionner avec une attribution de cas et comme nous l'avons exposé précédemment, nous gelons la dérivation en attendant l'élimination de $\times$, que nous allons maintenant voir.

4.3.2 Le déplacement

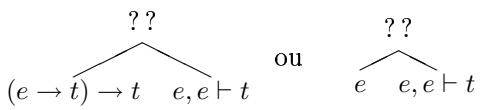
Le déplacement vient à la suite d'un gel de la construction sémantique suite à l'apparition d'un terme sans type en lieu et place de k. Du point de vue de la forme sémantique, un déplacement correspond à la première projection de l'élément $k \times d$ qui est toute la formule associée à l'élément $k \times d$ dans la position d'attribution de cas.

On utilise ici la coindexation des composants dans l'analyse syntaxique pour relier les places qui forment le couple de données.

On se trouve dans la situation suivante :



La construction a été arrêtée à la rencontre du nœud k. Le déplacement associe donc à cette place la forme sémantique de $(k \times d)$. k reçoit donc un élément de type e ou $(e \to t) \to t$ son *type raising*. A ce moment la construction peut alors reprendre. Dans l'arbre sémantique, le nœud correspondant à $k \times d$ disparaît et on se retrouve alors dans le nœud dominant l'élimination de $\times$. Reprenons au niveau de k pour lequel nous avons plusieurs possibilités :



Nous sommes confrontés à un conflit de type. Pour le résoudre nous sortons du contexte la variable qui correspond au d du couple $k \times d$.

Proposition 4 *Nous pouvons sortir une variable d'un contexte, ce qui correspond à une application sur le type sémantique, soit :* $\dfrac{e \vdash e \to t}{e, e \vdash t}$*, ce qui se traduit par une introduction de variable pour le λ-terme, i.e. une abstraction.*

On obtient :

$$\dfrac{(e \to t) \to t \quad \dfrac{e \vdash e \to t}{e, e \vdash t}}{t} \quad \text{ou} \quad \dfrac{e \quad \dfrac{e \vdash e \to t}{e, e \vdash t}}{t}$$

Cette montée de type nous permet de savoir sur quelle variable notre λ-terme va être appliqué. Une fois cette opération effectuée, l'élimination de $\times$ est terminée et on reprend la dérivation là où elle a été gelée, c'est-à-dire sur une fusion. Pour notre formule sémantique on continue la dérivation comme pour une fusion comme nous l'avons précédemment expliqué.

Exemple 14 *Dans notre exemple, nous sommes dans le premier cas. On veut que notre objet soit :*

$$\dfrac{\vdash \lambda P(\lambda Q.\forall x(P\ x) \to (Q\ x), u) \quad \vdash \lambda x.(etud\ x)}{\vdash (\lambda Q.\forall x(etud\ x) \to (Q\ x), u)}$$

On a alors :

$$\dfrac{\vdash \lambda Q.\forall x(etud\ x) \to (Q\ x) \quad \dfrac{y \vdash \lambda u.(correspond\_\ \grave{a}\ y\ u)}{y, x \vdash (correspond\_\ \grave{a}\ y\ x)}}{y \vdash \forall x(etud\ x) \to (correspond\_\ \grave{a}\ y\ x)}$$

A présent, nous pouvons formaliser l'élimination de $\times$.

Définition 34 *L'élimination de $\times$ correspond à une projection du premièr composant du couple $(k_i \times d)$ dans la position indéxé k_i dans la dérivation syntaxique.*

$$\dfrac{\Delta \vdash \alpha \times \beta\ (F, u) \quad \Gamma[\alpha_i(z), \beta_i(x)] \vdash \gamma(F')}{\Gamma[\Delta] \vdash \gamma(let\ (z, x) = (F, u)\ in\ F')}\ [\times]$$

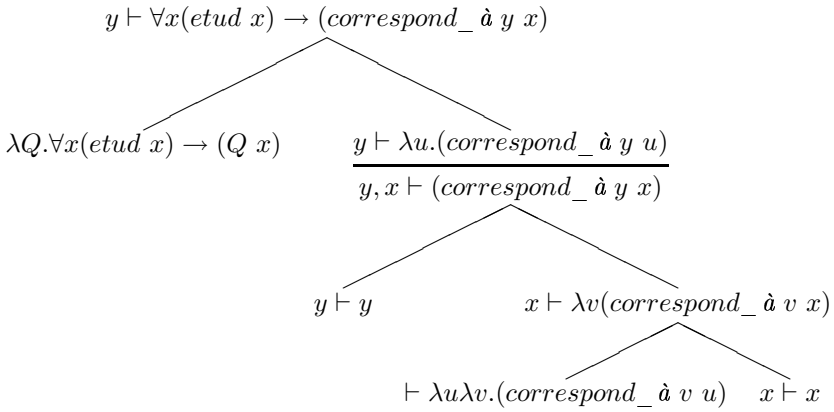
Où γ est le type de la formule F.

Ce qui permet de reprendre la dérivation sémantique au niveau de la fusion.

Remarque 8 *Le nœud de la dérivation syntaxique correspondant à l'élimination de* $\times$ *disparait dans l'arbre d'analyse sémantique.*

Le plus souvent, dans le λ-terme, le contexte est vide et l'analyse sémantique n'est acceptante que si le contexte est vide.

Exemple 15 *Nous obtenons donc sur notre exemple la dérivation sémantique suivante pour régler la première élimination de* $\times$*.*

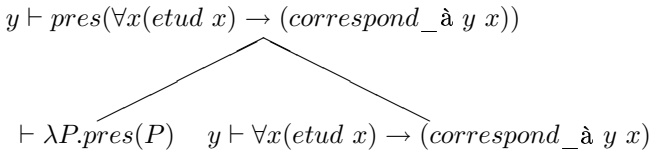


4.3.3 Suite de l'exemple, fin de la construction de la représentation sémantique

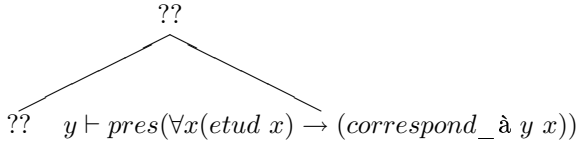
[Suite de l'exemple] Ainsi nous pouvons poursuivre la dérivation sur notre exemple : "Un numéro correspond à tout étudiant".

Nous avions, après la première élimination de $\times$, le λ-terme : $y \vdash \forall x(etud\ x) \rightarrow (correspond\_\ \grave{a}\ y\ x)$.

A présent, nous devons appliquer l'inflexion du verbe qui porte le λ-terme : $\vdash \lambda P.pres(P)$



A l'étape suivante de la construction, nous rencontrons à nouveau une attribution de cas, qui doit être associé à la variable qui à été déchargée lors de la deuxième étape d_2. Nous devons donc attendre l'élimination de $\times$ pour continuer la construction.



La construction parallèle du sujet nous donne :

$$\vdash (\lambda Q.\exists z(numero\ z) \wedge (Q\ z), u)$$

$$\vdash (\lambda P \lambda Q.\exists z(P\ z) \wedge (Q\ z), u) \quad \vdash \lambda y.(numeroy)$$

On peut donc poursuivre notre arbre en attente qui correspond au premier arbre ci-dessous par le second :

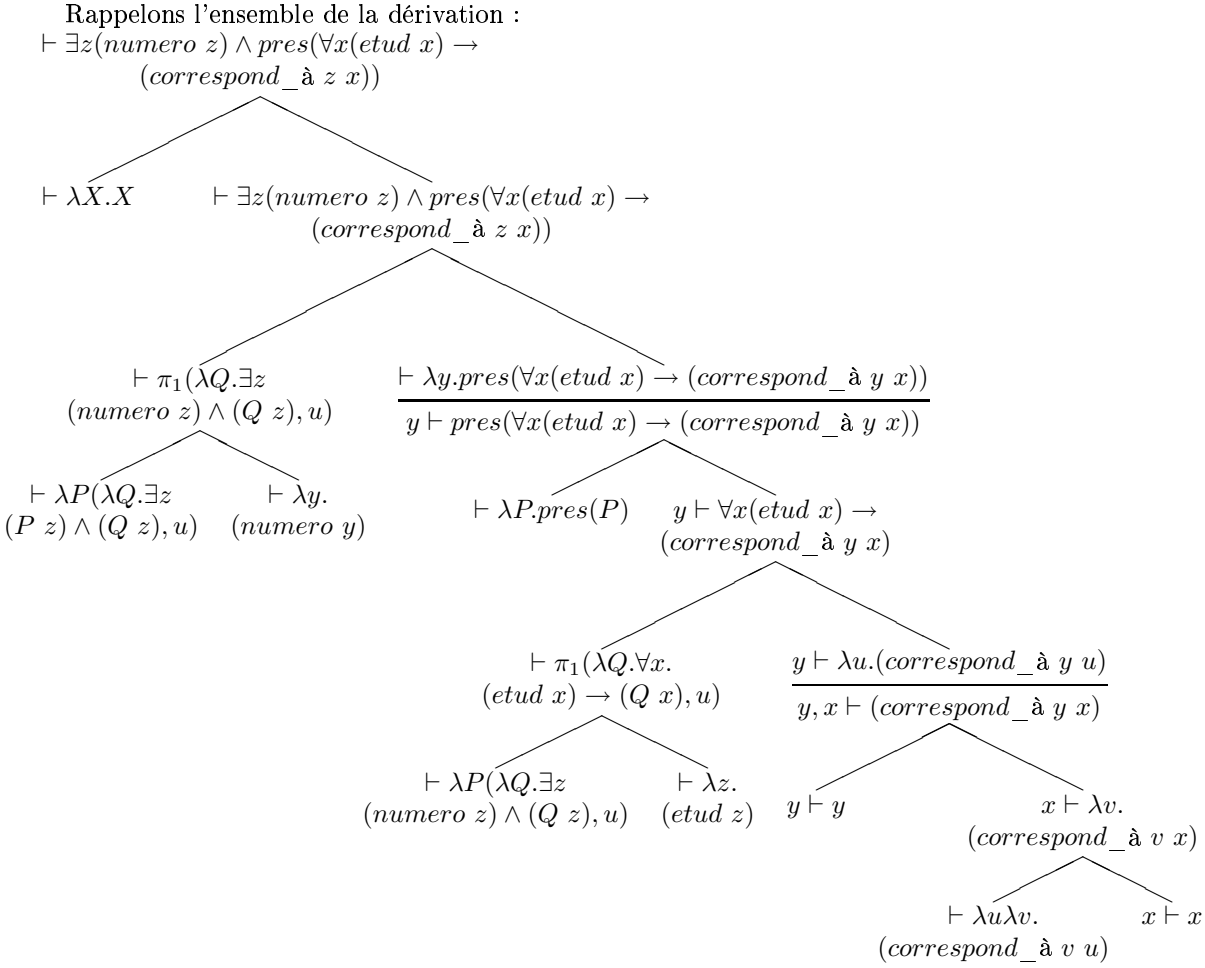
$$??$$

$$?? \qquad y \vdash pres(\forall x(etud\ x) \rightarrow (correspond\_\text{à}\ y\ x))$$

$$\vdash \exists z(numero\ z) \wedge pres(\forall x.(etud\ x) \rightarrow (correspond\_\text{à}\ z\ x))$$

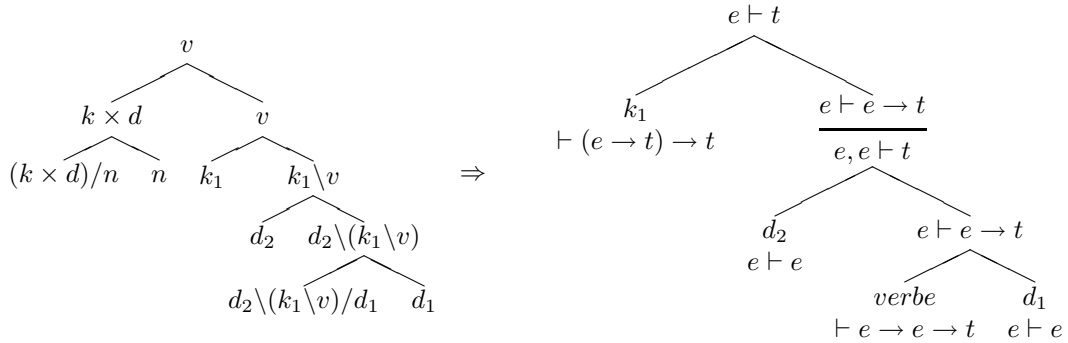
$$\vdash \lambda Q.\exists z(numero\ z) \wedge (Q\ z) \qquad \frac{\vdash \lambda y.\ pres(\forall x(etud\ x) \rightarrow (correspond\_\text{à}\ y\ x))}{y \vdash pres(\forall x(etud\ x) \rightarrow (correspond\_\text{à}\ y\ x))}$$

La dernière étape pour le *comp* n'a aucun apport sémantique, on se contente donc d'une identité qui nous permet d'obtenir la forme sémantique de notre phrase : $\vdash \exists z(numero\ z) \wedge pres(\forall x(etud\ x) \rightarrow (correspond\_\text{à}\ z\ x))$

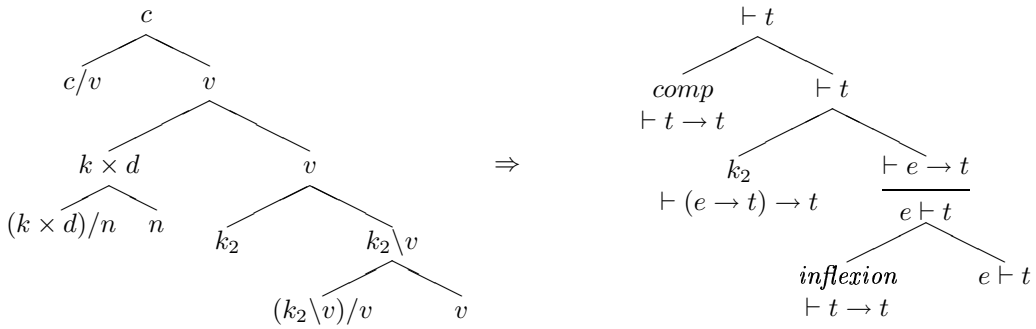


Remarque 9 *Comme nous pouvons donner la structure générale d'analyse syntaxique d'une phrase, nous pouvons la transformer en arbre d'analyse sémantique :*

Dans un premier temps :



Puis pour finir la dérivation :



4.4 Double lecture des quantificateurs, portée et déplacement

Il se pose de manière évidente un problème de portée pour la double quantification dans une phrase. Sur notre exemple nous obtenons la lecture $\exists\forall$. Cependant, il existe une seconde lecture qui donne $\forall\exists$ que nous voudrions aussi générer. Pour cela, nous devons donner une autre formule sémantique à *comp*. Il faudrait donc ajouter des traits uniquement sémantique. En effet, si on arrive à ne déplacer que la forme sémantique dans le *comp*, on appliquera la formule contenant le $\forall$ à celle contenant le $\exists$. Ainsi, on aura les deux lectures possibles pour notre phrase.

On obtient alors la formule :

$\vdash \forall x(etud\ x) \rightarrow pres(\exists z(numero\ z) \wedge (correspond\_\text{à}\ z\ x))$

Une remarque intéressante vient de Radford [RAD97]. Il expose le principe suivant lequel la structure de constituants internes aux syntagmes verbaux est complexe. Elle se divise en deux parties, le VP interne et un vp externe. Certains arguments (agent) prennent leur origine dans le vp, tandis que d'autres (thèmes) prennent leur origine dans le VP. Ainsi, la position d'origine du sujet est dans la structure profonde du verbe, donc tout ce qui porte sur le verbe, porte aussi sur le sujet. Le déplacement conserve également cette position d'origine, donc c'est la même chose pour l'objet. Dans tous les cas, l'inflexion porte sur tous

les constituants de la phrase. Le fait d'avoir le temps du verbe en début de formule peut nous permettre une concordance des temps entre les morceaux de la phrase. On obtient alors les deux lectures :

$\vdash pres(\exists z(numero\ z) \wedge \forall x(etud\ x) \rightarrow (correspond\_\text{à}\ z\ x))$

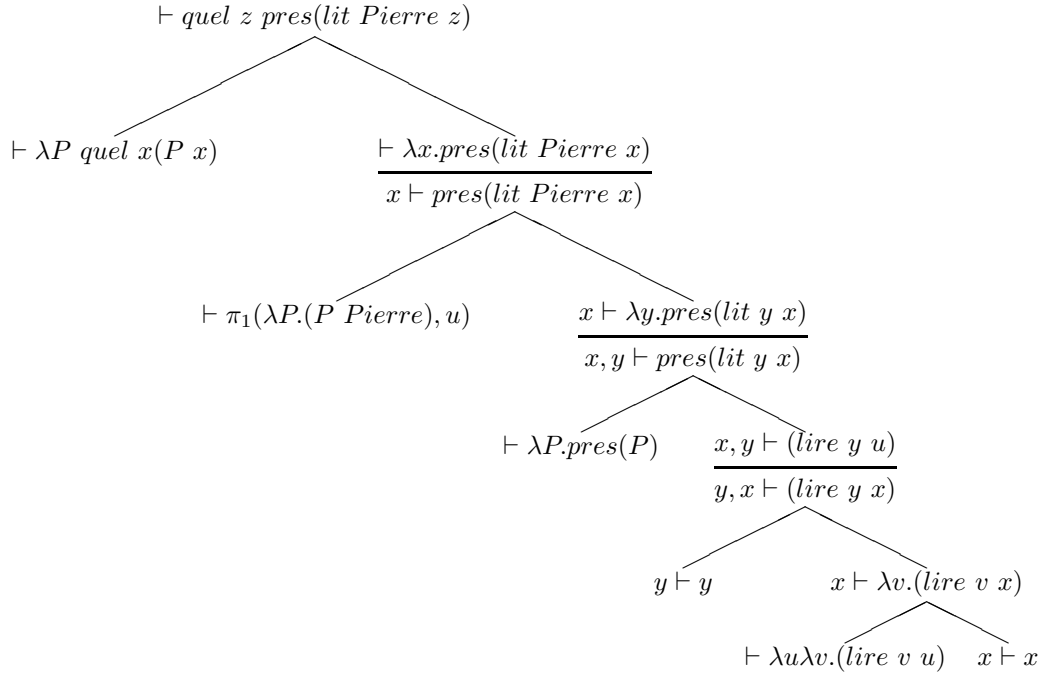
$\vdash pres(\forall x(etud\ x) \rightarrow \exists z(numero\ z) \wedge (correspond\_\text{à}\ z\ x))$

4.5 Pour les phrases interrogatives

Comme nous avons vu que nous pouvions faire l'analyse syntaxique d'une question nous devons pouvoir faire leur analyse sémantique. Pour cela, nous utilisons la deuxième clause du déplacement, qui au niveau sémantique correspond à une inversion des variables dans le contexte. Le fait qu'une entrée lexicale demande deux attributions de cas signifie qu'elle passera après les autres puisque le verbe ne peut pas lui fournir sa seconde attribution de cas, elle viendra donc d'un autre élément plus tard dans la dérivation. La formule logique que l'on obtient est :

$\vdash quel\ z\ pres(lit\ Pierre\ z)$

Nous présentons ci-dessous l'arbre de dérivation qui nous donne ce résultat :



De la même manière, on peut analyser notre phrase en anglais "Which books does John like ?" qui nous donne la formule logique :

$\vdash quel\ z\ (livre\ z) \wedge pres(aime\ Jean\ z)$

qui est la même que celle qu'on obtiendrait en français pour "Quels livres jean aime ?" - en sous-entendant que c'est aimer lire.

On voit ici l'intérêt de la forme logique qui serait une langue pivot dérivable vers d'autres langues naturelles. Ainsi, on mettrait en œuvre le multilinguisme, via les grammaires minimalistes.

5 Parser pour les grammaires minimalistes

Une première version d'un parser a été proposé par E. Stabler, en Prolog, CKY, qui a servi de noyau pour une implémentation enrichie MGCKY - Multimodal Grammar CKY. Plusieurs autres versions ont ensuite été dérivées de ce parser. Il existe une interface web développée par W. Vermaat qui permet de soumettre des fragments de phrase à MGCKY. Une autre version en C++ est développée par H. Harkema. Pour récuperer le parser, visitez la page d'accueil de MGCKY ou la page du software

http ://dhcp-190-177.let.uu.nl/mgcky/
http ://www.linguistics.ucla.edu/people/stabler/epssw.htm

Une version pour les grammaires minimalistes est en train d'être développée par J. Hale à Baltimore en Ocaml. Comme l'équipe *Signe* du LaBRI a choisi ce langage comme langage principal d'implémentation, c'est sur cette version du parser que j'ai travaillé. Je présente ici une première approche du fonctionnement général du parser tel qu'il existe pour le moment, ainsi que diverses fonctions qui offre une interaction plus aisée avec le parser et une implémentation d'une couche sémantique dans celui-ci. Les modifications qu'entrainnent l'ajout d'une couche logique dans le parser sont importante et demeure un travail à réaliser.

5.1 Présentation du parser

Nous donnons ici les grandes lignes de l'implémentation du parser pour les grammaires minimalistes avec un langage fonctionnel. Ainsi, les partie suivantes mettent en avant les types particuliers utilisés, ce qui donne une idée du fonctionnement général du parser tel que J. Hale le développe. Les perspective d'évolution se tournent à présent vers une optimisation du fonctionnement et l'ajout d'une couche sémantique.

5.1.1 Grammaires

Pour écrire la grammaire minimaliste, on utilise le même parser que dans la version prolog, ce qui permet une comparaison efficace des deux fonctions d'analyse puisqu'elles travaillent sur la même base. Ce parser est automatiquement généré par les outils Ocamlyacc et Ocamllex.

La grammaire est composée de quatre ensembles. Le premier et le plus important est le lexique, c'est à dire une liste de liste de catégories, où chaque sous liste représente une entrée lexicale. Puis deux ensembles, l'un contenant les axiomes des opérations d'adjonction sur la gauche, l'autre sur la droite. Ces opérations n'ont pas de réalisation dans la phrase, on peut dire qu'elles sont internes au déroulement de l'analyse. Et enfin, l'ensemble qui contient les catégories acceptantes pour l'analyse.

Dans les grammaires minimalistes, les entrées lexicales et les catégories sont construites sur la même base : des *listes de feature* - caractéristique. On utilise donc en ML des types énumérés, pour lesquels un item est simplement une liste de *chain*. Chacune est alors une liste de catégories annotées contenant l'information sur la portée de la chaîne d'entrée.

5.1.2 Items

A partir des entrées lexicales, on construit des élément de type *item* pour représenter chaque étape de la dérivation. On ajoute certains champs pour différencier les éléments de notre construction. Ainsi, nous aurons un type *complex* booléen qui marque la différence entre entrée lexicale et non lexicale - c'est à dire les nœuds intermédiaires de construction. D'autre part, une autre composante d'*item* est *exposed* correspondant à la séquence de traits qui reste accessible à la suite de la dérivation pour le parser.

Un autre ensemble est masqué par les définitions ci-dessus, c'est la partie non instanciéé des items. En effet, on essaie de minimiser cet ensemble pour obtenir une dérivation. On définit donc deux types pour ces derniers *flist* et *clist* qui sont des listes variables utilisées pour représenter la dérivation comme des règles de déduction des grammaires minimalistes elles-mêmes, en déduction naturelle.

5.1.3 Chart

Chart est la structure de donnée où les items dérivés sont mémorisés. Cette partie est divisée en deux grandes partie : lookup et derivation.

Look up : dans les grammaires minimalistes, l'application des règles de construction est complètement déterminée par le trait en tête de la liste. D'après les définitions ci-dessus, c'est la tête de la *exposed list*. De manière intuitive, si le parser rencontre un sélecteur, il voudra utiliser une fusion. Pour cela, il cherchera un élément ayant le type de base correspondant. C'est la même chose pour le mouvement.

Donc, Chart est indéxé par le premier élément de la liste de traits et est donc composé de :

1. un tableau de listes d'entrée.
2. une table de hashage qui recherche le tableau d'indice à partir du premier élément de la liste de traits.

Derivation : c'est l'ensemble qui mémorise la dérivation pour la restituer à l'utilisateur par la suite. La manière la plus naturelle de le fabriquer est de lui assigner l'union de l'ancienne dérivation et de la nouvelle étape trouvée par le parser. Cette approche nous donne alors un parser polynômial. Malgré cela, si le nombre de dérivations est exponentiel, comme sur une approche naïve, nous aurons un nombre exponentiel d'ensembles de dérivation.

Pour limiter ce nombre, on utilise *Backpointers*. Au lieu de stocker toute la dérivation comme un arbre ou une structure complexe, on mémorise le genre de dérivation obtenue pour une entrée donnée et un pointeur vers l'entrée précédente. Nos entrées doivent également connaître leur propre adresse et ainsi on a un ensemble de *Backpointer* pour retrouver la dérivation à partir d'une analyse acceptante et ce de manière unique.

5.1.4 Queue

L'ordre des opérations effectuées est l'un des champ des items qui retrace le résultat de la dérivation. Pour l'instant, cette opération n'est pas intégrée dans Chart. On la modélise alors par un *queue* qui dirige la stratégie d'exploration - on peut aussi la modéliser par une pile ou une queue de priorité. En effet, c'est elle qui mémorise l'ensemble de toutes les dérivations lancées en même temps et cherche à les faire aboutir dans la même direction.

Pour que l'extraction d'éléments dans la queue soit plus rapide, l'ordre est indexé par le premier trait comme pour les items. La stucture générique utilisée est *IndexedQueue* où les éléments arrivent avec une fonction de classification qui leur assigne un entier. A l'intérieur de cette queue, on utilise des pointeurs pour marquer le début et la fin.

5.1.5 Unification

L'unification permet à l'auteur des règles de déduction un grand degré de liberté. En effet, il n'est pas nécessaire de donner tous les détails de composition des items mais seulement les conditions à satisfaire. En particulier, le parser utilise l'algorithme *d'unification par transformation*, qui essaie tous les types de fusion ou de déplacement sur les items en tête de liste et n'accepte que les opérations réalisables répondant aux principes minimalistes. Ainsi, l'ensemble des items à unifier diminue jusqu'à rencontrer le symbole acceptant pour le fragment ou la phrase proposée.

A chaque pas, un item *déclenchant* une opération est sorti de la queue. Pour celui-ci, on applique toutes les opérations possibles en sortant avec Chart la liste de tous les items dont le premier trait peut se conbiner avec l'item déclenchant. On récupère l'ensemble des résultats positifs et on recommence avec la nouvelle queue.

Cette description du parser est très succincte car le fonctionnement interne est assez complexe et, pour le moment, en changement car J. Hale essaie d'intégrer la *Queue* à Chart ce qui donnera une version plus efficace. Je ne m'attache donc pas à l'actuelle. Toutefois, cette version du parser ne prend pas en compte la sémantique, il ne fonctionne et n'est écrit que pour l'analyse syntaxique.

5.2 Contributions personnelles au parser

En étudiant le fonctionnement du parser, je me suis rendu compte que la vérification du fonctionnement du parser était assez compliquée. En effet, il ne renvoie qu'un booléen, et l'ensemble de la dérivation récupéré est difficilement lisible. Il fallait une fonction d'affichage de la dérivation. De plus, pour tester la possibilité d'ajouter une couche sémantique, j'ai rédigé un nouveau parser de grammaire donnant un lexique indexé et plusieurs fonctions qui sont utiles à la manipulation d'élément dans le parser.

5.2.1 Affichage de la dérivation

Pour répondre à un réel besoin d'utilisation du parser, j'ai implémenté une fonction d'affichage qui récupère le résultat de la dérivation syntaxique dans l'ensemble *Derivation* et qui la transforme en arbre d'analyse.

L'idée est de construire un fichier dot de graphviz en "matchant" la suite de dérivation et en faisant correspondre à son évolution des nœuds d'un *arbre de dérivation* comme exposé dans la partie 3 de ce mémoire.

1. Si on rencontre un élément de type *Binary*, c'est que nous faisons une fusion entre deux éléments. Nous représentons cette opération par un sous arbre binaire ayant pour fils les deux éléments de la fusion et pour nœud le résultat de la fusion.
2. Si c'est un élément de type *Unary*, c'est que nous faisons un déplacement, et donc c'est une relation unaire dans l'arbre qui consomme le premier trait de la liste principale et un assigné dans le reste de la liste de traits.
3. Et enfin, si c'est une entrée lexicale, *Lexical*, c'est une feuille de l'arbre.

Cette analyse est encapsulée dans une fonction qui crée un fichier de type dot et qui ajoute le début et la fin du code. Cette fonction est présentée dans l'*annexe A*. Sur l'exemple suivant, *fichier.dot* est une chaîne de caractères qui sera le nom du fichier résultat et *result*, le résultat d'une analyse syntaxique.

Dans une session Ocaml :

```
# fichier "fichier.dot" result;;
```

Ce fichier doit ensuite être compiler avec dot. Dans un shell, on exécute :

```
dot -Tps fichier.dot -o fichier.ps
```

Ici le résultat est dans un fichier .ps. Le principal problème de ce parser, c'est qu'il perd certaines informations au fur et à mesure de la dérivation qui nous seraient utiles pour pouvoir afficher les autres types d'arbres. On trouvera dans l'*annexe A* un exemple d'utilisation de cette fonctioin.

5.2.2 Bibliothèque d'outils utiles

Je présente ici uniquement les fonctions qui peuvent servir à la rédaction d'un module sémantique du parser. Pour cela, on a donc besoin de pouvoir récupérer certains éléments d'une entrée lexicale en particulier sa sémantique.

Le problème est alors d'ajouter un champ position à la liste des caractéristiques d'une entrée lexicale. Pour cela, il faut mettre à jour les différentes interfaces de Feature - un champ de position appelé Positionitem - et modifier le parser qui construit la grammaire à partir d'un fichier .pl. C'est le rôle du parserprolog.mly qui génère le parser construisant la grammaire. J'ai ajouté à ce dernier un compteur qui permet la création du champ indexant le lexique.

Toutes ces modifications sont mises en avant dans l'*annexe B*. Le compteur est incrémenté à chaque construction d'une entrée lexicale et est remis à zéro à la fermeture du fichier, ainsi, chaque entrée lexicale possède un unique numéro qui sera toujours le même à chaque construstion du lexique.

De plus, voici des fonctions qui permettent de manipuler des entrées lexicales. Ici *is_phonetic* teste si le champ est ou non la partie phonétique d'un élément.

```
val is_phonetic : Feature.t -> bool = <fun>
```

Comme les entrées lexicales sont des listes de Feature.t, nous devons tester pour l'une d'elle la liste de tous ses traits pour vérifier la présence ou non de partie phonétique. C'est ce que fait *is_phonetic_in* en testant pour chacun des éléments de la liste s'il correspond à une partie phonétique.

```
val is_phonetic_in : Feature.t list -> bool = <fun>
```

De la même manière, nous pouvons renvoyer la partie phonétique plutôt qu'un booléen. C'est ce que fait la fonction *get_phonetic* via *is_phonetic*.

```
val get_phonetic : Feature.t list -> string = <fun>
```

Cependant, nous devons considérer que nos entrées lexicales sont dans un lexique qui est la composante d'une grammaire. Pour cela, il faut isoler le lexique et connaître la position de l'entrée lexicale. En donnant cette position à la fonction *phone*, elle renvoie la forme phonologique de l'éntrée lexicale :

```
val phone : int -> string = <fun>
```

On peut dériver ces fonctions pour obtenir d'autres champs d'une entrée lexicale, par exemple la sémantique. On trouvera dans l'*annexe B* le code de ces fonctions.

5.3 Vers une implémentation

L'implémentation du parser est en cours de réalisation. Le travail demandé a aussi été d'analyser le fonctionnement du parser proposé par John Hale, comme exposé ci-dessus.

On peut à partir du formalisme présenté dans ce mémoire envisager une implémentation de la sémantique. Pour cela, il faut associer à chaque entrée du lexique un nouveau champ qui sera un λ-terme. Ce qui peut être réalisé de la même façon que pour l'ajout d'un champ de position.

Ensuite, il faut alors pouvoir récupérer dans un lexique, pour chacune des entrées lexicales sa forme sémantique. Or, le parser travaille sur un sous ensemble du lexique d'origine jusqu'à obtenir une validation de la dérivation. Il faut donc lors de la construction d'un élément item à partir d'une entrée lexicale conserver la position dans le lexique. Ce qui permettra également de conserver cet indice lors de la création de la Dérivation. Ainsi, depuis item ou dérivation, on peut avoir accès aux autres champs de l'entrée lexicale. On peut alors récupérer

la forme sémantique et entammer la construction de la formule logique en appliquant les règles précédemment exposées. Ces règles étant des compositions ou des abstraction de variables sur des λ-termes.

Pour cela, certains outils parraissent particulièrement bien adaptés. En effet, la bibliothèque CCS de Gerard Huet, permet de construire en Ocaml des λ-termes et de les composer. Ce qui est l'une des étapes du protocole. Elle paraît donc un outil indispensable à l'implémentation de la sémantique. Malheureusement, toutes ces modifications demandent de réécrire une grande partie des éléments du parser et ce dernier n'est pas encore à sa version définitive.

Il nous faudra donc :

1. Ajouter d'un champ sémantique pour chaque entrée lexicale.
2. Faire passer l'indice de position dans la construction d'un *item*
3. Faire passer cet indice lors de la construction de *Derivation*.
4. Récupérer le résultat dans dérivation et l'interpréter pour la sémantique.

Si l'on veut envisager une coopération de la couche sémantique dans le parser, il faut que ce qu'il se passe pour les items soit répercuté dans une nouvelle structure représentant la sémantique. De plus il faut lui associer une fonction retournant un booléen pour valider la sémantique à la même étape que dans la construction syntaxique.

6 Conclusion

Ce rapport avait pour intention, dans un premier temps de présenter de manière simple et complète le formalisme de Stabler pour les grammaires minimalistes. Ce dernier étant assez long à mettre en œuvre. En effet, il demande de maîtriser un certain nombre de définitions, mais une fois cette étape franchie, l'utilsation du formalisme et les perspectives qu'il offre me semblent en faire un outil de premier ordre pour l'analyse d'une langue naturelle. Il répond à la fois aux perspectives linguistiques mises en avant par Chomsky autant dans les années 50 où il voulait proposer un modèle universel de formalisme pour les langues naturelles, que dans les années 90 avec son programme minimaliste où les principes régissant une langue sont en nombre limité.

Une fois cette présentation effectuée, il fallait proposer un formalisme partant de l'analyse syntaxique qui fournisse une formule logique d'ordre supérieur représentant le sens de la phrase analysée à partir d'un sens donné à chaque élément de la phrase. Cette étape peut être vue comme une solution de langage pivot pour le multilinguisme. La couche sémantique que je propose est assez complète pour s'envisager implémentable dans le parser Ocaml de John Hale.

Cette phase sera intéressante dans la validation de ce nouveau formalisme pour les grammaires minimalistes qui ouvrira véritablement vers la production d'algorithmes comme la résolution des problèmes d'anaphore posés par les pronoms - ce qui est le sujet de la thèse de Roberto Bonato au LaBRI - université de Vérone - mais aussi la définition de paramètres précis de chaque langue naturelle en vu d'obtenir des énoncés communs - sujets ouverts pour ce formalisme.

Malgré tout, la coopération entre linguistes et informaticiens montre qu'actuellement ce formalisme ne répond pas à tous les phénomènes que ces premiers exposent. En effet, et comme je l'ai relaté dans ce rapport, certains linguistes remettent en cause les types que nous utilisons comme acquis, ce qui tendrait à prouver que des opérations internes à la structure ne sont pas encore comprises car sans représentation physique dans la phrase.

D'autre part, nous savons qu'un certain nombre de traits qui pourraient donner des informations capitales pour la réfutation d'énoncés ne sont pas pris en compte tels que les traits humains/non-humains, etc. Malheureusement, la performance du parser dépendant du nombre de traits utilisés, nous ne pouvons par conséquent pas faire croître de manière inconsidérée ce nombre. Une solution intermédiaire est à envisager.

Cependant, que ce soit pour les informaticiens et la production d'algorithmes ou pour les linguistes qui utiliseront ces algorithmes d'analyse, notre nouvelle priorité est de passer à la phase d'implémentation de la couche sémantique.

Il faut enfin mettre en avant la performance des grammaires minimalistes qui permettent l'analyse d'énoncés complexes que d'autres formalismes n'arrivent pas à intégrer, tout en tentant de minimiser la taille du lexique comme les principes mis en œuvre. En outre, les grammaires minimalistes ne sont peut-être pas encore la dernière étape mais sûrement un

intermédiaire vers un formalisme plus général qui reprendra leurs avantages en gommant leurs inconvénients.

Enfin, si ce sujet m'a particulièrement intéressé c'est parce qu'il se situe au confluent de deux disciplines qui ouvrent des perspectives de recherches prometteuses pour la compréhension et l'utilisation des langues naturelles.

Remerciement

Je souhaite naturellement remercier Pr Christian Retoré et Pr Alain Lecomte pour leur encadrement lors de mes travaux, ainsi qu'Hélène pour son soutient et tous les re-lecteurs.

Références

[BetM76] Bondy et Murphy : *Graph Theory with Application*, the MacMillan press limited, 1976.

[CMP01] Chailloux E., Manoury P., Pagano B. : *Développement d'applications avec Ocaml*, O'Reilly, 2001.

[Chom95] Chomsky, Noam : *The Minimalist Program*, MIT Press, Cambridge, MA, 1995.

[Chom98] Chomsky, Noam : *Minimalist Enquiries : the framework*, MIT, draft, 1998.

[Dow01] Dowek, G : *Higher-Order Unufication and Matching*, Elsevier Science Publishers, Handbook of automated reasoning, 2001.

[HetK98] Heim, Irène & Kratzer, Angelika : *Semantics in Generative Grammar*, Blackwell, 1998.

[Huet75] Huet, Gérard : *A Unification Algorithm for Typed λ-Calculus*, Journal of Theoretical Computer Science, 1 :27–57, 1975.

[Huet02] Huet, Gérard : *The Zen Computational Linguistics Toolkit*,ESSLLI 2002 Lecture, 2002.

[Jack02] Jackendoff, Ray : *Foundations of Language*, Oxford University Press, 2002.

[Lec02] Lecomte, Alain : *Rebuilding MP on a logical ground*, Research on Language and Computation 1(3), 2002.

[LetR99] Lecomte, Alain & Retoré, Christian : *Towards a Logic for Minimalist*, in G. van Kruijf Formal Grammar'99, ESSLLI 99, Utrecht, 1999.

[LetR01] Lecomte, Alain & Retoré, Christian : *Extending Lambek Grammars*, in Proceedings of ACL 2001, Toulouse, pp. 354-361, 2001.

[Pol97] Pollock, Jean-Yves. : *Langage et Cognition*, Presses universitaires de France, 1997.

[Rad97] Radford, A : *Syntactic theory and the structure of English*, Cambridge textbooks in linguistics, 1997.

[RetS03] Retoré,Christian & Stabler, Edward : *Generative Grammars in Resource Logics*, Research on Language and Computation 1(3), 2003.

[Stab97] Stabler, Edward : *Derivational Minimalism*, in C. Retoré, editor, "Logical Aspects of Computational Linguistics LACL'96", volume 1328 du LNCS/LNAI, Springer-Verlag, 1997.

[Stab99] Stabler, Edward : *Remnant Movement and Complexity*, in Bouma, Hinrichs, Kruijf et Oehrle Constraints and Resources in Natural Language Syntax and Semantics, CSLI, 1999.

[Stab01] Stabler, Edward : *Recognizing Head Movement*, in de Groote, Morrill and Retoré "Logical Aspects of Computational Linguistics", LNAI-LNCS, n 2099, Springer, 2001.

7 Annexes A - affichage de la dérivation

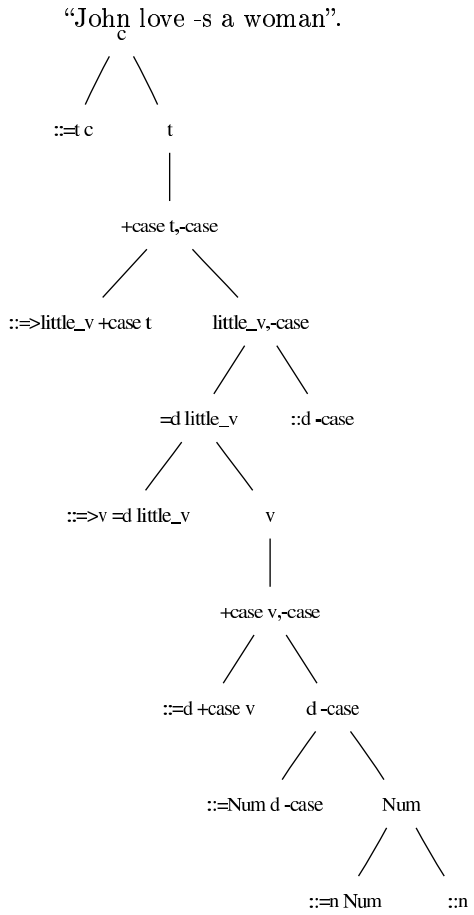
7.1 Fonctions utilisées

```
let ajout2 lk =
   let rec affichag2 te i =
   match te with
   Chart.Binary (parent,child0,child1,d0,d1) ->
         let re1 = affichag2 d0 (i+1) in
         let re2 = affichag2 d1 ((Utilities.second re1)+1)
      in
    "n"^(string_of_int i)^" [label = \""^parent^"\"]; \n n"^
     (string_of_int i)^"-> n"^(string_of_int (i+1))^"; \n"^
     (Utilities.first re1)^"\n"^(Utilities.first re2)^"\n n"^
     (string_of_int i)^"-> n"^(string_of_int
                          ((Utilities.second re1)+1))
      ^"; \n" , (max (Utilities.second re1)
                         (Utilities.second re2))

   |Chart.Unary (parent,child,d) ->
         let re = (affichag2 d (i+1)) in
     "n"^(string_of_int i)^" [label = \""^parent^"\"]; \n n"^
     (string_of_int i)^"-> n"^(string_of_int (i+1))^"; \n"^
     (Utilities.first re) , (Utilities.second re)

   |Chart.Lexical (a,b) -> "n"^(string_of_int i)^" [label =
                          \""^a^"\"];" , i in
    let rec affic3 lk i =
       match lk with
          a::ml -> let re = affichag2 a i in
     (Utilities.first re)^(affic3 ml
                     ((Utilities.second re)+1))

   |_ -> " " in
     let affich3 lk = affic3 (Chart.DerivationSet.elements
                               lk) 0 in

"digraph G {\n node [shape = plaintext]; \n edge
[arrowhead = none]; \n"^affich3 lk^"}"
;;

let fichier nom str2 =
   let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
```

```
   let fic = Unix.openfile nom mode 0o644 in
   let str = ajout2 str2 in
   let n = Unix.write fic str 0 (String.length str) in
    n ;
    Unix.close fic
;;
```

7.2 Résultat d'un affichage

Voici le résultat d'un affichage de l'analyse de la phrase :



8 Annexes B - quelques fonctions utiles

8.1 Indexation du lexique

8.1.1 Modification du parser de construction de la grammaire

Fichier ocamlyacc parsant un fichier pour construire la grammaire avec ajout de l'indexation des éléments. Ajout d'un compteur dans la partie de déclaration :

```
%{
open Feature;;
open Grammar.C185;;
let depth = ref 0;;
%}
```

Mise à zéro du compteur en fin de construction de la grammaire :

```
facts :
  entry facts { add_lexical_entry $1 $2 }
| adjunctionstmt facts { (grammar_union $1 $2) }
| startstmt facts { grammar_union $1 $2 }
| EOF { depth := 0 ; null_grammar }
;
```

Ajout d'un champ position pour chaque entrée du lexique :

```
entry :
  single COLONS featurelist PERIOD  {incr depth ; $3 @ [Phonetic (Const $1)] @
     [Positionitem (Const (string_of_int (!depth)))]; }
| single COLONS single PERIOD  {incr depth ; [Category (Const $3)] @
    [Phonetic (Const $1)] @ [Positionitem (Const (string_of_int (!depth)))];}
| empty COLONS featurelist PERIOD  { incr depth ; $3 @ [Phonetic (Const $1)] @
    [Positionitem (Const (string_of_int (!depth)))] ;}
| empty COLONS single PERIOD  { incr depth ; [Category (Const $3)] @
    [Phonetic (Const $1)] @ [Positionitem (Const (string_of_int (!depth)))] ;}
;
```

8.1.2 Modification du type Feature

Ajout d'un champ position dans le type Feature

```
    type t =
Select of name
      | Category of name
      | Attract of name
      | Licensee of name
```

```
      | Phonetic of name
      | RIncorp of name
      | LIncorp of name
      | RAffhop of name
      | LAffhop of name
      | FeatureVar of string
      | Positionitem of name

    let show = function
Select x -> "="^(show_name x)
      | Category x -> show_name x
      | Attract x -> "+"^(show_name x)
      | Licensee x -> "-"^(show_name x)
      | Phonetic x -> "/"^(show_name x)^"/"
      | RIncorp x -> (show_name x)^"<="
      | LIncorp x -> "=>"^(show_name x)
      | RAffhop x -> (show_name x)^"=>"
      | LAffhop x -> "<="^(show_name x)
      | FeatureVar x -> ("?"^x)
      | Positionitem x -> (show_name x)
```

Il faut aussi répercuter ces modifications dans la signature des foncteurs qui utilisent le type Feature.

8.2 Manipulation des éléments d'un lexique à partir de leur position indéxée

```
let is_phonetic l =
   match l with
    Feature.Phonetic k -> true
   |_ -> false
;;

let rec is_phonetic_in lk =
   match lk with
   |a::jh -> if (is_phonetic a) then true
      else (is_phonetic_in jh)
   |[] -> false
;;

let rec get_phonetic lk =
   match lk with
   |a::jh -> if (is_phonetic a) then (Feature.show a)
```

```
       else (get_phonetic jh)
   |[] -> ""
;;

let phone n =
   let grammaire =  read "tests/larsonian.pl" in
   let lexique = grammaire.Grammar.C185.lexicon in
   let test = List.nth lexique n in
   get_phonetic test
;;
```

Table des matières

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399