Université Bordeaux I
UFR Mathématiques et Informatique
Département d'Informatique

Mémoire de recherche
en vue de l'obtention du diplôme de
MASTER Sciences et Technologies
mention Informatique

# Analyse syntaxique dans les grammaires minimalistes

Benoit WAGLER
<benoit.wagler@etu.u-bordeaux1.fr>

Encadrants :
Gregory M. KOBELE,
Christian RÉTORÉ

Bordeaux, 7 juin 2007

# Table des matières


**1 Introduction** **1**

**2 Les Grammaires Minimalistes** **3**

2.1 Définitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.1.1 Généralités . . . . . . . . . . . . . . . . . . . . . . . . 3

2.1.2 Les arbres . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.3 Les traits . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1.4 Les fonctions génératrices . . . . . . . . . . . . . . . . 7

2.2 Un exemple simple . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 L'arbre de dérivation . . . . . . . . . . . . . . . . . . . . . . . 14

2.4 Reformulation des Grammaires Minimalistes . . . . . . . . . . . 16

2.4.1 Représentation sous forme de chaînes compactes . . . . . 16

2.4.2 Reformulation des règles génératrices . . . . . . . . . . . 17

**3 Description d'un algorithme** **20**

3.1 La reconnaissance par parcours en profondeur récursif . . . . . . 21

3.1.1 Les items et l'*invariant* . . . . . . . . . . . . . . . . . . 21

3.1.2 Les axiomes . . . . . . . . . . . . . . . . . . . . . . . . 22

3.1.3 Les règles d'inférence . . . . . . . . . . . . . . . . . . . 23

3.1.4 Les items but . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2 Exemple . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.3 Preuve et complexité . . . . . . . . . . . . . . . . . . . . . . . 27

3.4 Extension de l'algorithme et améliorations . . . . . . . . . . . . . 28

3.4.1 Fonction de prédiction . . . . . . . . . . . . . . . . . . . 28

3.4.2 La propriété de préfixe correct . . . . . . . . . . . . . . . 28

**4 Vers une implémentation de l'algorithme en oCaml** **30**

4.1 Forme générale du programme . . . . . . . . . . . . . . . . . . . 31

4.2 La grammaire . . . . . . . . . . 32
4.3 Les items . . . . . . . . . . 32
4.3.1 Les variables . . . . . . . . . . 33
4.4 Le tableau . . . . . . . . . . 34
4.5 les règles d'inférence . . . . . . . . . . 35
4.5.1 Cas particuliers . . . . . . . . . . 35

**5 Conclusion** **39**

**A Exemple de fonction** **44**

**Résumé**

Les grammaires minimalistes d'Edward Stabler formalisent le récent programme minimaliste de Noam Chomsky. Elles permettent de décrire les structures syntaxiques des phrases du langage naturel sous forme d'arbre. Les différentes étapes de la construction d'une telle structure peuvent être syntétisées à l'aide d'un arbre de dérivation.

Le travail que j'ai effectué pendant ce semestre a consisté à réfléchir à l'implémentation d'un analyseur pour ces grammaires minimalistes basé sur un principe de parcours récursif en profondeur des arbres de dérivation. L'algorithme de cet analyseur fonctionne à la manière d'un système de déduction, avec des axiomes et des règles d'inférence.

En suivant le paradigme de l'analyse vue comme une déduction, notre programme est un système de déduction dans lequel les items représentent des arbres générés par des Grammaires Minimalistes.

Ce rapport propose une présentation des grammaires minimalistes, une description d'un tel algorithme, et une proposition d'implémentation dans un langage fonctionnel typé.

# Chapitre 1

# Introduction

L'un des aspects de la linguistique computationnelle, ou linguistique informatique est la réalisation d'outils de validation des théories de la linguistique classique. L'étude du langage naturel est en effet l'un des domaines d'application les plus anciens de l'informatique, apparu à l'époque de la seconde guerre mondiale avec les premières tentatives de traduction automatique. Les applications sont aujourd'hui multiples, et nous pouvons citer la correction orthographique, ou encore l'interrogation de bases de données en langage naturelle qui sont parmi les plus connues du grand public.

L'étude du langage naturel a connu ses premiers balbutiements il y a plusieurs siècles, mais ce n'est que récemment que la linguistique s'est intéressée aux structures pouvant le représenter. Nous nous intéressons ici à un sous-domaine spécifique de la linguistique, la syntaxe, qui étudie l'agencement des mot dans la phrase. Le but est de reconnaître les suite de mots qui sont des phrases, mais également de leur assigner une structure. La syntaxe est souvent étudiée en parallèle avec la sémantique, qui s'intéresse au sens des phrases analysées, en dehors du contexte où elles peuvent être utilisé ([Amb03]). Néanmoins nous ne nous intéresserons qu'à un aspect strictement syntaxique des grammaires étudiées.

Dans la hiérarchie des langages établie par Chomsky, les langues naturelles se situent à cheval sur les langages algébriques et les langages contextuels. En effet, les grammaires algébriques, ou grammaires hors-contexte, ne sont pas assez puissantes pour permettre de reproduire certains phénomènes du langage humain, alors que les grammaires contextuelles ne sont pas assez strictes et permettent de générer des langages beaucoup plus complexes, analysables en temps NP-complet.

Plusieurs formalismes, qualifiés de légèrement contextuels, ont donc été proposés pour reproduire l'ensemble des phénomènes linguistiques existants tout en permettant une analyse performante des langages qu'ils permettent de générer. D'un point de vue linguistique, une analyse est considérée comme performante lorsqu'elle peut être réalisée en temps au plus polynomial. Bien sur, la structure obtenue pour une expression à la fin d'une analyse doit également correspondre à la structure linguistique de celle-ci.

Parmi ces différents formalismes, les Grammaires Minimalistes (MG) introduites par Stabler ([Sta97]) permettent de formaliser le récent programme minimaliste de Noam Chomsky ([Cho95]). Ces grammaires permettent de décrire la structure syntaxique des phrases des langues naturelles sous forme d'arbres.

Plusieurs algorithmes d'analyse pour les grammaires minimalistes ont été proposées. Néanmoins, nous n'avons répertorié d'implémentation logicielle que pour une adaptation de l'algorithme CKY[1] pour les grammaires minimalistes. Bien qu'ils permettent une analyse en temps polynomial des phrases du langage étudié, ces *reconnaisseurs* n'ont aucun pouvoir de prédiction et calculent une quantité importante d'informations inutiles dans la dérivation de la phrase analysée. De plus, leur fonctionnement reste assez éloigné de celui supposé de l'être humain.

L'algorithme de reconnaissance présenté dans ce rapport possède un pouvoir de prédiction, et peut être amélioré pour analyser les phrases en les parcourant de gauche à droite, comme le fait un être humain. Néanmoins, cet algorithme peut ne pas terminer pour certaines grammaires, et reste essentiellement une étape à l'élaboration d'un algorithme plus élaboré dans le style de celui établi par Earley ([Ear70]), qui utilise la même stratégie de parcours récursif en profondeur des arbres de dérivations.

Nous allons donc décrire la structure des grammaires minimalistes et les dérivations qu'elles permettent de réaliser, avant de les reformuler pour les besoins de notre algorithme, en tirant parti de la faible équivalence qui existe entre elles et les Grammaires Algébriques Multiples.

Nous détaillerons ensuite la structure de l'algorithme de reconnaissance et son fonctionnement, puis nous détaillerons ses propriétés, et quelques améliorations qui peuvent lui être apportées afin de le rendre plus efficace.

Dans une dernière partie, nous discuterons d'une possible implémentation et des difficultés qu'elle soulève, et discuterons de quelques idées d'optimisation.

[1]CKY du nom des auteurs Cocke, Younger et Kasami ([You67])

# Chapitre 2

# Les Grammaires Minimalistes

En 1997, Edward Stabler [Sta97] définit formellement les *Grammaires Minimalistes* (MG) à partir du programme minimaliste de Noam Chomsky [Cho95]. Ces grammaires permettent de définir des langages de classe supérieure à ceux définissables par des grammaires algébriques comme les grammaires à adjonction d'arbre (TAG) ([JS97]).

Dans les grammaires minimalistes, les expressions sont représentées par des arbres binaires, ordonnés, finis, et dont seules les feuilles sont étiquetées. De tels arbres sont définis formellement dans la section 2.1.2.

Le langage défini par une grammaire minimaliste correspond à un sous-ensemble particulier des arbres qu'elle permet de générér.

## 2.1 Définitions

### 2.1.1 Généralités

D'une façon générale, une grammaire est vue comme la spécification d'un lexique et d'un ensemble de fonctions génératrices permettant de construire des expressions complexes.

Une grammaire sera donc notée $G =< \Sigma, Cat, Lex, F >$ où :

- $\Sigma$ est un ensemble de symboles terminaux (pour la syntaxe, ce sont les mots des phrases) ;
- *Cat* est un ensemble de symboles non terminaux (il s'agit pour nous de catégories grammaticales) ;
- *Lex* est un ensemble de symboles de départ (le lexique) ;

– $F$ représente l'ensemble des fonctions génératrices ou règles de dérivation.

L'ensemble des expressions générées par la grammaire correspond à la clôture du lexique à l'aide des fonctions génératrices, notée $\cup_{n=0}^{\infty} CL_n(Lex)$, et que l'on peut définir récursivement de la façon suivante :

$$CL_0(Lex) = Lex$$

$$CL_{n+1}(Lex) = F(CL_n(Lex)) \cup CL_n(Lex)$$

où $F(CL_n(Lex))$ correspond à l'ensemble des expressions générées en appliquant de toutes les façons possibles chacune des fonctions génératrices aux éléments de l'ensemble $CL_n(Lex)$.

Les grammaires minimalistes, issues du programme minimaliste, ont été élaborées dans le but de reproduire les mécanismes semblant être communs à l'ensemble des langages naturels. Leur principe est fondé sur la notion de *mouvement* des éléments au sein d'une expression, comme c'est notamment le cas pour les relatives et dans la plupart des formes interrogatives de la langue française.

Par exemple, la construction de la phrase *combien de livres que Chomsky a écrit a-t-il aimé ?*[1] passe par une étape intermédiaire incorrecte *il a aimé combien de livre que Chomsky a écrit*. L'interrogatif "combien" déclenche alors une transformation qui déplace le constituant souligné en tête de phrase.

Voici quelques autres exemples, où le complément du verbe est déplacé à différent endroit de l'expression selon qu'il s'agit d'une proposition relative (2) ou d'une question (3) :

1. J'ai embrassé une femme
2. Une femme que$_i$ j'ai embrassée $t_i$
3. Qui$_i$ as-tu brassé $t_i$ ?

### 2.1.2 Les arbres

Dans les grammaires minimalistes, les expressions générées sont des arbres ordonnés munis d'une, et d'une relation de projection. Pour une définition formelle détaillée d'une telle structure, le lecteur pourra se référer aux travaux de E. Stabler [Sta97] et de M. Amblard [Amb03].

[1] Cet exemple est emprunté à C. Rétoré ([Rét07]). On peut noter que la phrase subit une autre transformation : l'inversion du sujet *il*.

D'une façon générale, les grammaires minimalistes utilisent des arbres binaires, finis, ordonnés, et dont seules les feuilles sont étiquetées. Un arbre est *complexe* si et seulement si il possède plus d'un nœud, sinon il est *simple*.

La relation de projection ajoutée à la structure d'arbre permet de donner un ordre entre les sous-arbres qui vient se rajouter à l'ordre gauche-droite classique. Cette relation, représentée par le symbole $<$, permet par exemple de réprésenter le fait qu'un déterminant `d` se projette sur un nom `n` pour former un groupe nominal (DP pour *determinant phrase* en anglais [Abn87]). On notera alors $\mathtt{d}<\mathtt{n}$ une telle relation, que l'on peut également représenter sous forme d'arbre

$$\begin{array}{c} < \\ \mathtt{d} \quad \mathtt{n} \end{array}$$

Par la suite, on notera $[_<\tau, \upsilon]$ un arbre de sous-arbres immédiats $\tau$ et $\upsilon$, et dont la racine de $\tau$ précéde et se projette sur la racine de $\upsilon$. De la même façon, $[_>\tau, \upsilon]$ représente un arbre de sous-arbres immédiats $\tau$ et $\upsilon$, et dont la racine de $\tau$ précéde la racine de $\upsilon$, et la racine de $\upsilon$ se projette sur la racine de $\tau$.

**Tête :** Etant donné un arbre $\tau$ tel que défini précédemment, et deux nœuds $x$ et $y$ de cet arbre, alors $x$ est la tête de $y$ si et seulement si :

- $y$ est une feuille et $x = y$, ou
- il existe un nœud $z$ de $\tau$ et de parent $y$ qui se projette sur tous les enfants de $y$, et $x$ est la tête de $z$.

Quelques propriétés intéressantes peuvent être obtenues à partir de cette définition :

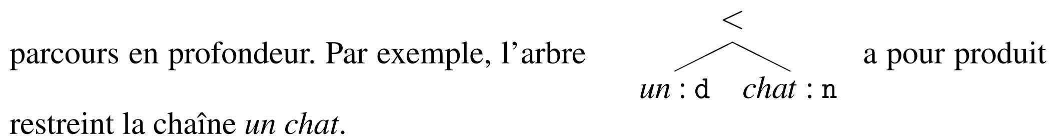
- Toute tête est une feuille, et chaque feuille est sa propre tête ;
- Tout nœud possède une unique tête.

La tête d'un arbre donné est la tête de la racine de cet arbre. On peut retrouver facilement la tête d'un arbre à l'aide d'un parcours en profondeur, en suivant l'ordre de projection, et jusqu'à obtenir une feuille.

**Projection maximale :** Un nœud $y$ est la projection maximale d'une feuille $x$ si et seulement si $x$ est la tête de $y$ et tous les frères de $y$ se projettent sur $y$.

Par extension, on dit qu'un arbre est *maximal* si et seulement si sa racine est la projection maximale d'une tête.

**Produit restreint (ou rendu) :** Le produit restreint d'un arbre (*narrow yield* en anglais) est la concaténation des traits phonétiques apparaissant aux feuilles de l'arbre, ordonnés de gauche à droite comme dans l'arbre, i.e. résultant d'un parcours en profondeur. Par exemple, l'arbre




a pour produit restreint la chaîne *un chat*.

### 2.1.3 Les traits

Les grammaires minimalistes sont entièrement définies par leur lexique, c'est à dire leurs symboles de départ, ou entrées lexicales. Une entrée lexicale est une expression construite à partir de traits non-syntaxiques et de traits syntaxiques, et peut être considérée comme un arbre ne possédant qu'un seul nœud.

L'ensemble $\Sigma$ des traits non syntaxiques peut généralement comporter des traits phonétiques (les mots des phrases) et des traits sémantiques. Toutefois, nous ne nous intéressons ici qu'à l'aspect strictement syntaxique des grammaires minimalistes et donc aux seuls traits phonétiques. Une entrée lexicale correspondant au mot du langage *chat* comportera donc un trait phonétique `chat`.

L'ensemble $\Sigma$ des traits syntaxiques se divise en quatre sous-ensembles :
- L'ensemble $B$ des catégories de base regroupe les catégories syntaxiques des expressions, comme `v` pour les verbes, `n` pour les noms, etc. L'ensemble $B$ contient également le trait spécial `c` appelé également trait *acceptant* ;
- l'ensemble $S$ des sélecteurs, notés `=x` où `x` est un trait de base. Comme nous le verrons plus tard, la fonction *merge* permet à deux arbres de fusionner en combinant un sélecteur avec un trait de base.
- l'ensemble $+L$ des assignateurs, notés `+f`. Ces traits correspondent à une propriété requise par les *assignés*, et pouvant donner lieu à un déplacement. Ces traits sont notamment utilisés pour représenter les cas grammaticaux (accusatif, datif, etc.) ;
- l'ensemble $-L$ des assignés, notés `-f`. Ces traits permettent d'indiquer une demande de propriété.

Les entrées qui composent le lexique de la grammaire sont des listes de traits :

$$Lex \subset \Sigma \times Cat^*, Cat = S \cup B \cup +L \cup -L$$

Il est d'usage même de se limiter à des entrées lexicales formatées de la façon suivante :

$$Lex \subset \Sigma \times S^* \times (+L)^? \times S^* \times B \times (-L)*$$

Toutefois, cette restriction n'est pas obligatoire, comme le montre l'exemple de la section 2.2. Nous en reparlerons d'ailleurs plus spécifiquement dans la section 4.5.1.

### 2.1.4 Les fonctions génératrices

Les grammaires minimalistes manipulent des arbres tels que nous les avons définis dans la section 2.1.2 de ce chapitre. De nouveaux arbres sont construits soit en fusionnant deux arbres en un seul, soit en déplaçant un sous-arbre au sein d'un arbre. Nous avons donc besoin de définir deux fonctions génératrices correspondant à chacune de ces opérations, et dont l'application sera determinée par les traits syntaxiques se trouvant sur les feuilles des arbres manipulés.

Dans la suite, on dira qu'un arbre $\tau$ possède un trait `f` si le trait syntaxique le plus à gauche de la tête de $\tau$ est `f`.

**La fonction *merge* :** Il s'agit d'une fonction partielle de fusion qui permet à une tête de sélectionner une expression du type approprié, pour combiner avec elle.

Deux arbres $\tau$ et $\upsilon$ sont dans le domaine d'application de *merge* si le premier trait syntaxique de la tête de $\tau$ est `=x`, et le premier trait de la tête de $\upsilon$ est `x`, pour `x` appartenant à l'ensemble $B$ des catégories de base. Dans ce cas :

- $merge(\tau, \upsilon) = [_< \tau', \upsilon']$ si $\tau$ est simple, et
- $merge(\tau, \upsilon) = [_> \upsilon', \tau']$ si $\tau$ est complexe,

où $\tau'$ est identique à $\tau$ excepté que `=x` est supprimé, et $\upsilon'$ est identique à $\upsilon$ excepté que `x` est supprimé.

**La fonction *move* :** C'est une fonction partielle de déplacement qui correspond à la réalisation d'une assignation, c'est à dire à la mise en correspondance d'un assigné et de l'assignateur correspondant.

Un arbre $\tau$ est dans le domaine d'application de *move* si il possède un trait `+f` appartenant à l'ensemble des assignateurs, et si il possède exactement un sous-arbre maximal $\tau_0$ possédant un trait `-f` appartenant à l'ensemble des assignés. Dans ce cas :

- $move(\tau) = [> \tau_0', \tau'],$

où $\tau_0'$ est identique à $\tau_0$ excepté que `-y` est supprimé, et $\tau'$ est identique à $\tau$ excepté que `+y` est supprimé, et que le sous-arbre $\tau_0$ est remplacé par un nœud simple sans traits.

**Remarque :** Une contrainte de déplacement de plus court chemin (SMC) inspirée de Chomsky ([Cho95]) a été incorporée dans la définition de la fonction move. Elle empêche un arbre dont la tête posséde un assignateur `+f` et dont plusieurs sous-arbres maximaux possédent l'assigné correspondant `-f` d'être dans le domaine d'application de la fonction.

On définit un arbre comme étant *complet*, et donc appartenant au langage défini par la grammaire, lorsque sa tête ne contient plus aucun trait syntaxique excepté le trait acceptant `c`, et qu'aucun des ses autres nœuds ne porte de traits syntaxiques.

Notre définition des grammaires minimalistes se limite aux besoins de l'algorithme présenté dans la section 3.1 de ce rapport, mais ne permet pas de représenter tous les phénomènes de mouvements que permet la définition originale dans [Sta97]. Toutefois, il est possible de modifier notre algorithme pour qu'il tire parti de toute la puissance génératrice des grammaires minimalistes. Harkema propose une méthode permettant d'incorporer tous les types de mouvements à notre algorithme ([Har01]).

## 2.2 Un exemple simple

Nous prendrons comme exemple une grammaire permettant de générer le langage abstrait $L = \{a^n b^n d^n | n \geq 0\}$. Ce langage n'est pas hors-contexte, et permet donc d'illustrer le pouvoir génératif des grammaires minimalistes. Cet exemple est inspiré de [Har01].

Le lexique contient les huit éléments suivant :

$$\varepsilon : \texttt{c} \tag{2.1}$$
$$\varepsilon : \texttt{=a +Z +Y +X c} \tag{2.2}$$
$$a : \texttt{=b a -X} \tag{2.3}$$
$$b : \texttt{=d b -Y} \tag{2.4}$$
$$d : \texttt{d -Z} \tag{2.5}$$
$$a : \texttt{=b +X a -X} \tag{2.6}$$
$$b : \texttt{=d +Y b -Y} \tag{2.7}$$
$$d : \texttt{=a +Z d -Z} \tag{2.8}$$

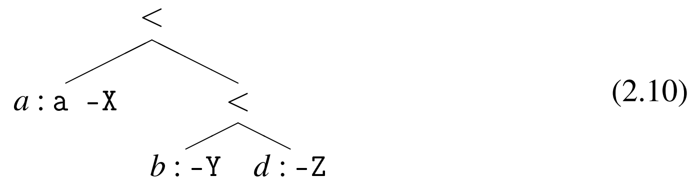
Chaque élément lexical est en fait un arbre d'un seul nœud, étiqueté avec une partie phonétique et des traits syntaxiques de différentes sortes. Les deux premiers éléments de ce lexique possèdent une partie phonétique vide, notée $\varepsilon$.

Les traits déterminent de quelle manière les fonctions génératrices vont être appliquées aux éléments du lexique et aux arbres qui en sont dérivés. Les entrées lexicales $b$ : `=d b -Y` et $d$ :`d -Z` peuvent par exemple fusionner pour former l'arbre suivant :

(2.9)

On retrouve dans cet arbre les mêmes traits syntaxiques que dans les entrées lexicales qui ont fusionné pour le produire, exceptés les traits `=d` et `d` qui ont permis l'application de la fonction *merge* et qui ont donc été supprimés. Le signe $<$ indique la tête de l'arbre, dont le trait syntaxique le plus à gauche est `b`.

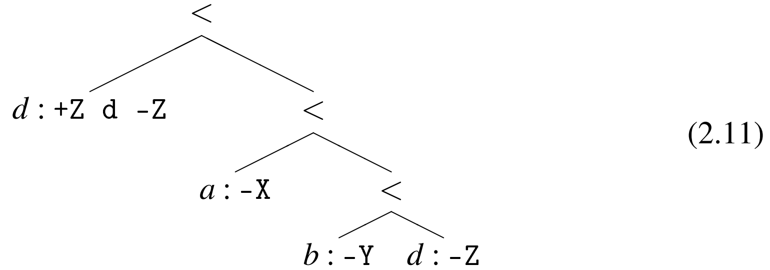
Une seconde étape de la dérivation est la fusion de l'entrée lexicale $a$ : `=b a -X` avec l'arbre que nous venons d'obtenir. De cette étape résulte l'arbre suivant :



(2.10)

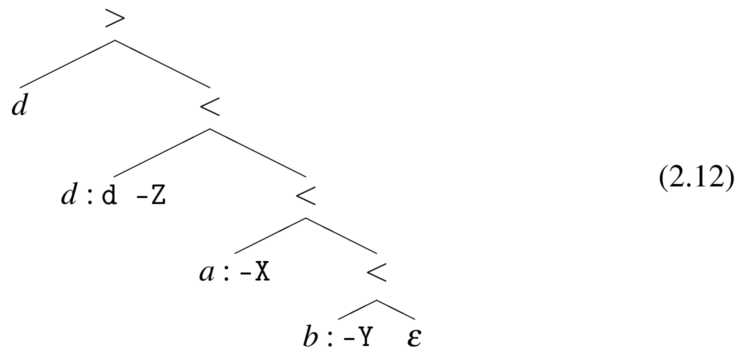
La tête de ce nouvel arbre est la feuille étiquetée $a$ : `a -X`. Comme le trait le plus à gauche est `a`, une nouvelle fusion peut être opérée entre l'entrée lexicale

*d* : `=a +Z d -Z` et celui-ci. Voici donc l'arbre obtenu :



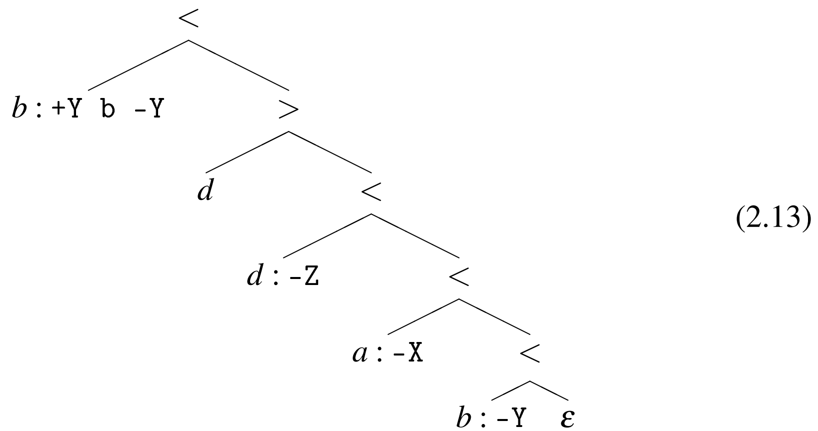

(2.11)

Ici, le trait le plus à gauche de la tête de l'arbre est `+Z`. De plus, l'arbre possède une autre feuille dont le trait le plus à gauche est `-Z`. Dans une telle situation, la règle *move* peut être appliquée. Comme c'est le cas pour la fonction *merge*, les traits permettant l'application de la fonction *move* sont supprimés dans le nouvel arbre.



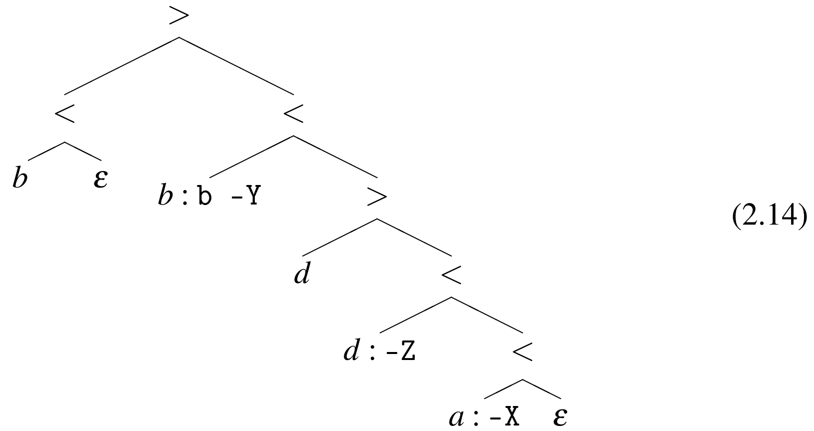

(2.12)

Cet arbre peut à son tour être fusionné avec l'entrée lexicale *b* : `=d +Y b -Y` :




(2.13)

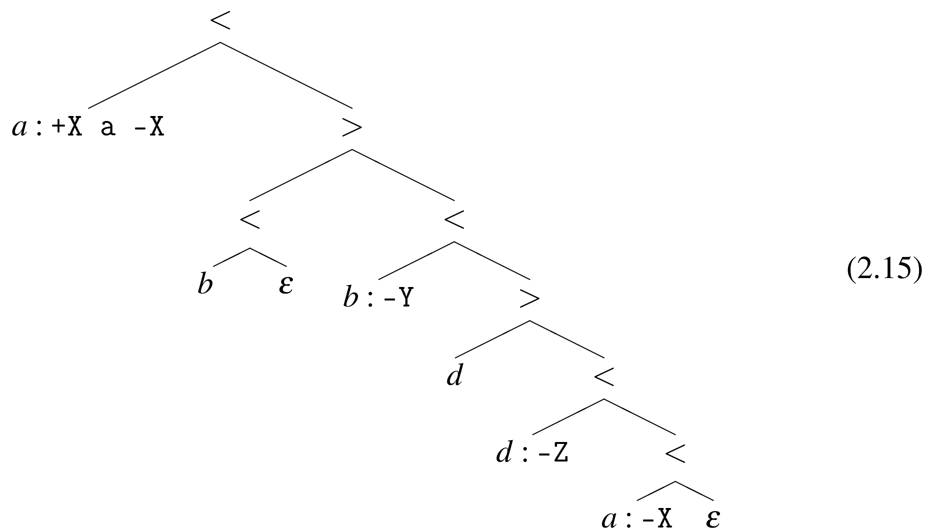
Le trait syntaxique le plus à gauche de la tête de cet arbre est `+Y`. Ici encore, une autre feuille possède l'assigné correspondant `-Y` comme trait le plus à gauche. L'application de la fonction *move* permet donc de produire un autre arbre :




(2.14)

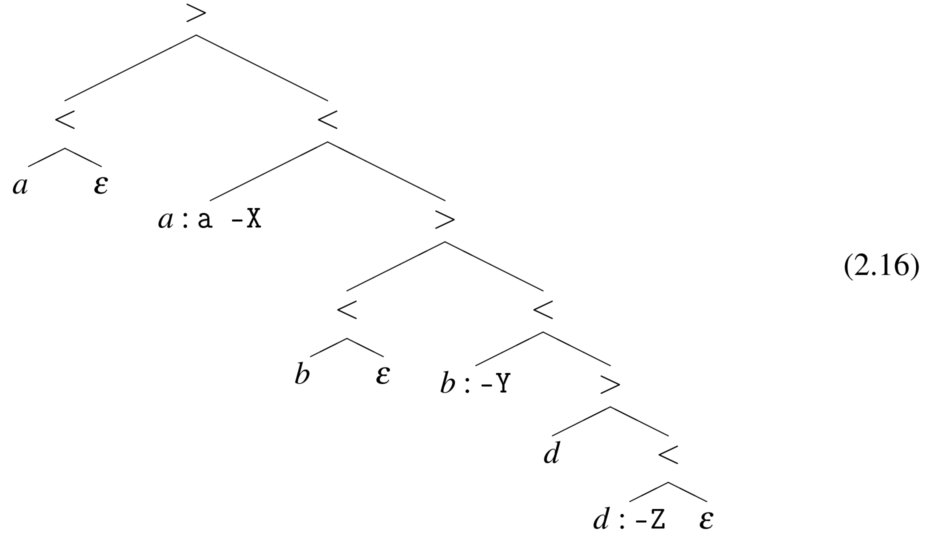
Remarquons qu'ici, ce n'est pas seulement une feuille qui a été déplacé, mais tout un sous-arbre. En effet, lors de l'application de la fonction *move*, c'est le sous-arbre maximal de tête la feuille possédant le trait `-Y` correspondant qui est déplacé.

Les dernières étapes de notre dérivation sont semblabes à celle que nous venons de voir. L'entrée lexicale $a$ : `=b +X a -X` fusionne avec l'arbre représenté en 2.6 :



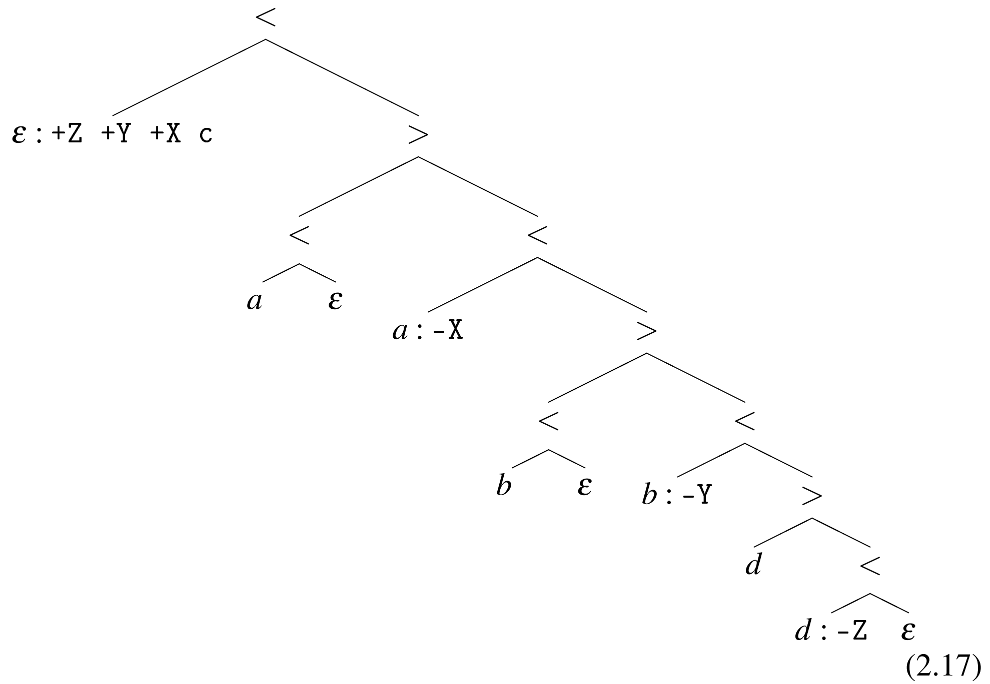

(2.15)

La fonction *move* s'applique à l'arbre représenté en 2.7 :



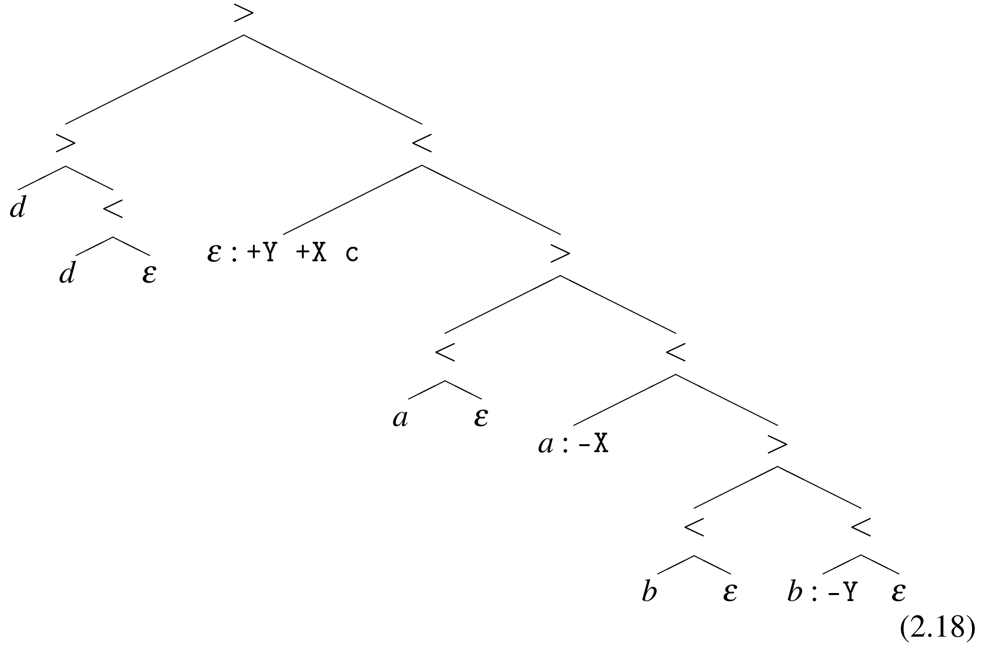

(2.16)

L'arbre ainsi obtenu et l'entrée lexicale ε : `=a +Z +Y +X c` peuvent ensuite fusionner :



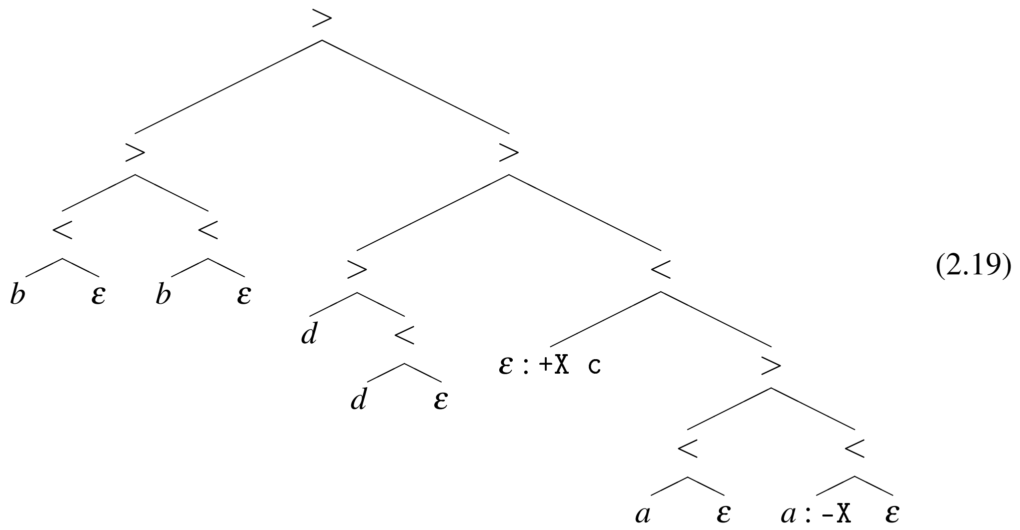

(2.17)

Une nouvelle application de la fonction *move*…



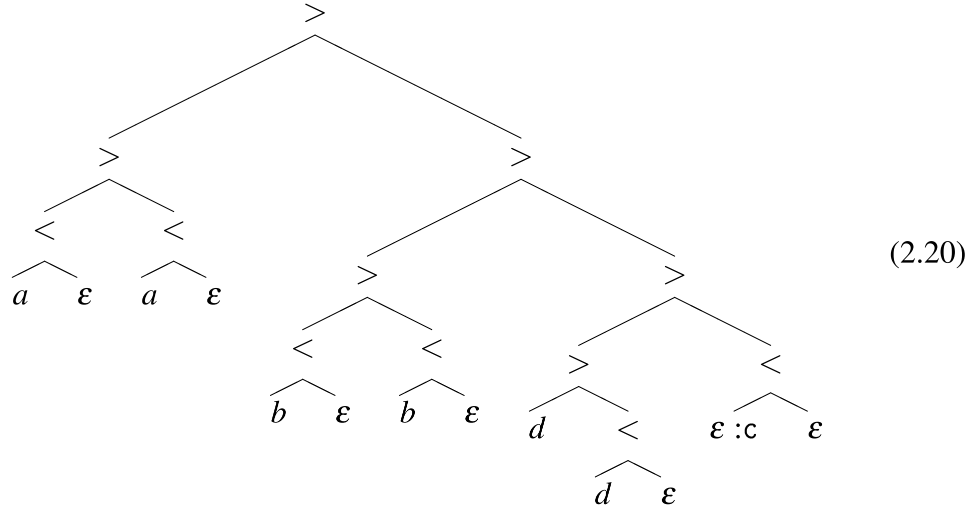

(2.18)

Puis une autre…




(2.19)

Et enfin une dernière permettent de finalement obtenir l'arbre suivant :




(2.20)

Cet arbre ne contient plus qu'un seul trait syntaxique, le trait acceptant c, qui est porté par la tête. Il s'agit donc d'un arbre complet, dont le produit restreint est *aabbdd*. On peut donc en conclure que cette expression appartient au langage généré par notre grammaire.

## 2.3 L'arbre de dérivation

Dans la dérivation de l'expression *aabbdd* de l'exemple précédent, le nombre de traits qui ont été utilisés est de 25. Par conséquent, il nous a fallu 12 étapes, c'est-à-dire 12 applications des fonctions *merge* et *move* pour arriver à un arbre complet. En effet, chacune de ces applications a permis de supprimer deux traits, et il reste un trait dans l'arbre final.

La dérivation peut être résumée à l'aide d'un arbre de dérivation, dans lequel chaque nœud correspond en fait à un arbre obtenu à une étape de la dérivation.

Voici une représentation d'un tel arbre de dérivation pour l'exemple de la section 3.2. Ici, chaque nœud est étiqueté par le numéro de l'arbre auquel il correspond, entrées lexicales incluses.

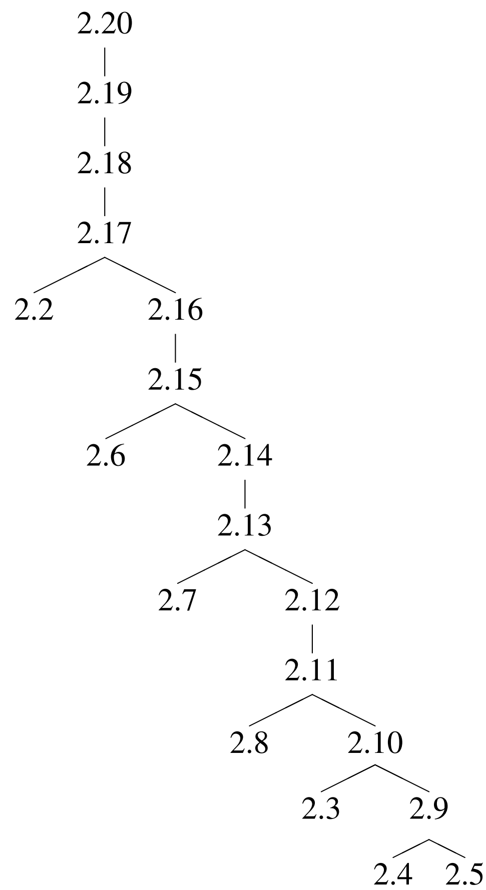



FIG. 2.1 – Arbre de dérivation de la phrase $\omega = aabbdd$

## 2.4 Reformulation des Grammaires Minimalistes

Dans une grammaire minimaliste, les nouveaux arbres sont dérivés à partir d'autres arbres par l'application des fonctions génératrices *merge* et *move*. Michaelis a montré que pour chaque Grammaire Minimaliste, il existe une grammaire algébrique multiple[2] (MCFG) qui génère le même langage ([Mic01b],[MADR06]). Il s'en suit de la construction d'un telle grammaire à partir d'une Grammaire Minimaliste donnée que la dérivation est complètement déterminée par les traits syntaxiques apparaissant dans l'arbre et par le fait que l'arbre soit simple ou complexe.

### 2.4.1 Représentation sous forme de chaînes compactes

Les propriétés syntaxiques d'un arbre, et donc le fait qu'il puisse intervenir dans l'application des fonctions *merge* et *move*, sont complètement et uniquement déterminée par les traits syntaxique qu'il porte, et par le fait qu'il soit simple ou complexe. De ce fait, la géometrie de l'arbre est un artefact de moindre intérêt, excepté que les éléments syntaxiques de la tête doivent être distingués des éléments syntaxiques apparaissant aux autres noeuds de l'arbre.

Il en résulte que pour une utilisation purement syntaxique, et en suivant la terminologie introduite par Stabler dans [Sta01], un arbre peut être représenté par une séquence de chaînes, où chaque chaîne est composée d'une chaîne de caractères pour la partie phonétique, et d'une séquence de traits syntaxiques. Une information supplémentaire est ajoutée à cette séquence pour indiquer si l'arbre est simple ou complexe.

Formellement, un arbre $\tau$ peut donc être représenté par une séquence de la forme $[\gamma_0 : \delta_0, \ldots, \gamma_n : \delta_n]_t$ où $\gamma_i \in \Sigma^*, \delta_i \in Cat^*, 0 \leq i \leq n, t \in \{c, s, l\}$, si les quatres conditions suivantes sont remplies :

- Si $t = s$, $\tau$ est un arbre simple ; si $t = c$, $\tau$ est un arbre complexe ; et si $t = l$, $\tau$ est un arbre lexical. La distinction faite entre des arbres simples et des arbres lexicaux vient du fait que tous les arbres lexicaux sont considérés comme simples, mais qu'un arbre simple peut ne pas être lexical.

[2]Les MCFG (Multiple Context-Free Grammars) sont une extension des grammaires hors-contexte, portant sur des tuples de chaînes, chaque non-terminal étant produit par l'application d'une fonction quelconque à d'autres non-terminaux. La portée des fonctions qu'elle utilise est restreinte, puisqu'elle permet l'effacement de variables, mais en interdit la copie. Le formalisme de ces grammaires est présenté dans [SMFK91]

- Quel que soit $i, 0 \leq i \leq n$, $\tau$ possède une feuille $l_i$ telle que les traits syntaxiques de $l_i$ sont $\delta_i$ ;
- La feuille $l_0$ est la tête de $\tau$ ;
- $\tau$ ne comporte aucune feuille avec des traits syntaxiques, en dehors des feuilles $l_0, \ldots, l_n$ .

Par exemple, l'arbre 2.2 de la section 2.2 peut être représenté ainsi :

$$[b : \texttt{b -Y}, d : \texttt{-Z}]_c$$

### 2.4.2 Reformulation des règles génératrices

La transformation d'une grammaire minimaliste en grammaire algébrique multiple donnée dans [Mic01a] nous permet de reformuler les fonctions génératrices *merge* et *move* sous forme de règles de déduction utilisant des formes compactes d'arbres.

La fonction *merge* peut être appliquée de plusieurs manières différentes, selon que l'arbre possédant le sélecteur est simple ou complexe et que la feuille portant le trait de base intervenant dans la fusion possède d'autre traits syntaxiques ou non.

Ces différentes applications peuvent être représentées à l'aide de trois règles de déduction distinctes :

**Merge-1**

$$\frac{[s : \texttt{=x}\gamma]_s \quad [t : \texttt{x}, \alpha_1, \ldots, \alpha_k]_t}{[st : \gamma, \alpha_1, \ldots, \alpha_k]_c}$$

**Merge-2**

$$\frac{[s : \texttt{=x}\gamma, \alpha_1, \ldots, \alpha_k]_c \quad [t : \texttt{x}, \iota_1, \ldots, \iota_l]_t}{[ts : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l]_c}$$

**Merge-3**

$$\frac{[s : \texttt{=x}\gamma, \alpha_1, \ldots, \alpha_k]_t \quad [t : \texttt{x}\delta, \iota_1, \ldots, \iota_l]_c}{[s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l]_c} \quad \delta \neq \emptyset$$

De la même façon, selon que le sous-arbre déplacé lors de l'application de la fonction *move* peut à nouveau être déplacé ou non, le nouvel arbre obtenu sera représenté de deux façons différentes :

**Move-1**

$$\frac{[s : \texttt{+f}\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \texttt{-f}, \alpha_{i+1}, \ldots, \alpha_k]_c}{[ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k]_c}$$

**Move-2**

$$\frac{[s : \texttt{+f}\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \texttt{-f}\delta, \alpha_{i+1}, \ldots, \alpha_k]_c}{[s : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k]_c} \quad \delta \neq \emptyset$$

Dans l'algorithme que nous allons définir dans le chapitre suivant, nous utiliserons des règles d'induction qui peuvent être considérées comme les règles inverses de celles-ci.

Finalement, en représentant les arbres minimalistes sous forme de chaînes compactes, l'arbre de dérivation de la figure 2.1 page 15 peut également être représenté de la façon suivante :

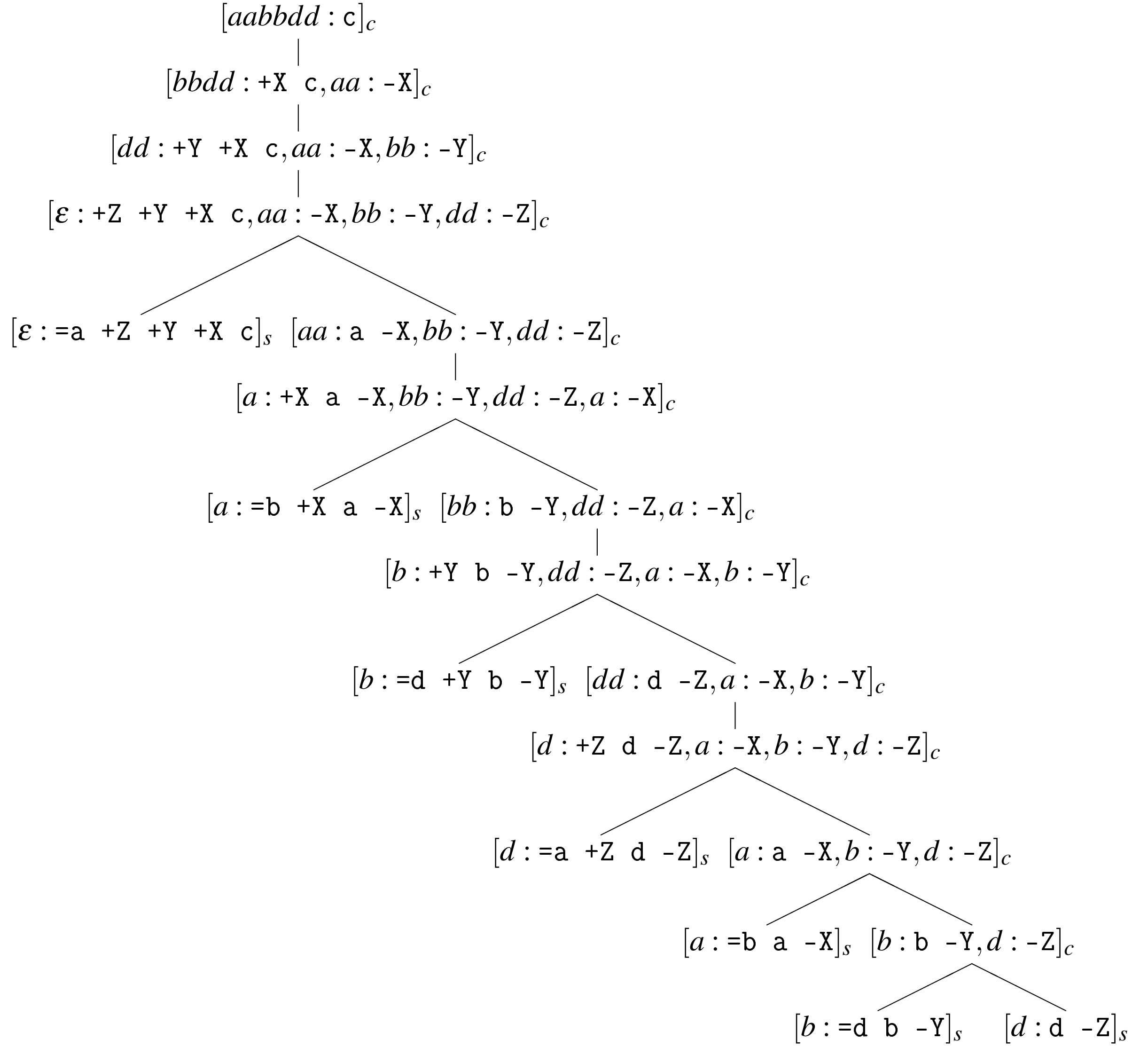



FIG. 2.2 – Arbre de dérivation de la phrase $\omega = aabbdd$ avec une représentation des arbres minimalistes sous formes de chaînes compactes

# Chapitre 3

# Description d'un algorithme

Il existe d'ores-et-déjà plusieurs propositions d'algorithme pour la reconnaissance des langages générés par les Grammaires Minimalistes. Ces algorithmes sont la plupart du temps des adaptations de l'algorithme Cocke-Younger-Kasami (CKY - [AU72]) pour les Grammaires Minimalistes. Il existe plusieurs implémentations de tels algorithmes, comme la version de Stabler écrite dans le langage fonctionnel *prolog*, ou l'adaptation plus récente et enrichie de Hale dans le langage fonctionnel fortement typé *objective Caml*, qui permet des analyses significativement plus rapides. Bien qu'ils permettent une analyse en temps polynomial des phrases du langage étudié, ces *reconnaisseurs* n'ont aucun pouvoir de prédiction et calculent une quantité importantes d'informations inutiles dans la dérivation de la phrase analysée.

L'algorithme présenté ici, bien que défini à la même époque que celui utilisé par les analyseurs de Stabler et Hale, n'a encore jamais été implémenté. Notre travail a donc été de travailler à la réalisation d'un tel programme.

Dans cette section, nous allons décrire un algorithme d'analyse des langages générés par des Grammaires Minimalistes, et dont le fonctionnement peut être assimilé à un parcours récursif en profondeur (*top-down* en anglais) d'arbres de dérivation tels qu'ils ont été définis dans la section 2.3. Les Grammaires Minimalistes utilisées ici sont telles qu'elles ont été définies dans la section 2.1.

Cette algorithme a été formulé par Harkema et est présenté plus en détail dans [Har01].

Bien que les Grammaires Minimalistes permettent de générer des langages d'une classe supérieure à ceux générés par des grammaires non contextuelles, les

dérivations au sein de ce formalisme sont hors-contexte. Cette propriété a été mise en évidence par Michaelis en prouvant que ces grammaires étaient faiblement équivalentes aux Grammaires Algébriques Multiples.

Un tel algorithme possède la faculté de pouvoir prédire où dans une phrase la présence de catégories phonétiquement vides est requise, faculté que les algorithmes actuellement implémentés ne possède pas. Cette capacité est basée sur la structure de données utilisée, structure qui sera décrite plus bas. De plus, cet algorithme peut être amélioré pour avoir un fonctionnement plus proche de celui de l'être humain. Une de ces améliorations permet par exemple d'obtenir un analyseur avec la propriété de préfixe correct. Un tel programme arrête son analyse dès qu'une structure non grammatricale est rencontrée dans la phrase. Les différentes améliorations seront discutées dans la section 3.4.

## 3.1 La reconnaissance par parcours en profondeur récursif

L'algorithme que nous présentons est basé sur la notion d'une analyse vue comme une déduction, comme décrite par Shieber et al. ([SSP95]). L'analyseur utilise un *tableau* (*chart* en anglais) pour stocker des *items* qui permettront de prédire la structure syntaxique de la phrase à analyser. Ce tableau est initialisé avec un ensemble d'*axiomes*. Ces axiomes vont ensuite être dérivés au moyen d'un ensemble de *règles d'inférence*. Si les nouveaux items ainsi générés contiennent un *item but* distinct, la phrase appartient au langage défini par la grammaire. Dans le cas contraire, et en considérant que le système de déduction est *complet* et *correct*[1], la phrase n'appartient pas au langage.

Chacun des termes utilisés est défini dans les sous-sections suivantes.

### 3.1.1 Les items et l'*invariant*

On définit un *item* comme une séquence $\Delta_1 + \ldots + \Delta_m$, où chaque sous-item $\Delta_i$ $(1 \leq i \leq m)$ est une prédiction au vu des traits syntaxiques de l'arbre utilisé dans

[1]i.e. il génère toutes les phrases appartenant au langage défini par une Grammaire Minimaliste donnée (complet), et uniquement celles-ci (correct).

la dérivation. Un sous-item est une *catégorie* avec des vecteurs de position de la forme :

$$[(x_0, y_0) : \gamma_0 \cdot \delta_0, \ldots, (x_n, y_n) : \gamma_n \cdot \delta_n]_t$$

où $\gamma_i, \delta_i \in Cat^*$, $t \in \{c, s, l\}$, et $x_i, y_i \in \mathbb{N}^+$ pour $0 \leq i \leq n$. Un sous-item $\Delta$ est donc une abréviation pour un ensemble d'arbres $T_s(\Delta)$.

Pour une phrase $\omega = \omega_1 \ldots \omega_k$ et un sous-item $\Delta = [(x_0, y_0) : \gamma_0 \cdot \delta_0, \ldots, (x_n, y_n) : \gamma_n \cdot \delta_n]_t$, un arbre $\tau$ appartient à $T_s(\Delta)$ si et seulement si :

- $\tau$ peut être représenté sous la forme d'une chaîne compacte $[\varphi_0 : \delta_0, \ldots, \varphi_n : \delta_n]_t$, où $\varphi_j \in Phon^*, 0 \leq j \leq n$ et $Phon$ est l'ensemble des traits syntaxiques ;
- il existe une entrée lexicale possédant les traits syntaxiques $\gamma_j \delta_j$, $0 \leq j \leq n$ ;
- le rapport restreint de la projection maximale de feuille $l_i$ étiquetée $\delta_i$ de l'arbre $\tau$ correspond à $\omega_{x_i+1} \ldots \omega_{y_i}, 0 \leq i \leq n$.

En considérant que les phrases valides sont des chaînes de la catégorie distincte $\mathtt{c}$, les items générés par l'algorithme seront invariablement interprétés de la façon suivante : étant donné un item $\Delta_1 + \cdots + \Delta_m$, la séquence d'arbres $\tau_1, \ldots, \tau_m$ telle que $\tau_i \in T_s(\Delta_i), 0 \leq i \leq m$, correspond à une coupure[2] d'un arbre de dérivation partiel dont la racine est un arbre complet $\rho$ de catégorie $\mathtt{c}$ et de rendu $\omega$.

Puisque chaque nœud d'un arbre de dérivation correspond en fait à un arbre généré par une Grammaire Minimaliste, ceci définit la notion d'*invariant*.

### 3.1.2 Les axiomes

Les *axiomes* sont des items avec lesquels le *tableau* sera initialisé, et dont seront déduits d'autres items par l'application des règles d'inférence.

Si la phrase à analyser est de longueur $k$, $\mathtt{c}$ étant le trait acceptant pour la grammaire $G$, alors pour chaque entrée lexicale $l$ ne possédant que le trait syntaxique $\mathtt{c}$, il y aura un axiome $[(O, k) : \cdot \mathtt{c}]_s$. De la même façon, pour chaque entrée lexicale $l$ possédant les traits syntaxiques $\gamma\mathtt{c}, \gamma \in Cat^+$ et $Cat$ étant l'ensemble des traits syntaxiques, un axiome $[(O, k) : \gamma \cdot \mathtt{c}]_c$ est ajouté au tableau.

[2]Un ensemble $C$ de nœuds dans un arbre $T$ est une coupure de l'arbre $T$ si aucun nœud de $C$ ne domine un autre nœud de $C$, et si chaque nœud de $T$ et n'appartement pas à $C$ domine ou est dominé par un nœud de $C$.

### 3.1.3 Les règles d'inférence

En fabriquant une Grammaire Hors-Contexte Multiple, à partir d'une Grammaire Minimaliste, de telle façon que les deux grammaires genèrent le même langage, une règle de reécriture hors-contexte va être créée pour chaque application des règles de construction *merge* et *move*. Le nombre de ces règles de reécriture croit donc avec la taille du lexique de la Grammaire Minimaliste de départ de façon exponentielle.

Néanmoins, Harkema montre que toutes ces règles peuvent être condensées en cinq règles d'inférence, du fait même de leur définition ([Har01]) : trois règles de "<défusion"> (*Unmerge*), et deux règles de "<replacement"> (*Unmove*). Ces règles peuvent être vues comme les fonctions inverses des règles de déduction formulées dans la section 2.4.2.

Etant donné un sous-item particulier $\Delta_i$ apparaissant dans un item $\Delta_1 + \cdots + \Delta_m$, les règles d'inférence vont prédire comment les arbres appartenant à $T_s(\Delta_i)$ peuvent être "défusionnés" et "replacés" en arbres plus petits, en spécifiant les sous-items particuliers auquels ils appartiendront. En plus de ces règles, une autre règle *Scan* permet de lire la chaîne fournie en entrée.

Voici une description formelle de chacune de ces règles :

**Unmerge-1** Etant donné un item $\Delta_1 + \cdots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : = \mathrm{x} \cdot \gamma, S]_c, 1 \leq i \leq m,$$

alors pour chaque entrée lexicale avec les traits syntaxiques $\beta\mathrm{x}$, ajouter les items

$$\Delta_1 + \cdots + \Delta_{i-1} + [(p,v) : \cdot = \mathrm{x}\gamma]_s + [(v,q) : \beta \cdot \mathrm{x}, S]_t + \Delta_{i+1} + \cdots + \Delta_m$$

au tableau, pour toutes les valeurs de $v$ telles que $p \leq v \leq q$, et pour toute les valeurs de $t$ telles que $t = s$ si $\beta = \emptyset$, $t = c$ si $\beta \neq \emptyset$, et si $t = s$ alors $S = \emptyset$.

**Unmerge-2** Etant donné un item $\Delta_1 + \cdots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : \alpha = \mathrm{x} \cdot \gamma, S]_c, 1 \leq i \leq m, \alpha \neq \emptyset,$$

alors pour chaque entrée lexicale avec les traits syntaxiques $\beta\mathrm{x}$, ajouter les items

$$\Delta_1 + \cdots + \Delta_{i-1} + [(v,q) : \alpha \cdot = \mathrm{x}\gamma, U]_c + [(p,v) : \beta \cdot \mathrm{x}, V]_t + \Delta_{i+1} + \cdots + \Delta_m$$

au tableau, pour toutes les valeurs de $v$ telles que $p \leq v \leq q$, pour toutes les valeurs possibles de $U, V$ telles que $U \cup V = S$, et pour toute les valeurs de $t$ telles que $t = s$ si $\beta = \emptyset$, $t = c$ si $\beta \neq \emptyset$, et si $t = s$ alors $V = \emptyset$.

**Unmerge-3** Etant donné un item $\Delta_1 + \dots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : \alpha{=}\,\mathrm{x} \cdot \gamma, S, (v,w) : \beta\mathrm{x} \cdot \delta, T]_c, 1 \leq i \leq m,$$

ajouter les items

$$\Delta_1 + \dots + \Delta_{i-1} + [(p,q) : \alpha \cdot = \mathrm{x}\gamma, U]_{t_1} + [(v,w) : \beta \cdot \mathrm{x}, V]_{t_2} + \Delta_{i+1} + \dots + \Delta_m$$

au tableau, pour toutes les valeurs possibles de $U, V$ telles que $U \cup V = S \cup T$, pour toute les valeurs de $t_1$ telles que $t_1 = s$ si $\alpha = \emptyset$, $t_1 = c$ si $\alpha \neq \emptyset$, et pour toute les valeurs de $t_2$ telles que $t_2 = s$ si $\beta = \emptyset$, $t = c$ si $\beta \neq \emptyset$, et si $t_1 = s$ alors $U = \emptyset$, si $t_2 = s$ alors $V = \emptyset$.

**Unmove-1** Etant donné un item $\Delta_1 + \dots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : alpha{+}\mathrm{y} \cdot \gamma, S]_c, 1 \leq i \leq m, \alpha \neq \emptyset,$$

alors pour chaque entrée lexicale avec les traits syntaxiques $\beta{-}\mathrm{y}, \beta \neq \emptyset$, ajouter les items

$$\Delta_1 + \dots + \Delta_{i-1} + [(v,q) : \alpha \cdot +\mathrm{y}\gamma, (p,v) : \beta \cdot -\mathrm{y}, S]_c + \Delta_{i+1} + \dots + \Delta_m$$

au tableau, pour toutes les valeurs de $v$ telles que $p \leq v \leq q$.

**Unmove-2** Etant donné un item $\Delta_1 + \dots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : \alpha{+}\mathrm{y} \cdot \gamma, S, (v,w) : \beta{-}\mathrm{y} \cdot \delta, T]_c, 1 \leq i \leq m, \alpha \neq \emptyset, \beta \neq \emptyset,$$

ajouter l'item

$$\Delta_1 + \dots + \Delta_{i-1} + [(p,q) : \alpha \cdot +\mathrm{y}\gamma, S, (v,w) : \beta \cdot -\mathrm{y}\delta, T]_c + \Delta_{i+1} + \dots + \Delta_m$$

au tableau.

**Scan** Etant donné un item $\Delta_1 + \dots + \Delta_m$ comportant un sous-item

$$\Delta_i = [(p,q) : \cdot\gamma]_s, 1 \leq i \leq m,$$

alors si il existe une entrée lexicale $l$ avec les traits syntaxiques $\gamma$ et de traits phonétiques couvrant la sous-chaîne $\omega_{p+1} \dots \omega_q$ de la chaîne en entrée[3], ajouter l'item suivant au au tableau :

$$\Delta_1 + \dots + \Delta_{i-1} + [(p,q) : \cdot\gamma]_l + \Delta_{i+1} + \dots + \Delta_m$$

[3] c'est-à-dire que $l \in T_s(\Delta_i)$

De plus, une restriction est apportée aux règles d'inférence *Unmove-1* et *Unmove-2* afin qu'elles ne génèrent aucun item ne répondant pas à la contrainte de déplacement de plus faible longueur[4].

### 3.1.4 Les items but

N'importe quel item de la forme $[(x_0,y_0) : \cdot\gamma_0]_l + \cdots + [(x_n,y_n) : \cdot\gamma_n]_l$ est un item but.

En effet, d'après la définition de la règle *Scan*, il existe des entrées lexicales $l_i \in T_s([(x_i,y_i) : \cdot\gamma_i]_l), 0 \leq i \leq m$. Comme $l_1,\ldots,l_m$ est une coupure d'un arbre de dérivation partiel dont la racine est un arbre complet de catégorie `c` et dont le produit restreint correspond à $\omega$, et que chaque $l_i$ correspond à une entrée lexicale, l'arbre de dérivation est en fait un arbre de dérivation complet.

Autrement dit, $\omega$ appartient au langage généré par la grammaire $G$.

## 3.2 Exemple

Pour illustrer le fonctionnement de cet analyseur, nous allons reprendre la grammaire donnée en exemple dans la section 2.2, et analyser si la phrase $\omega = aabbdd$ appartient au langage qu'elle permet de générer.

Puisque la phrase testée est de longueur $k = 6$, l'analyse débute par l'initialisation du tableau avec les axiomes

$$[(0,6) : \cdot\mathtt{c}]_s \tag{3.1}$$

et

$$[(0,6) : \mathtt{=a\ \ +Z\ \ +Y\ \ +X} \cdot \mathtt{c}]_c \tag{3.2}$$

créés respectivement à partir des entrées lexicales $\varepsilon$ : `c` et $\varepsilon$ : `=a +Z +Y +X c`. On constate que l'axiome 3.2 correspond à une coupure de l'arbre de dérivation de la figure 2.1 page 15.

Lorsque la règle *Scan* va être appliquée à l'axiome 3.1, l'analyseur va rechercher dans le lexique une entrée ayant `c` pour traits syntaxiques, et dont les traits

[4]Un item viole la contrainte de déplacement de plus faible longueur s'il contient un sous-item $[(x_0,y_0) : \gamma_0 \cdot \delta_0, \ldots, (x_n,y_n) : \gamma_n \cdot \delta_n]$ pour lequel il existe $\delta_i$ et $\delta_j$ $(1 \leq i < j \leq n)$ pour lesquel le trait le plus à gauche est le même assigné `-y`.

phonétiques couvrent la chaîne $\omega_{0+1}\dots\omega_6 = aabbdd$. Une telle entrée n'existe pas et aucun nouvel item n'est donc ajouté au tableau.

Pour l'axiome 3.2, en revanche, la règle *Unmove-1* va être appliquée, puisqu'il existe plusieurs items possédant des traits syntaxiques de la forme $\gamma$-X, $\gamma \in Cat^*$.

Un des items produit est alors $[(2,6)$ : =a +Z +Y · +X c, $(0,2)$ : =b +X a · -X$]_c$.

Ce nouvel item correspond à une coupure de l'arbre de dérivation de la figure 2.1. L'application de la règle *Unmove-1* va en fait produire quatorze items $[(v,6)$ : =a +Z +Y · +X c, $(0,v)$ : =b +X a · -X$]_c$ et $[(v,6)$ : =a +Z +Y · +X c, $(0,v)$ : =b a · -X$]_c$, , $0 \leq v \leq 6$. Chacun de ces items respecte l'invariant défini dans la section 3.1.1, mais ne correspond pas à une coupure de l'arbre de dérivation de la figure 2.1.

Plusieurs applications successives de la règle *Unmove-1* aux nouveaux items produits permettent d'obtenir, entre autres, l'item $[(6,6)$ : =a · +Z +Y +X c, $(0,2)$ : =b +X a · -X, $(2,4)$ : =d +Y b · -Y, $(4,6)$ : =a +Z d · -Z$]_c$.

Cet item correspond lui aussi à une coupure de l'arbre de dérivation au niveau du nœud étiqueté 2.17.

La règle *Unmerge-3* peut lui être appliquée afin de produire l'item $[(6,6)$ : ·=a +Z +Y +X c$]_s + [(0,2)$ : =b +X · a -X, $(2,4)$ : =d +Y b · -Y, $(4,6)$ : =a +Z d · -Z$]_c$.

Cet item peut être reécrit de deux manières différentes. Une première possibilité est de lui appliquer la règle *Scan* pour produire l'item :

$[(6,6)$ : ·=a +Z +Y +X c$]_l + [(0,2)$ : =b +X · a -X, $(2,4)$ : =d +Y b · -Y, $(4,6)$ : =a +Z d · -Z$]_c$.

En effet, le lexique contient bien une entrée dont les traits syntaxiques sont =a +Z +Y +X c et dont la partie phonétique $\varepsilon$ couvre la sous-chaîne $\omega_{6+1}\dots\omega_6 = \varepsilon$ de la phrase à analyser.

La seconde possibilité est une nouvelle application de la règle *Unmove-1* qui produit alors trois items $[(6,6)$ : ·=a +Z +Y +X c$]_s + [(v'',2)$ : =b · +X a -X, $(0,v'')$ : =b a · -X, $(2,4)$ : =d +Y b · -Y, $(4,6)$ : =a +Z d · -Z$]_c$, $0 \leq v'' \leq 2$.

Notons que puisque la règle *Scan* n'a pas encore été appliquée pour réécrire le premier sous-item, elle peut toujours l'être à ce stade.

En continuant à appliquer chacune des règles d'induction aux nouveaux items générés, on finit par obtenir un item :

$[(6,6):\cdot\texttt{=a +Z +Y +X c}]_l + [(1,2):\cdot\texttt{=b +X a -X}]_l + [(3,4):\cdot\texttt{=d +Y b -Y}]_l + [(5,6):\cdot\texttt{=a +Z d -Z}]_l + [(0,1):\cdot\texttt{=b a -X}]_l + [(2,3):\cdot\texttt{=d b -Y}]_l + [(4,5):\cdot\texttt{d -Z}]_l$.

Cet item est un item but qui correspond à une coupure à travers les feuilles de l'arbre de dérivation de la figure 2.1 page 15.

## 3.3 Preuve et complexité

L'algorithme d'analyse présenté ci-dessus est *correct* et *complet*. Harkema donne une démonstration détaillée de telles propriétés dans [Har01]. Bien que les différentes preuves soient complexes, nous pouvons en donner ici les grandes lignes directrices.

Il est tout d'abord correct car chaque item qu'il permet de générer respecte l'invariant défini dans la section 3.1.1. Pour le prouver, il suffit de montrer que les axiomes respectent l'invariant et sont donc *corrects*, puis que les règles d'inférence ne permettent de générer que des items corrects. Ces preuves sont basées sur la définition même des items, de leur structure, et des expressions qu'ils représentent.

Pour ensuite démontrer que cet algorithme est complet, il faut prouver que pour chaque phrase appartenant au langage défini par la grammaire, il existe une déduction d'un item but à partir des axiomes. Pour celà, Harkema explique que si une phrase $\omega$ appartient au language $L(G)$ d'une Grammaire Minimaliste $G$, alors il existe un arbre de dérivation $T$ complet pour $\omega$ dans $G$, et que cet algorithme permet de générer un item but correspondant à une coupure de cet arbre au niveau des feuilles.

Comme nous l'avons vu, les items générés par l'algorithme sont des coupures d'arbres de dérivation. Il en résulte que malheureusement, et si la grammaire étudiée permet une infinité de phrases ambigües[5], l'algorithme peut ne pas se terminer, produisant une infinité de nouveaux items. Une grammaire dont le lexique comporterait les deux entrées $\varepsilon$ : `c` et $\varepsilon$ : `=c c` peut en être un exemple abstrait.

Pour calculer la complexité de cet algorithme, nous nous intéresserons donc à la déduction la plus courte d'un item but. Le nombre d'items générés au cours d'une déduction est fonction de la taille de la phrase à analyser puisqu'elle détermine le nombre d'item qui vont être créés à l'application d'une règle d'inférence.

[5]L'existence de telles phrases semblent probable dans les langues naturelles, au vu du nombre important d'éléments abstraits et phonétiquement vide apportés par les linguistes.

Il dépend également de plusieurs autres caractéristiques de la grammaire, comme le nombre d'assignés contenu dans l'ensemble $(-L)$, et le nombre maximal de traits utilisés par une entrée lexicale. On peut donc établir une complexité générale en temps polynomial en $O(n^p)$, pour une phrase de longueur $n$, avec $p$ dépendant des caractéristiques de la grammaire. Une estimation théorique précise reste cependant difficile à calculer, et des tests expérimentaux effectués sur une implémentation seraient intéressants.

## 3.4 Extension de l'algorithme et améliorations

Afin de rendre l'algorithme présenté dans la section 3.1 plus efficient, Harkema propose de l'enrichir en lui permettant de prévoir si un item produit est inutile pour le reste de la preuve. C'est la fonction de *prédiction*, ou *look-ahead* en anglais. Une autre amélioration proposée vise à apporter à l'algorithme la propriété de préfixe correct.

### 3.4.1 Fonction de prédiction

Nous avons vu dans l'exemple de la section 3.2 qu'un certain nombre d'items générés par les règles d'inférence ne correspondaient pas à des coupures de l'arbre de dérivation de la phrase à analyser. La présence de tels items vient du fait qu'une règle décompose les vecteurs de position des sous-items de toutes les manières possibles.

Muni d'une fonctionnalité supplémentaire de prédiction, l'algorithme décrit précédemment peut vérifier, pour chaque nouvel item sur le point d'être ajouté au tableau, si celui-ci peut conduire à une déduction d'un item but pour la phrase à analyser.

Il est intéressant de noter qu'un tel ajout n'affecte en rien la complétude de l'algorithme de départ, tout en en améliorant l'efficacité puisque les nombres d'items utilisés par les règles est réduit de façon significative.

### 3.4.2 La propriété de préfixe correct

Un analyseur a la propriété de préfixe correct si il analyse la phrase de la gauche vers la droite, et dans le cas d'une phrase syntaxiquement incorrecte, il s'arrête au premier mot qui ne rentre pas dans une structure grammaticale. En

d'autres termes, il n'analysera pas de préfixe qui ne peut pas être étendu à une phrase grammaticalement correcte dans le langage.

Un tel fonctionnement semble être proche de l'analyse humaine d'une phrase, à l'écrit comme à l'oral. Les algorithmes qui sont une adaptation de l'algorithme CKY aux Grammaires Minimalistes ne possède pas une telle propriété. Une telle amélioration est donc intéressante, tant au point de vue computationnel que dans l'optique d'une optimisation de l'algorithme, puisque l'analyse s'arrête dès que la phrase est reconnue comme non-grammaticale.

# Chapitre 4

# Vers une implémentation

Une implémentation telle que celle proposée en partie ici présente essentiellement un intérêt au niveau de la recherche sur certaines propriétés des Grammaires Minimalistes par le biais des tests qu'elle permettra d'effectuer, et une étape vers l'implémentation d'un algorithme "à la Earley" ([Har01],[Ear70]).

En effet, comme cela a été dit dans le chapître précédent, il existe des grammaires pour lesquelles l'exécution de cet analyseur peut ne pas terminer, mais ces grammaires n'ont pas encore été caractérisées. L'analyse pourra également permettre de calculer les quantités d'informations utilisées dans l'analyse d'une phrase, et ainsi permettre une comparaison statistique des performances de cet algorithme par rapport à ceux déjà implémentés.

Certaines recherches semblent démontrer que l'utilisation d'un langage de programmation fonctionnel à typage fort facilite le développement de programme sur les langages naturels ([Ska98],[Lju02a],[Lju02b]), notemment par rapport aux langages de programmation impératifs.

Etant donnée l'existence d'une implémentation en `objective Caml`[1] (oCaml) d'un algorithme de type CKY généralisé aux grammaires minimalistes, il nous est apparu naturel de choisir ce langage pour développer notre implémentation.

[1]Objective Caml est un langage de programmation fonctionnel, objet, modulaire et à typage fort - voir `http://caml.inria.fr`

## 4.1 Forme générale du programme

Le programme final devra prendre en entrée une Grammaire Minimaliste et une chaîne de caractères[2], et retourner en sortie les dérivations correspondant à la construction d'une expression valide.

En suivant la description faite de l'algorithme dans le chapître précédent, cette fonction s'exprime comme un système de déduction utilisant des items, et permettant de rechercher une preuve pour un item but. Les règles d'inférence constituent donc la spécification de notre analyseur, dont le fonctionnement est basé sur une structure de tableau.

Chaque règle ne s'appliquant qu'à un seul item du tableau, et éventuellement à une entrée lexicale, on peut alors prendre les items contenus dans le tableau un par un et dans leur ordre d'insertion dans le tableau, et leur appliquer chacune des règles de déduction. Les nouveaux items issus de ces applications seront ajouter à la suite des autres dans le tableau. Le tableau peut ainsi être assimilé à un agenda, où chaque règle n'est appliquée qu'une seule fois à chaque item.

Chaque règle est représentée par une fonction qui prend en entrée un item, et le lexique de la grammaire pour certaines, et qui retourne une liste de solutions (liste d'items), contrairement à un prédicat qui pourrait retourner plusieurs résultats différents.

Le problème de la terminaison du programme soulève quelques interrogations. Comme on l'a vu, si la grammaire étudiée permet une infinité de phrases ambigües, l'algorithme va tourner en boucle, et ne retourner aucun résultat. Pourtant, si la phrase analyser appartient bien au language, au moins un item but est trouvé au bout d'un temps polynomial, fonction de la longueur de la phrase. Afin de minimiser le problème de la non-terminaison de notre programme, sans pour autant le règler, on peut décider de retourner un résultat dès qu'un item but est trouvé, et de ne pas rechercher d'éventuelles autres solutions.

Comme l'analyseur présenté a surtout vocation à réaliser des tests sur les grammaires étudiées, on peut donner à l'utilisateur le choix de la stratégie à utiliser en lui proposant une option lors du lancement de l'application.

[2]i.e. la phrase à analyser

## 4.2 La grammaire

Les grammaires acceptées par l'analyseur sont des grammaires écrites dans le même format que les grammaires utilisées par les analyseurs de Stabler ([Sta01]) et Hale ([Hal03]).

Un tel choix nous permet dans un premier temps d'utiliser une bibliothèque de fonctions déjà existantes pour la manipulation de telles grammaires, et plus tard de pouvoir comparer pour une même grammaire l'efficacité de notre implémentation à celles des analyseurs déjà existants.

Le format utilisé permet à l'utilisateur de spécifier un ensemble d'entrées lexicales, et la catégorie acceptante. Les règles d'induction étant communes à toutes les grammaires minimalistes, l'utilisateur peut donc définir lui-même la grammaire qu'il souhaire étudier.

## 4.3 Les items

Dans les Grammaires Minimalistes, les entrées lexicales sont composées de listes de traits. Ces traits peuvent être de plusieurs sortes, et possèdent généralement un nom grammaticalement spécifique (par exemple `n` pour un nom). Partant de là, un type peut être représenté en oCaml à l'aide d'une énumération de type :

```
type feature =
    Select of name
  | Category of name
  | Attract of name
  | Licensee of name
  | Phonetic of name
type flist = feature list
```

En utilisant un tel type, un sous-item[3] est simplement une liste de chaîne munie d'une information sur la complexité du sous-item (complexe, simple ou lexical), où chaque chaîne est composée de deux listes de traits et d'un vecteur de positions, informations utiles pour l'application des règles d'inférence.

[3]noté formellement $[(x_0,y_0) : \gamma_0 \cdot \delta_0, \ldots,]_t$ où pour $0 \leq i \leq n$, $x_i$ et $y_i$ représentent des positions, $\gamma_i$ et $\delta_i$ des listes de traits.

```
type index = Position of int
type yesno = No | Yes | Lex
type cinfo = {left:index; right:index;
              used:flist; exposed:flist}
type chain = Chain of cinfo
type sitem = chain list * yesno
type item = sitem list
```

Les champs `left` et `right` représentent ici le vecteur de position d'une chaine, `used` les traits syntaxiques déjà utilisés par l'arbre minimaliste que représente le sous-item, et `exposed` les traits restants.

### 4.3.1 Les variables

Lors de l'application d'une règle d'inférence à un item, plusieurs nouveaux items peuvent être créés et dont la seule différence des uns par rapport aux autres est la position de chaînes composant un de leurs sous-items.

Plutôt que de tous les créer, on peut imaginer utiliser des variables et ne créer qu'une seule nouvelle règle, et espérer ainsi gagner un peu en compléxité et réduire le coût en espace mémoire. Pourtant cette solution posent plusieurs problèmes, et ne sera pas retenue dans cette première implémentation.

Tout d'abord, une telle variable ne doit pas représenter toutes les valeurs d'entier possible, mais seulement celle d'un intervalle défini par l'item auquel est appliquée la règle d'inférence. Cet intervalle doit donc lui aussi être stocké en mémoire.

Ensuite, des valeurs sont assignées à ces variables lors de l'application de la règle *Scan*, et la présence de plusieurs variables non-instantiées dans un même item compléxifie le fonctionnement de la règle elle-même, peut poser des problèmes de déterminisme.

Enfin, l'existence de telles variables complique considérablement les tests de redondance effectués lors de l'ajout d'un nouvel item au tableau.

Néanmoins, si cette solution est pour le moment écartée, les types définis prennent déjà en compte la possibilité d'utiliser de telles variables dans l'optique d'une éventuelle optimisation future.

```
[...]
 type index = Position of int | PositionVar of string
[...]
```

## 4.4 Le tableau

Le tableau est la structure de données où les axiomes et les nouveaux items vont être stockés. Pour pouvoir reconstruire l'arbre de dérivation d'une phrase reconnue comme appartenant au langage généré par la grammaire, ou tout du moins pour retrouver les différentes étapes de la déduction qui ont conduit à un item but, il est donc intéressant de ne pas simplement utiliser une liste d'item en guise de tableau, mais d'ajouter quelques informations supplémentaires.

Un premier champ à ajouter est l'adresse de l'item, c'est à dire son numéro d'arrivée dans le tableau. La valeur de ce champ est incrémentée automatiquement lors de l'ajout d'un nouvel item.

Un second champ correspond à l'adresse de l'item dont est issu l'item inséré, après l'application d'une des règles d'induction. Il fonctionne donc un peu à la manière d'un pointeur.

Remarquons que bien qu'un item ne soit présent qu'une seule fois dans le tableau, il peut être déduit plusieurs fois d'items différents et/ou par l'application de règles différentes. Pour pouvoir prendre cela en compte, ce second champ doit en fait être un tableau d'adresses modifiable. Lors d'une tentative d'ajout d'un item déjà présent au tableau, ce champ sera mis à jour avec l'ajout d'une nouvelle adresse.

Notons qu'un tel ajoût d'informations n'augmente pas la complexité de l'algorithme général, pour un faible sacrifice en espace mémoire. Notre tableau est donc en fait une liste d'item, chacun étant muni d'une adresse, et d'un tableau d'adresse :

```
 type cpt = Position of int
 type cptlist = cmpt list
 type titem = {entree:item; adresse:cpt;
               mutable peres:cptlist}
 type tableau = titem list
```

```
val dessin : item -> tableau -> tableau = <fun>
```

## 4.5 les règles d'inférence

Chaque règle d'induction va être représentée par une fonction qui prend en entrée un item, et éventuellement la liste des entrées lexicales, et qui retourne une liste de nouveaux items.

Dans un souci d'optimisation, la liste des entrées lexicales passée aux fonctions est en fait simplifiée comme une liste des listes de traits syntaxiques présents dans le lexique, sauf pour la règle *Scan*. Imaginons que le lexique contient les deux éléments suivant :

- chat : `=d n -case`
- livre : `=d n -case`

Dans ce cas, la liste fournie aux fonctions a seulement besoin de contenir une entrée `=n d -case` correspondant aux deux entrées lexicales.

Pour un item, et une de ces entrées, une règle comme *Unmerge-1* peut être vue comme une sorte de règle de déduction :

$$\frac{\Delta_1 + \ldots + \Delta_{i-1} + [(p,q) : \texttt{=x} \cdot \gamma, S]_c + \Delta_{i+1} + \ldots + \Delta_m \quad [\texttt{x}\delta]}{\Delta_1 + \ldots + \Delta_{i-1} + [(p,v) : \texttt{=x} \cdot \gamma]_s + [(v,q) : \delta \cdot \texttt{x}, S]_c + \Delta_{i+1} + \ldots + \Delta_m}$$

avec les restrictions données dans le chapitre précédent.

Pour chaque sous-item dans le domaine d'application de la règle, pour chaque entrée lexicale correspondante, et pour chaque position de vecteur possible, un nouvel item va donc être créé.

Un exemple du code permettant de représenter les règles d'induction est présenté en annexe A.

### 4.5.1 Cas particuliers

Bien que la mise en place des règles d'inférence sous forme de fonction semble assez simple, certaines propriétés posent quelques difficultés.

Nous allons spécialement nous intéresser à la problématique posée par la fonction *Unmerge-3*, bien que la fonction *Unmerge-2* soulève également les mêmes questions.

Pour que la règle *Unmerge-3* puisse être appliquée à un item, celui-ci doit posséder un sous-item $\delta_i$ de la forme

$$\Delta_i = [(p,q) : \alpha{=}\mathtt{x} \cdot \gamma, S, (v,w) : \beta\mathtt{x} \cdot \delta, T]_c, 1 \leq i \leq m,$$

où $\beta$ correspond à une chaîne non vide de traits syntaxiques, et $S$ et $T$ correspondent à des suites de chaînes de la forme $(x,y) : \mu \cdot \nu$, où $\mu$ et $\nu$ sont également des chaînes de traits syntaxiques pouvant être vides.

$S$ et $T$ peuvent chacun être vus comme un ensemble de chaînes, puisque l'ordre des chaînes au sein de chaque liste n'importe pas.

Pour créer les nouveaux items qui vont être ajoutés au tableau, la règle va devoir construire tous les couples d'ensembles $U$ et $V$ possibles tels que $U \cup V = S \cup T$. La compléxité d'une telle construction est exponentielle, et dépend de la taille des ensembles $S$ et $T$ de départ. En effet, si $S$ contient $m$ éléments, et $T$ en contient $n$, le nombre de couples d'ensemble $(U,V)$ distincts est de $2^{m+n}$.

Néanmoins, cette compléxité peut être diminuée si l'on suit la définition formelle de la règle d'induction vue dans le chapitre précédent. Il suffit par exemple d'observer l'item dérivé pour obtenir quelques informations sur les ensembles à calculer.

En effet, si $\alpha$ correspond à une chaîne vide de traits syntaxiques, alors $U$ doit être vide. De la même façon, si $\beta$ est vide dans l'item de départ, alors $V$ doit également être vide.

De plus, nous pouvons étudier quelques propriétés des grammaires minimalistes et de notre analyseur qui peuvent également permettre de relativiser une telle complexité.

Premièrement, les entrées lexicales d'une grammaire minimaliste ont un nombre d'assignés fini, et généralement faible. De plus, il est généralement d'usage de n'utiliser que des entrées lexicales dont la liste des traits syntaxiques est de la forme :

$$S^* \times (+L)^? \times B \times (-L)^*,$$

où $B, S, +L$ et $-L$ sont les sous-ensembles de traits définis dans le premier chapitre.

Ensuite, en analysant les règles de déduction de notre algorithme, on constate que seule la fonction *Unmove-1* permet d'augmenter le nombre de chaînes qui composent un sous-item. Les chaînes qui peuvent être ajoutées de cette façon sont des chaînes de la forme $(x,y) : \beta \cdot \texttt{-f}\delta$, où `-f` appartient à l'ensemble des assignés utilisés par la grammaire.

La contrainte de déplacement de plus faible longueur nous permet donc d'affirmer que le nombre maximum de chaînes qui composent un sous-item est égal au nombre d'assignés de la grammaires (plus un pour la chaîne représentant la tête de l'arbre).

On voit donc qu'en règle générale, le nombre d'éléments inclus dans $S \cup T$ reste faible, et donc que la compléxité du calcul des sous-ensembles $U$ et $V$ n'est pas facteur à augmenter significativement la complexité générale de l'algorithme. Toutefois, il n'est pas exclu de travailler avec des grammaires comportant un nombre important d'assignés, dans une optique purement théorique par exemple. Dans cette optique, nous avons donc cherché à optimiser le fonctionnement même de notre algorithme.

Une solution intéressante semble être de modifier la structure des items dans le but de stocker des ensembles de valeurs possibles. Plutôt que de produire plusieurs items correspondant à chaque couple d'ensembles $(U,V)$, il suffit de mémoriser dans la structure de l'item un ensemble de chaînes qui peuvent être utilisées par plusieurs sous-items. Lorsqu'un des éléments d'un tel ensemble est nécessaire à un sous-item pour l'application d'une règle de déduction, l'opération se passe comme si le sous-item possédait cette propre chaîne. Il suffit alors de supprimer la chaîne qui a été utilisée de l'ensemble des valeurs à disposition.

Cette modification permet, comme l'idée d'utiliser des variables dans les vecteurs de position, de réduire le nombre d'items créés tout en évitant de perdre du temps à calculer l'ensemble des couples $(U,V)$. On obtient donc un algorithme plus efficace.

Dans notre implémentation, il va donc falloir ajouter une information à chacun de nos items. En plus de la liste de sous-items qui le composent, nous allons donc ajouter une liste de chaînes de type `cinfo`. Chacun des éléments qui vont être placés dans cette liste n'est pas utilisable par tous les sous-items, mais par deux seulement. Il va donc également falloir stocker pour chaque chaîne quels sont ces

sous-items. Ceci peut être fait en ajoutant un couple d'entiers à chaque chaîne, qui représente les deux sous-items.

```
type newchain = chain * int * int
type item = sitem list * newchain list
```

Cette modification entraine également une modification de l'ensemble des règles d'induction qui doivent alors regarder dans l'ensemble des chaînes qui peuvent composer un sous-item.

Quite à devoir modifier nos fonctions de la sorte, on peut alors imaginer modifier profondément la structure même des items pour qu'un sous-item ne contienne plus qu'une information sur la tête de l'arbre qu'il représente. Les autres chaînes qui permettent de représenter le reste de la structure des arbres seraient alors stockés dans un tableau de chaînes fourni avec l'item.

# Chapitre 5

# Conclusion

Un premier objet de ce travail était de présenter le formalisme des grammaires minimalistes, et d'introduire une reformulation des arbres qu'elles permettent de générer, dans l'optique de pouvoir décrire de façon précise un algorithme d'analyse de ces grammaires.

La première partie de ce mémoire nous a donc servi à définir tous les objets qui sont utilisés par l'algorithme, et à expliquer tous les termes utilisés dans les différents chapitres.

Un second travail à consister à présenter l'algorithme d'analyse dans les grammaires minimalistes défini par Harkema, dont le fonctionnement général découle du concept de l'analyse vue comme une déduction, et correspond à un parcours récursif en profondeur des arbres de dérivations d'une expression donnée. L'un des buts à la définition d'un tel algorithme et de pouvoir définir un analyseur "à la Earley", fonctionnant sur le même principe de parcours en profondeur, mais applicable à une plus grande partie des grammaires minimalistes.

On a vu qu'une version simple d'implémentation de l'algorithme d'analyse de Grammaires Minimalistes avec parcours en profondeur récursif soulevait déjà plusieurs questions, notamment en termes d'optimisation. Les modifications proposées dans la section 4.5.1 nous étant apparues comme très intéressantes, nous avons commencé à modifier notre implémentation pour qu'elle les prennent en compte. Malheureusement, ces modifications sont arrivées tardivement, et nous ne pouvons pas actuellement proposer une version de l'analyseur complète.

Harkema ([Har01]) propose une série d'améliorations à apporter à son algorithme pour en améliorer l'efficacité et pour le rendre plus proche encore du

fonctionnement du cerveau humain lors de l'analyse d'une phrase. Un travail intéressant serait donc logiquement d'ajouter ces nouvelles fonctionnalités à notre programme.

Bien sûr, d'autres travaux peuvent également être effectués à plus long terme, comme l'ajout d'un module permettant d'analyser sémantiquement les phrases considérées comme valides syntaxiquement par notre programme. Un tel travail a déjà été initié pour l'analyseur de Hale ([Amb03]).

# Bibliographie


[Abn87] S.P. Abney. The English noun phrase in its sentential aspect. 1987.

[Alb00] D. Albro. An Earley-style Recognition Algorithm for MCFGs. 2000.

[Alb02] D. Albro. An Earley-Style Parser for Multiple Context-Free Grammars. 2002.

[Amb03] M. Amblard. Representations semantiques pour les grammaires minimalistes. *Memoire de DEA, Universite Bordeaux 1*, 2003.

[AU72] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[BH53] Y. Bar-Hillel. A quasi arithmetical notation for syntactic description. *Language*, 1953.

[Bou98] P. Boullier. Proposal for a Natural Language Processing Syntactic Backbone. *Research Report*, 3342, 1998.

[Cho95] N. Chomsky. *The Minimalist Program*. MIT Press, 1995.

[Ear70] J.C. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the Association for Computing Machinery*, 13(2), 1970.

[Gui04] M. Guillaumin. Conversions between mildly context sensitive grammars. Master's thesis, University of California, Los Angeles, 2004.

[Hal03] J.T. Hale. *Grammar, Uncertainty and Sentence Processing*. PhD thesis, Johns Hopkins University, 2003.

[Har00] H. Harkema. A recognizer for minimalist grammars. *Sixth International Workshop on Parsing Technologies, IWPT*, 2000, 2000.

[Har01] H. Harkema. *Parsing Minimalist Languages*. PhD thesis, University of California, Los Angeles, California, 2001.

[JS97] A. Joshi and Y. Schabes. Tree-adjoining grammars. *Handbook of formal languages, vol.3 : beyond words*, pages 69–123, 1997.

[Kob06] G.M. Kobele. *Generating Copies : An investigation into structural identity in language and grammar*. PhD thesis, University of California, 2006.

[Lju02a] Peter Ljunglöf. Functional programming and NLP, January 2002. Term paper for the GSLT graduate course *Natural Language Processing*.

[Lju02b] Peter Ljunglöf. *Pure Functional Parsing – an advanced tutorial*. Licenciate thesis, Göteborg University and Chalmers University of Technology, April 2002.

[Lju04] Peter Ljunglöf. Grammatical Framework and multiple context-free grammars. In *FG'04, 9th Conference on Formal Grammar*, Nancy, France, August 2004.

[MADR06] B. Mery, M. Amblard, I. Durand, and C. Retoré. A Case Study of the Convergence of Mildly Context-Sensitive Formalisms for Natural Language Syntax : from Minimalist Grammars to Multiple Context-Free Grammars. 2006.

[Mic01a] J. Michaelis. Derivational minimalism is mildly context-sensitive. *Logical Aspects of Computational Linguistics (LACL'98), Lecture Notes in Artificial Intelligence*, 2014 :179–198, 2001.

[Mic01b] J. Michaelis. *On Formal Properties of Minimalist Grammars*. PhD thesis, Postdam, Germany, 2001.

[NSSW06] M. Namur, D. Schlachter, P. Schweitzer, and B. Wagler. Ajoût d'un module sémantique au parser de J. Hale. 2006.

[Pol84] C.J. Pollard. *Generalized Phrase Structure Grammars, Head Grammars, and Natural Language*. PhD thesis, Stanford University, Stanford, California, 1984.

[Rad97] A. Radford. *Syntactic Theory of the structure of English : A Minimalist approach*. Cambridge Textbooks in Linguistics, 1997.

[Rét07] C. Rétoré. Les Mathématiques de la Linguistique Computationnelle - I. *La Gazette des mathématiciens*, 2007.

[Ska98] C. Skalka. An implementation of Montague-style semantics in standard ML. *Introduction to Natural Language Processing*, 1998.

[SMFK91] H. Seki, T. Matsumara, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2) :191–229, 1991.

[SSP95] S.M. Shieber, Y. Schabes, and F.C.N. Pereira. Principles and Implementation of Deductive Parsing. *JLP*, 24(1&2) :3–36, 1995.

[Sta97] E. Stabler. Derivational minimalism. *Lecture Notes in Computer Science*, 1328 :68–96, 1997.

[Sta01] E. Stabler. Minimalist grammars and recognition. *Rohrer et al.(2001)*, 2001.

[VSWJ87] K. Vijay-Shanker, D.J. Weir, and A.K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. *Proceedings of the 25th conference on Association for Computational Linguistics*, pages 104–111, 1987.

[WJ88] D.J. Weir and A.K. Joshi. Characterizing mildly context-sensitive grammar formalisms. 1988.

[You67] D.H. Younger. Recognition and Parsinf of Context-Free Languages in $n^3$. *Information and Control*, 10(2), 1967.

# Annexe A

# Exemple de fonction

Voici une manière de coder la règle d'induction *Unmerge-1*. Les autres règles sont implémentées d'une façon similaire, excepté que chacune d'entre elles ne s'applique pas aux mêmes items. Les règles particulières *Unmerge-2* et *Unmerge-3* posent quelques questions supplémentaires, qui sont discutées dans la section 4.5.1.

La règle doit être appliquée indépendamment à chaque sous-item qui se trouve dans le domaine d'application. Dans cette optique, une sous-fonction *s-unmerg1* permet de vérifier qu'un sous-item donné peut être défussioné, et de créer les nouveaux sous-items correspondants lorsque c'est le cas.

La fonction *unmerge-1* doit donc finalement isoler chaque sous-item à tour de rôle pour les passer à la sous-fonction *s-unmerg1*, et de reconstruire des items complets à partir de la liste récupérée. Elle retourne la liste des items ainsi générés.

La liste d'items qui est retournée peut contenir des items qui sont déjà présents dans le tableau. C'est lors de la tentative d'ajout de chacun des items au tableau qu'un test de redondance va être effectué afin d'éviter la présence de doublons dans le tableau.

```
let unmerge1 item lexique =
  let rec s-unmergl i subitem lexical =
    match subitem with
      (p,q,sel::r,u)::s ->
        if i<(int_of_index p) then
          (s-unmergl (i + 1) subitem lexical)
        else if i>(int_of_index q) then
          []
        else
          (([(p,Position i,r,sel::u)],false)::
            [(Position i,q,(List.tl lexical),
             [(List.hd lexical)])::s,
              (match (List.tl lexical) with
                [] -> false
              | _ -> true)]):: 
              (s-unmergl (i + 1) subitem lexical)
    | _ -> [] in
  let newsubitems = function
    (((p,q,[(Select (Const f)) as tete],gamma)::s)
      as sitem,true) ->
        List.map
          (s-unmergl (int_of_index p) sitem)
            (lexfilter lexique tete)
    | _ -> [] in
  let newitems = ref [] in
  for i = 0 to (List.length item - 1) do
    newitems :=
      List.map
        (function x ->
          ((firstsi item (i - 1))
            @ x
            @ (lastsi item (i + 1))))
        (List.concat (newsubitems (List.nth item i)))
      @ !newitems
  done;
  !newitems;;
```