Contribution à la construction d'un système robuste d'analyse du français

Damien Genthial

Université Joseph Fourier
Science, Technologie, Médecine

THESE

présentée par

Damien GENTHIAL

pour obtenir le titre de

Docteur de l'Université Joseph Fourier - Grenoble I
(arrêté ministériel du 5 juillet 1984)

(Spécialité : **Informatique)**

Contribution à la construction d'un système robuste d'analyse du français

Soutenue le 10 Janvier 1991

Jury :

M. Christian Boitet	**(Président)**
M. Jacques Courtin	**(Directeur)**
M. Jacques-Henri Jayez	**(Rapporteur)**
M. Guy Pérennou	**(Rapporteur)**
M. Jacques Rouault	**(Examinateur)**

Thèse préparée dans l'équipe TRILAN
Laboratoire de Génie Informatique
Institut de Mathématiques Appliquées de Grenoble
Université Joseph Fourier

Avant-propos

Je tiens à exprimer ici ma gratitude à :

Monsieur Christian Boitet, Professeur à l'Université Joseph Fourier et Directeur Adjoint du Groupe d'Etudes pour la Traduction Automatique de Grenoble, qui m'a fait l'honneur de présider le jury ;

Messieurs Jacques-Henri Jayez, Professeur à l'Université de Nantes, et Guy Pérennou, Professeur à l'Université Paul Sabatier de Toulouse, qui ont accepté d'être rapporteurs de cette thèse ;

Monsieur Jacques Rouault, Professeur à l'Université des Sciences Sociales de Grenoble, qui a bien voulu participer au jury ;

Monsieur Jacques Courtin, Professeur à l'Université des Sciences Sociales de Grenoble et responsable de l'équipe TRILAN, qui m'a accueilli dans son équipe et m'a toujours accordé une grande confiance ;

Mesdames Irène Kowarski, Maître de Conférences à l'Université des Sciences Sociales de Grenoble, et Danièle Dujardin, Ingénieur CNRS, qui m'ont soutenu tout au long de ce travail et qui, grâce à leur expérience et leur compétence, m'ont permis d'éviter beaucoup d'écueils ;

Madame Véra Lucia Strube de Lima, Docteur en Informatique de l'Université Joseph Fourier, qui par sa gentillesse et son travail a beaucoup contribué à l'avancement des travaux présentés ici.

Les travaux présentés dans cette thèse ont été partiellement supportés par le GRECO-PRC Communication Homme-Machine, Pôle Langage Naturel.

A Jeannine, Christine, et Rachel.

Table des matières

Table des figures

Introduction générale

L'idée d'utiliser l'ordinateur pour manipuler la langue naturelle est à peu près aussi vieille que l'ordinateur lui-même. Les premiers travaux (fin des années 50 et dans les années 60) ont surtout porté sur la traduction automatique, mais dès cette époque on a tenté de détecter les erreurs dans les textes [DAMERAU 64].

L'échec des premiers travaux sur la traduction automatique et le développement de l'intelligence artificielle ont montré qu'un minimum de compréhension du texte par la machine était indispensable. Les années 70 ont vu le développement de théories permettant des manipulations formelles de la langue, manipulations indispensables pour approcher le sens. Le domaine dit de la *linguistique informatique* (computational linguistics) est né de ce développement.

Les années 80 se caractérisent par un important accroissement des travaux menés dans ce domaine, à tel point qu'on parle maintenant d'*industrie* de la langue. Cette montée en puissance s'explique par l'extraordinaire démocratisation de l'outil informatique dont l'origine est l'apparition du micro-ordinateur. Aux ordinateurs lourds, fragiles et coûteux ont succédé les ordinateurs personnels, machines tenant sur un bureau, très fiables et d'un coût à la portée de toutes les entreprises et de nombreux ménages. En France, le développement de la télématique et du Minitel ajoute encore à cette démocratisation : le terminal (écran + clavier) est devenu un outil majeur de communication. Cette démocratisation a entraîné un remplacement progressif du document sur support papier par un document sur support électronique. Ils sont rares aujourd'hui les magazines, les livres, les lettres, les notes de service qui n'ont pas été réalisés sur un ordinateur. Le bureau sans papier est en passe de devenir réalité. Les progrès des matériels (capacité de traitement et de stockage) ont été tels qu'on peut dorénavant imaginer de mettre en œuvre sur des ordinateurs de bureau des applications d'envergure considérable. On peut par exemple aujourd'hui, avec un simple micro muni d'un lecteur de disque optique numérique, accéder au contenu complet du Grand Robert en 9 volumes.

Mais la création et la manipulation de ces masses considérables de textes repose encore sur des moyens de saisie relativement fragile (saisie manuelle au clavier ou reconnaissance optique de caractères). Il est donc important de fournir des outils capables de vérifier la correction de la langue utilisée (tout bon traitement de textes est aujourd'hui muni d'un vérificateur orthographique). Nous allons essayer ici d'apporter notre contribution à la réalisation d'un système de détection/correction d'erreurs dans les textes écrits, capable de proposer de bonnes corrections et intégrant non seulement le

niveau lexical (correction orthographique) mais également le niveau syntaxique (la phrase).

Situation du travail

Les travaux présentés dans cette thèse ont été réalisés au sein de l'équipe TRILAN (TRaitement Informatique de la LAngue Naturelle). Cette équipe fait partie du LGI (Laboratoire de Génie Informatique) qui est un des laboratoires de l'IMAG (Institut des Mathématiques Appliquées de Grenoble).

Cette équipe s'est fixé pour but le développement d'une boîte à outils logiciels pour le traitement automatique du français. Le terme de boîte à outils est lié au désir de réaliser des modules logiciels qui soient utilisables par le plus grand nombre. Dans cette optique un effort particulier est accordé à la réalisation d'applications en grandeur réelle et à la portabilité de ces applications.

L'équipe TRILAN dispose d'outils pour le traitement morphologique du français (analyse et génération) qui s'appuient sur un lexique de 35000 racines et désinences permettant de reconnaître ou d'engendrer environ 250000 formes du français. L'équipe dispose également d'un constructeur de structures de dépendances (analyse syntaxique). Ces outils sont entièrement paramétrés et les paramètres peuvent être crées et modifiés grâce à des éditeurs interactifs.

Plus récemment, l'équipe TRILAN s'est fixé comme but le développement d'outils permettant la détection et la correction d'erreurs dans les textes écrits. De nombreux travaux ont été menés pour les erreurs de niveau lexical (mots inconnus). Ces travaux ont conduit à la mise en œuvre d'un module de correction par la méthode des clés de similarité et d'un module de correction par transcription phonétique. Parallèlement, des travaux ont porté sur la détection et la correction des erreurs de niveau syntaxique, et plus particulièrement les erreurs d'accord.

Buts poursuivis

Le travail présenté dans cette thèse s'insère dans les travaux de l'équipe TRILAN de deux façons :

- les travaux sur le niveau syntaxique ont donné lieu à la réalisation d'un prototype mais ils ont surtout montré la nécessité de développer des outils d'un niveau supérieur au niveau morphologique (outils syntaxiques puissants et outils sémantiques) ;
- les travaux sur le niveau lexical et les travaux sur le niveau syntaxique ont été mené séparément et il est nécessaire d'unifier les deux dans un système complet, cohérent, et qui répondent aux impératifs de l'équipe : portabilité et adaptabilité.

Le premier objectif, qui constitue l'essentiel du travail, a été de concevoir et de mettre en œuvre un outil d'analyse syntaxique capable de manipuler des informations syntaxiques complexes ainsi que des informations sémantiques. Compte tenu du deuxième objectif, il était nécessaire que cet outil soit tolérant vis à vis des erreurs.

Le deuxième objectif est de définir une architecture pour un système de détection/correction qui exploite de manière cohérente tous les outils dont nous disposons. La difficulté de cet objectif réside dans la définition des coopérations entre les différentes techniques et dans la conception d'une stratégie globale de correction.

Plan de la thèse

Cette thèse comprend deux parties correspondant aux deux objectifs indiqués ci-dessus.

Le chapitre 1 présente la problématique de l'analyse syntaxico-sémantique d'une langue naturelle à travers la description des notions de base et de quelques formalismes récents. Nous essayons de montrer quels sont les invariants de ces formalismes et comment ces invariants ont motivé nos choix.

Le chapitre 2 décrit le constructeur de structures de dépendances que nous proposons. On y montre les apports d'une hiérarchie de catégories à la souplesse et à la tolérance de l'analyse. Les arbres de dépendances produits sont décorées grâce à un formalisme de représentation de la connaissance qui est présenté au chapitre 3.

Le chapitre 4 termine la première partie en présentant le prototype que nous avons développé pour tester les concepts des chapitres précédents.

Le chapitre 5 traite de la détection/correction d'erreurs au niveau lexical et souligne comment le formalisme de représentation de la connaissance décrit au chapitre 3 peut être intégré aux outils existants.

Après avoir décrit les objectifs fixés pour la détection/correction d'erreurs au niveau syntaxique, le chapitre 6 donne un aperçu des outils développés dans l'équipe et souligne les apports des concepts et outils de la première partie à la robustesse des traitements.

Enfin, nous proposons au chapitre 7 l' architecture d'un système complet de détection/correction d'erreurs dans un texte écrit. Nous insistons sur la portabilité et l'adaptabilité du système.

Partie I

Analyse syntaxico-sémantique

1

Description et construction de structures syntaxiques

Les premiers travaux importants visant à la description formelle des langues naturelles sont dus à Chomsky [CHOMSKY 57] et ont donné naissance à la théorie des grammaires syntagmatiques, appelées aussi grammaires de constituants ou grammaire à structure de phrase (traduction de l'anglais Phrase Structure Grammars). Dans ce chapitre, nous décrivons les concepts fondamentaux associés à ces grammaires ; il ne s'agit pas de donner une description exhaustive et détaillée de tous les formalismes, pas plus que de citer tous les systèmes informatiques s'appuyant sur ces théories et leurs dérivées, mais plutôt de souligner les considérations linguistiques qui ont motivé les évolutions successives et de montrer les implications pratiques des changements théoriques. [WINOGRAD 83] et [SABAH 88] sont deux excellents ouvrages de synthèse présentant les travaux effectués dans le domaine du Traitement Automatique de la Langue Naturelle (TALN). Le premier met l'accent sur la syntaxe et sur la mise en œuvre des théories successives ; le deuxième aborde de manière plus large les phénomènes liés à la communication homme machine en langue naturelle : il décrit les approches formelles de la syntaxe mais aussi les problèmes liés à la compréhension de la langue par la machine (aspects sémantiques, contextuels et pragmatiques) et les problèmes liés aux situations de communication (dialogue).

Dans tout ce chapitre (et dans toute la thèse), nous considérons connus du lecteur les éléments fondamentaux de la théorie des langages formels. Le lecteur non familier de ces notions trouvera en annexe A la définition des éléments que nous utilisons ici.

1.1. Analyse Hors-Contexte

Les travaux les plus nombreux ont d'abord porté sur les grammaires dites de type 2, ou grammaires hors contexte (Context Free). En effet, leur facilité de mise en œuvre informatique et leur relatif pouvoir d'expression en ont fait l'outil formel de base pour la compilation des langages de programmation. Les travaux présentés ici ne sont d'ailleurs pas seulement issus du domaine du TALN mais aussi des recherches sur la compilation des langages de programmation.

Une grammaire hors contexte G définie par le quadruplet (V_T, V_N, S, P) est une grammaire *générative*. Les chaînes du langage L(G) décrit par la grammaire sont toutes les chaînes de V_T^* obtenues par dérivation à partir de S, en appliquant les règles de production de P. La structure syntaxique d'une chaîne est l'arbre de dérivation de cette chaîne, c'est-à-dire une arborescence dans laquelle tous les nœuds internes sont étiquetés par des symboles de V_N et toutes les feuilles par des symboles de V_T ou par la chaîne vide, notée ε.

Exemple de grammaire : voir figure 1.1

```
Le "+" dénote la concaténation des chaînes
La "|" dénote une alternative

G = ( { le, la, bébé, bouillie , mange },
      { S, GV, GN, Det, Nom, Verbe },
      S,
      { S -> GN + GV
        GN -> Det + Nom
        GV -> Verbe + GN
        Nom -> bébé | bouillie
        Det -> la | le
        Verbe -> mange } )
```

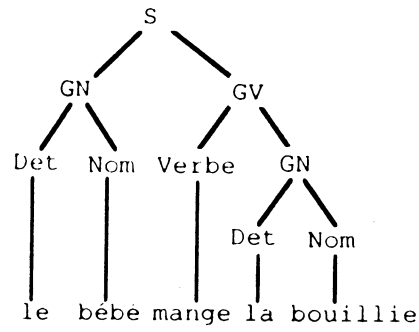


Figure 1.1 : Exemple de grammaire simple et d'arbre de dérivation

La grammaire elle-même constitue une description **statique** du langage. Elle peut être utilisée de manière **dynamique** :

- en génération, pour produire toutes les chaînes grammaticalement correctes, ou une chaîne particulière ;
- en analyse, pour indiquer si une chaîne est correcte ou non du point de vue de la grammaire (le processus mis en œuvre est appelé un *accepteur*), ou pour produire la (ou les) structure syntaxique associée à la chaîne (le processus est appelé *analyseur*).

Les deux processus (accepteur et analyseur) ont en général un noyau commun : l'analyseur est un accepteur auquel on a ajouté les actions de construction de la structure (arbre de dérivation). Notre principale motivation étant la construction de structures syntaxiques, nous nous placerons toujours au niveau de l'analyseur.

1.1.1. Schémas classiques

1.1.1.1. Analyse descendante (ou prédictive)

Le principe est le suivant : on construit l'arbre syntaxique d'une phrase en partant de la racine (un nœud étiqueté par l'axiome S de la grammaire) et en

descendant vers les feuilles (les symboles terminaux correspondant aux symboles de la chaîne à analyser). On peut choisir de développer l'arbre par la gauche d'abord ou par la droite d'abord, nous décrivons ici l'analyse gauche qui permet de parcourir la chaîne à analyser de la gauche vers la droite, c'est-à-dire dans le sens naturel de lecture.

Un état de l'analyseur est caractérisé par :

- l'arbre syntaxique déjà construit dans lequel un nœud particulier (dit *nœud courant*) est repéré ; en cours d'analyse, ce nœud est toujours une feuille de l'arbre partiel construit.
- la chaîne à analyser partagée en deux parties : la sous-chaîne déjà reconnue et la sous-chaîne restant à analyser ; le premier symbole de la sous-chaîne restant à analyser est dit *symbole courant.*

Le passage d'un état i à l'état i+1 s'effectue comme suit :

si le nœud courant est étiqueté par un symbole terminal alors
    si ce symbole est identique au symbole courant alors
        la feuille qui suit le nœud courant dans le mot des feuilles (liste des feuilles de l'arbre de la gauche vers la droite) devient le nœud courant, le symbole courant passe dans la sous-chaîne déjà analysée ;
    sinon
        échec de l'état i ;
sinon le nœud courant est étiqueté par un symbole non-terminal :
    déterminer l'ensemble des règles de la grammaire ayant ce symbole en partie gauche ;
    choisir une des règles, créer un fils du nœud courant pour chaque symbole de sa partie droite ;
    parmi les fils ainsi crées, le plus à gauche devient le nœud courant.

Nous expliquons plus bas (§1.1.1.3) comment sont traités les problèmes de l'échec et du choix d'une règle, nous nous contentons ici de donner un exemple simple pour éclairer le schéma d'algorithme ci-dessus. On trouvera figure 1.2 l'évolution commentée de l'état de l'analyseur pour la phrase "le bébé mange la bouillie" avec la grammaire de la figure 1.1.

1.1.1.2. Analyse ascendante

Le principe est ici de construire l'arbre en remontant des feuilles (c'est-à-dire des terminaux) vers la racine. On compare un morceau de la chaîne à analyser avec les parties droites des règles, s'il y a correspondance, on crée un nœud étiqueté par le symbole de la partie gauche et on rattache tous les éléments en correspondance comme des fils de ce nœud. On aura une analyse correcte si on a lu tous les symboles de la chaîne à analyser et si la racine de l'arbre est étiquetée par l'axiome S. On peut ici aussi parcourir la chaîne de gauche à droite ou de droite à gauche, nous choisissons encore la première solution.

La structure mise en œuvre est une forêt (liste d'arbres) vide au départ. A chaque étape on a deux types d'actions :

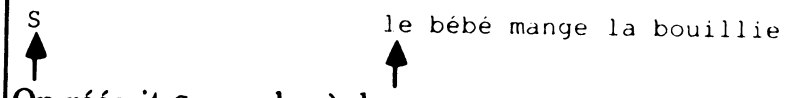
- *réduction* : s'il existe une règle de la grammaire telle que les symboles de sa partie droite correspondent aux symboles portés par les racines des arbres de la tête de la forêt alors remplacer ces arbres

par un arbre unique dont la racine est étiquetée par le symbole de la partie gauche de la règle et dont les fils sont les arbres sélectionnés par la partie droite ;

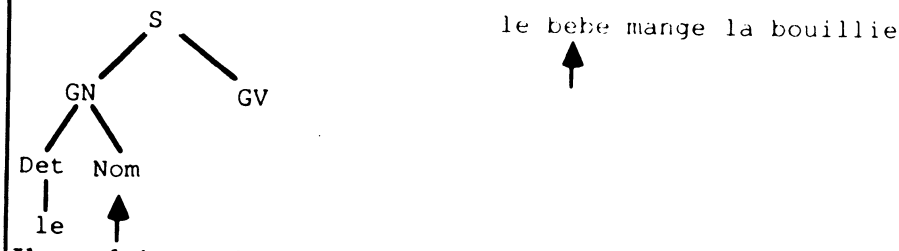
- *avance* : si aucune réduction n'est applicable, ajouter à la forêt un arbre à un seul nœud étiqueté par le symbole courant de la chaîne à analyser et avancer sur le symbole suivant de la chaîne à analyser.

La forêt se comporte comme une pile car on ne recherche les occurrences de parties droites de règle qu'en début de liste.

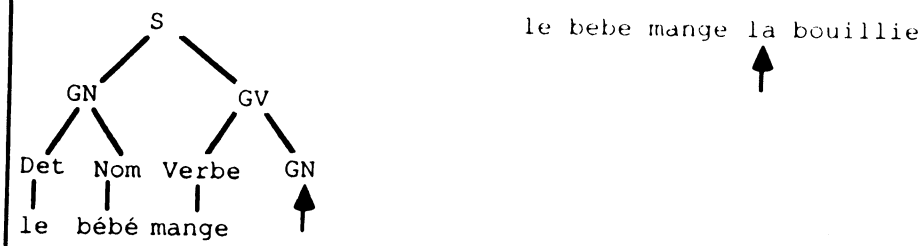
Les nœud et symbole courants sont repérés par une flèche.
L'état initial est le suivant :



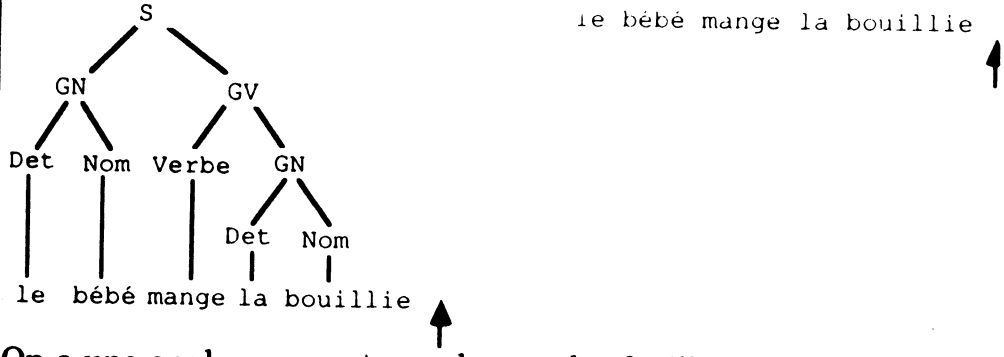
On réécrit S avec la règle S -> GN + GV, puis le symbole le plus à gauche : GN est réécrit Det + Nom, Det est réécrit "le". On vérifie alors que le symbole terminal obtenu : "le", est identique au symbole courant, comme c'est le cas on avance dans la chaîne d'entrée et dans le mot des feuilles de l'arbre et on a l'état :



Il est clair que le choix de la règle Det -> la au lieu de la règle Det -> le aurait conduit à un échec. L'analyse se poursuit par la réécriture du Nom (pour lequel deux choix sont possibles), puis du GV et du Verbe contenu dans le GV, on a alors l'état :



Le traitement du GN permet d'obtenir l'état final suivant :



On a une analyse correcte car le mot des feuilles et la chaîne à analyser sont devenus vides simultanément.

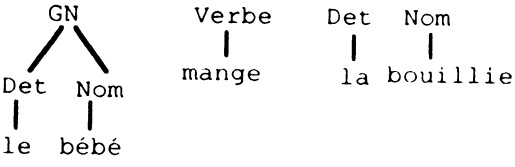
Figure 1.2 : Evolution de l'analyse descendante.

Après avoir avancé sur "le" puis réduit grâce à la règle Det -> le (on note qu'ici le problème du choix de "le" ou "la" ne se pose pas puisqu'on considère les parties droites des règles), on a la forêt :

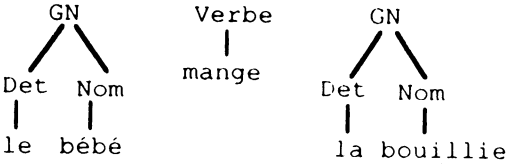
On doit avancer sur "bébé" puis réduire avec Nom -> bébé, on a alors la forêt :

qui se réduit avec la règle GN -> Det + Nom pour donner :

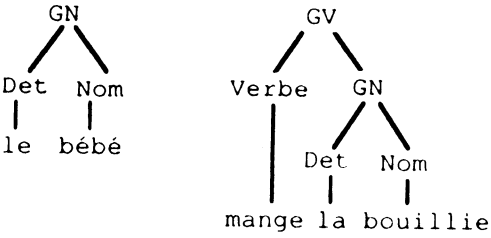
On doit ensuite avancer sur le verbe mais aussi sur tout le reste de la chaîne jusqu'à obtenir :



avant de pouvoir appliquer la réduction suivante en GN :



puis en GV :



et enfin en S :

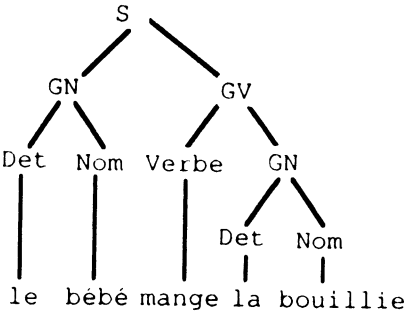


Figure 1.3 : Evolution de la forêt dans l'analyse ascendante.

On peut remarquer que les problèmes de choix qui se posaient pour l'analyse descendante se retrouvent partiellement ici : on peut avoir plusieurs réductions applicables. La détection d'un échec ne peut se faire qu'après avoir parcouru toute la chaîne à analyser. On donne figure 1.3 la table d'évolution de la forêt pour la même phrase et la même grammaire qu'au paragraphe précédent.

1.1.1.3. Problème de l'ambiguïté

Les deux schémas présentés ci-dessus sont non-déterministes, c'est-à-dire qu'en présence d'une ambiguïté (symbole pouvant se réécrire de plusieurs façons dans l'analyse descendante, multiple réduction dans l'analyse ascendante), ils effectuent le choix arbitraire d'une des alternatives.

Dans le cadre de l'analyse de langages formels comme les langages de programmation, on peut rendre l'analyseur déterministe (aucun choix à faire) en lui donnant la possibilité de regarder en avant k symboles de la chaîne à analyser. Plus k est grand, moins l'analyse est efficace, on choisit donc le plus petit k qui rend l'analyseur déterministe. On parle alors d'analyse LL(k) pour la méthode descendante et LR(k) pour la méthode ascendante (voir l'annexe A et [KNUTH 65, AHO&ULL 72, 77].

Mais ces méthodes ne sont pas applicables aux langues naturelles car ces dernières sont intrinsèquement ambiguës, c'est-à-dire qu'on ne peut donner une grammaire hors contexte non ambiguë décrivant la langue. Comparer par exemple les deux phrases "le bébé mange la bouillie à la crème" et "le bébé mange la bouillie à la cuiller". L'examen de k symboles de la chaîne à analyser n'est pas non plus possible à cause de l'adjonction toujours possible de chaînes de longueur quelconque entre deux éléments d'une phrase, comparer :
"le bébé mange la bouillie ..."
"le bébé de la voisine mange la bouillie ..."
"le bébé de la voisine qui a eu la rougeole la semaine dernière mange maintenant la bouillie avec bon appétit ..."

On peut remarquer qu'une des principales sources d'ambiguïté dans la méthode descendante est la réécriture des symboles pré-terminaux. Ces symboles sont les éléments de V_N qui ne se réécrivent qu'en un symbole de V_T. Ces symboles sont appelés catégories (ou classes) lexico-syntaxique des mots car chacun caractérise une classe de mots de la langue ayant le même comportement syntaxique. Pour des raisons d'efficacité, tous les systèmes de traitement de la langue possèdent un module chargé d'associer à un mot sa catégorie ; ce module est appelé analyseur morphologique (pour les analyseurs ascendants) ou module de réalisation lexicale (pour les analyseurs descendants). Nous traitons plus en détail ces concepts au §1.3 et dans la deuxième partie de la thèse. Nous considérons ici que chaque mot peut se voir attribuer une classe et les symboles terminaux de la grammaire ne sont plus les mots de la langue mais les catégories lexicales de ces mots. Notons que cela n'élimine pas toutes les ambiguïtés puisque de nombreux mots peuvent appartenir à plusieurs classes[1] (on parle d'homographies).

[1]Certains groupes de mots peuvent être considérés comme un seul et avoir une classe, c'est le cas de locutions comme "à ce propos", "au fur et à mesure", ...

Exemple :
"ferme" peut être un adjectif dans "une chair ferme", un nom dans "la ferme du père Peinard", un verbe dans "il ferme la porte", ou un adverbe dans "il neigeait ferme".

Un analyseur de la langue naturelle doit donc être capable de manipuler les ambiguïtés, c'est-à-dire de représenter dans son formalisme de sortie les multiples interprétations associées à une même chaîne d'entrée. La méthode la plus utilisée consiste à énumérer ces interprétations, c'est à dire à construire un arbre syntaxique pour chacune d'elle. Deux techniques peuvent être appliquées aux schémas classique présentés ci-dessus :

- développer en parallèle toutes les solutions possibles et ainsi construire systématiquement tous les arbres associés à une chaîne et une grammaire données. Les inconvénients sont le risque d'explosion combinatoire dans le cas de phrases très ambiguës et le développement de nombreux arbres qui n'aboutiront pas à une analyse correcte.
- utiliser une pile qui permet à chaque point de choix de mémoriser les alternatives ; chaque fois que l'analyse échoue, revenir en arrière (backtrack) jusqu'au dernier point de choix pour essayer l'alternative suivante. On peut ainsi arrêter l'analyse à la première solution trouvée (on n'obtient alors qu'un seul arbre), ou mémoriser les arbres corrects construits et revenir systématiquement en arrière pour examiner les alternatives restantes. Les inconvénients sont ici la complexité ajoutée par les méthodes de mémorisation et le risque d'effectuer plusieurs fois la même analyse d'un morceau de la chaîne d'entrée parce que ce morceau se trouve après un point de choix.

Nous ne détaillerons pas plus avant ces deux techniques mais nous présentons dans les deux paragraphes suivants deux solutions proposées qui essayent de dépasser les inconvénients cités.

1.1.2. La méthode des cartes ("charts")

Cette méthode s'inspire des techniques de programmation dynamique : l'idée est de conserver dans une structure de travail la trace des analyses déjà effectuées de manière à éviter les calculs redondants liés à la méthode du retour arrière. Cette structure est appelée *table des sous-chaînes bien formées* ou *carte* (en anglais *chart*).

1.1.2.1. Description statique : structure d'une carte

Une carte est un graphe ou les sommets représentent une position dans la chaîne à analyser ou plus précisément l'intervalle entre deux symboles successifs de cette chaîne ; les arêtes (les arcs) sont étiquetés par un symbole de la grammaire. Au départ, seules les arêtes liant deux sommets adjacents apparaissent, chaque arête portant une des catégories lexico-syntaxiques du mot correspondant. On trouvera un exemple de carte sur la figure 1.4 : on a fait apparaître l'ambiguïté lexicale des mots "le" et "la" qui sont à la fois pronoms et déterminants, ainsi que l'ambiguïté de "bouillie", à la fois nom et participe passé. Les arêtes GN liant les sommets 1 et 3 d'une part, et les sommets 4 et 6

d'autre part, indiquent qu'un GN a été reconnu sur la portion de phrase comprise entre les sommets correspondants.

Carte pour l'analyse (partielle) de la phrase : "le bébé mange la bouillie"

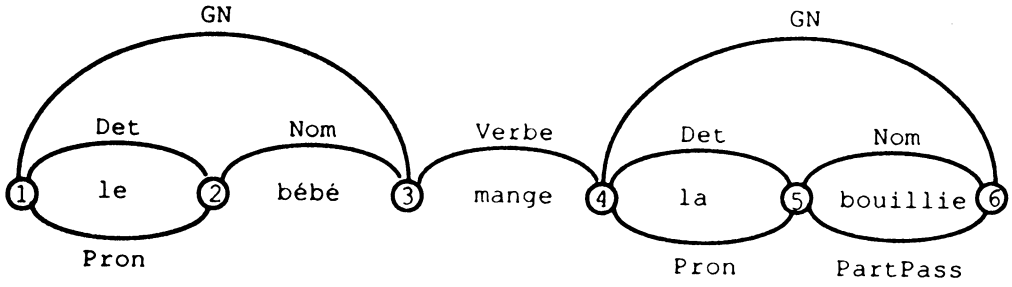
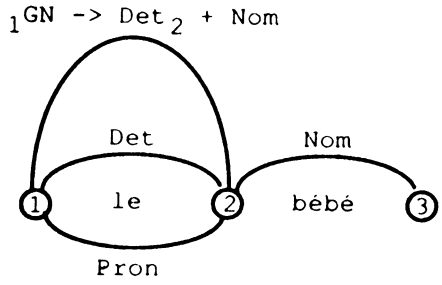


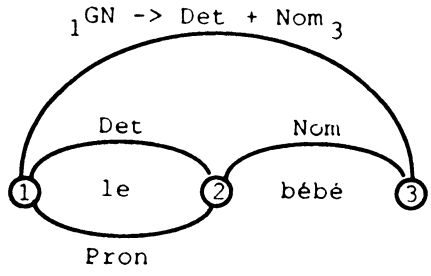
Figure 1.4 : Exemple de Carte

Pour permettre la construction des arbres syntaxiques, les arêtes doivent porter, en plus d'un symbole de la grammaire, la décomposition de ce symbole. Chaque arête porte donc la règle de la grammaire qui a permis de la créer.
En cours d'analyse, on distingue les arêtes *partielles* et les arêtes *complètes*. Une arête est partielle si on n'a pas encore complètement reconnu la partie droite de la règle qu'elle porte. Une arête portant un symbole terminal est évidemment complète. Dans ce qui suit on notera une arête avec la règle qu'elle porte à laquelle on ajoute : à gauche le numéro du sommet de départ et après le dernier symbole reconnu de la partie droite, le numéro du sommet d'arrivée[1].
Exemple :

Après avoir partiellement reconnu la partie droite de la règle GN -> Det + Nom, on aura la carte :



Dans une arête complète, le numéro du nœud d'arrivée est à droite de la règle comme sur la carte :



[1]Notation empruntée à [WINOGRAD 83]

1.1.2.2. Description dynamique : algorithme d'analyse

En plus de la carte elle-même, l'analyseur maintient une liste des arêtes à traiter qui est un sous ensemble de l'ensemble des arêtes de la carte. La méthode de gestion de cette liste (comme une file, comme une pile, comme une file d'attente avec priorités) détermine la stratégie d'analyse (largeur d'abord, profondeur d'abord, heuristique). L'algorithme d'analyse est schématiquement le suivant :

Initialiser la carte avec une arête complète pour chaque mot. Si un mot a plusieurs catégories, créer autant d'arêtes que de catégories (cf "le" qui est à la fois pronom et déterminant sur la figure 1.4). Pour chaque règle de la grammaire ayant l'axiome S en partie gauche créer une arête partielle :

$_1$S -> $_1$Partie Droite

et ajouter ces arêtes partielles à la liste des arêtes à traiter.

Tant que la liste des arêtes à traiter (LAT) n'est pas vide :

Supprimer la première arête de la LAT, elle devient l'arête courante (AC).

Si l'AC est complète, essayer de compléter (voir ci-dessous) chaque arête partielle dont le sommet d'arrivée est le sommet de départ de l'AC avec l'AC.

Si l'AC est partielle :

Essayer de la compléter avec chaque arête complète dont le nœud de départ est le nœud d'arrivée de l'AC.

Si aucune arête complète ne peut compléter l'AC et si le premier symbole restant à analyser dans la partie droite de la règle portée par l'AC est un non-terminal X alors :

Pour chaque règle de la grammaire ayant X en partie gauche ajouter une arête partielle de la forme

$_i$X -> $_i$PartieDroite où i est le numéro du nœud d'arrivée de l'AC.

Ajouter ces arêtes à la LAT.

Pour compléter une arête partielle P avec une arête complète C, on doit avoir :

$P = {}_i X \rightarrow \alpha\ {}_j Y\ \beta$ et $C = {}_j Y \rightarrow \gamma_k$ avec $X, Y \in V_N$, $a, b \in V^*$, $g \in V^+$

On complète en ajoutant à la LAT la nouvelle arête :

${}_i X \rightarrow \alpha\ Y_k\ \beta$

Note : A chaque fois qu'on crée une nouvelle arête, on s'assure qu'elle n'est pas déjà dans la carte, sinon on ne la crée pas ; on évite ainsi de construire plusieurs fois la même structure.

La gestion de la liste des arêtes à traiter permet de définir la stratégie de l'analyseur. Ainsi si cette liste est gérée comme une file FIFO[1], l'analyseur se comporte comme un analyseur descendant parallèle : il crée toutes les possibilités d'analyse en un point donné avant de les explorer. Si la liste est gérée comme une pile LIFO[2], l'analyseur va au contraire explorer à fond une hypothèse avant d'examiner les hypothèses suivantes. Le comportement est celui d'un analyseur descendant avec retour arrière. On peut dans ce cas là

[1]First In First Out (Premier Entré, Premier Sorti)

[2]Last In First Out (Dernier Entré, Premier Sorti)

envisager d'arrêter l'analyse dès qu'une solution est trouvée : une arête complète portant une règle dont la partie gauche est S, dont le nœud de départ est 1 et le nœud d'arrivée le dernier nœud de la chaîne à analyser. On peut enfin utiliser une stratégie mixte en utilisant des heuristiques qui permettent de trier la liste des arêtes à traiter.

La méthode des cartes, outre qu'elle est efficace, offre donc une grande souplesse dans la stratégie d'analyse. Elle a été, et est encore, largement utilisée dans les systèmes de TALN comportant un noyau hors-contexte. Citons par exemple le célèbre Earley [EARLEY 70] qui utilise un ordre fixe pour la liste des arêtes à traiter et qui permet l'examen des symboles de la chaîne à analyser (mécanisme de prédiction ou de droit de regard en avant). Kaplan [KAPLAN 73] a étendu l'usage des cartes aux réseaux de transition et plus récemment [LALLICH-BOIDIN 86] a mis en œuvre un analyseur dérivé de celui de Earley.

1.1.3. L'analyseur ascendant de Tomita

L'idée de Tomita [TOMITA 84a, 87, 88b] est d'utiliser un analyseur LR pour traiter le langage naturel. Les analyseurs LR sont en effet les analyseurs les plus efficaces pour les langages de programmation, mais ils ne peuvent être utilisés directement pour l'analyse des langues naturelles car ils s'appuient sur des grammaires ayant des propriétés que n'ont pas les grammaires des langues naturelles : ils ne sont notamment pas capable de traiter des grammaires ambiguës.

Tomita propose une extension de ces analyseurs capable de manipuler les ambiguïtés, d'intégrer les homographies (ambiguïté lexicale : un mot a plusieurs catégories) et même de manipuler des mots inconnus. Son analyseur tente de conserver toute l'efficacité des analyseurs LR et d'éviter toute redondance dans le processus de construction de structures et dans les structures elles-mêmes.

1.1.3.1. Analyse LR

Nous nous contentons ici d'une description rapide destinée à éclairer la suite, le lecteur intéressé peut se référer à l'abondante littérature consacrée à ce sujet (notamment [KNUTH 65], [AHO&ULL 72, 77]) ou à l'annexe A où on décrit une méthode de construction des tables. C'est cette méthode qui est utilisée pour construire les tables données dans les exemples.

Un analyseur LR est un analyseur ascendant du type décrit au §1.1.1, mais utilisable uniquement pour une certaine classe de grammaires hors-contextes, dîtes grammaires LR (ou LR(k)). L'idée est de tirer parti des propriétés de ces grammaires pour construire un automate à pile déterministe qui donne toute son efficacité à l'analyseur. A partir d'une grammaire on construit automatiquement une table de transition qui sera interprétée par l'automate. On trouvera figure 1.5 un exemple de grammaire et la table de transition associée.

La table comporte deux parties : la table de transition proprement dite (colonnes étiquetées par des terminaux et par $ qui désigne la fin de la chaîne à analyser), et une table des sauts (colonnes étiquetées par les non-terminaux).

La partie transition contient 4 types d'actions :

- `Erreur` : c'est l'action par défaut (non matérialisée sur l'exemple),
- `Succès` : indique que la chaîne à analyser est correcte,
- `Emp k` : empiler le symbole et l'état courant, l'état courant prend la valeur k,
- `Re n` : réduire la pile en utilisant la règle de numéro n.

```
1 : S -> GN + GV
2 : GN -> Det + Nom
3 : GV -> Verbe + GN
```

Etat	Symboles terminaux				Non-terminaux		
	Det	Nom	Verbe	$	S	GN	GV
0	Emp 3				1	2	
1				Succès			
2			Emp 4				5
3		Emp 6					
4	Emp 3					7	
5				Re 1			
6			Re 2	Re 2			
7				Re 3			

Figure 1.5 : Table de transition

L'analyseur LR utilise cette table, la grammaire et la chaîne à analyser ; il maintient un numéro d'état courant, un pointeur sur le symbole courant de la chaîne à analyser et une pile de paires (numéro d'état, arbre construit). L'algorithme d'analyse est le suivant :

Initialiser `EtatCourant` à 0, `Pile` à vide, `SymboleCourant` avec le premier symbole de la chaîne à analyser, ajouter le symbole `$` à la chaîne à analyser.

Tant que ni `Succès` ni `Erreur` faire :

En fonction de la ligne `EtatCourant` et de la colonne `SymboleCourant` de la table de transition :

- `Erreur` : on ne fait rien (arrêt de l'analyse)
- `Succès` : idem
- `Emp k` :
  créer un arbre à un seul nœud étiqueté par `SymboleCourant` ;
  empiler la paire (`EtatCourant`, arbre crée) ;
  `SymboleCourant` devient le symbole suivant de la chaîne ;

EtatCourant devient k.

- Re n :
  créer un nœud étiqueté par le symbole (nommé ici X) de la partie gauche de la règle de numéro n,
  pour chaque symbole de la partie droite de cette règle : dépiler deux fois (l'arbre puis l'état) et ajouter l'arbre dépilé en tête de la liste des fils du nœud crée ; on appelle D le dernier état dépilé ;
  empiler la paire (D, arbre construit)
  Si la ligne D colonne X de la table contient un numéro d'état, EtatCourant devient cet état, sinon Erreur.

Si Succès alors l'élément en sommet de pile contient l'arbre syntaxique de la chaîne à analyser.

Exemple d'évolution de la pile :

On donne ci-dessous l'évolution correspondant à l'analyse de la chaîne Det + Nom + Verbe + Det + Nom ("le bébé mange la bouillie"). Compte tenu de la simplicité de l'exemple, on donne les arbres sous forme parenthésée (préfixée). La pile est représentée sous forme d'une liste alternant les états et les arbres et on ajoute le numéro de l'état courant à la suite du sommet de pile.

Après analyse de Det + Nom on a la situation :

Pile	**Chaîne restant à analyser**
0-Det-3-Nom-6	Verbe+Det+Nom+$

La consultation de la ligne 6, colonne Verbe de la table décide de l'application d'une réduction utilisant la règle 2 (Re 2). On crée donc un arbre dont la racine porte GN et on dépile Nom qui est ajouté en tête de la liste des fils de GN, puis Det qui subit le même traitement. Le dernier état dépilé est l'état 0, on consulte donc la ligne 0 colonne GN de la table des sauts qui permet d'obtenir la nouvelle situation :

0-GN(Det,Nom)-2	Verbe+Det+Nom+$

On note ci dessous les situations avec à droite les actions effectuées :

Pile	**Chaîne restant à analyser**	**Action**
0-GN(Det,Nom)-2	Verbe+Det+Nom+$	Emp 4
0-GN(Det,Nom)-2-Verbe-4	Det+Nom+$	Emp 3

On va empiler successivement le Det et le Nom qui suivent le Verbe puis réduire Det+Nom en GN avec la règle 2, on a alors :

0-GN(Det,Nom)-2-Verbe-4-GN(Det,Nom)-7	$	Re 3
0-GN(Det,Nom)-2-GV(Verbe,GN(Det,Nom))-5	$	Re 1
0-S(GN(Det,Nom),GV(Verbe,GN(Det,Nom)))-1	$	Succès

L'état 1 avec le symbole $ est un état de succès et la pile porte le résultat de l'analyse.

1.1.3.2. Pile structurée

L'ambiguïté des langues naturelles se traduit dans le schéma d'algorithme ci dessus de deux façons :

- apparition dans la table d'entrées portant plusieurs actions ;
- apparition dans la chaîne à analyser d'un mot ayant plusieurs catégories, entraînant l'utilisation de plusieurs colonnes de la table.

Dans les deux cas, il faut introduire une forme de non déterminisme, soit par des analyses parallèles, soit par des retours arrières (cf §1.1.1.3).

On trouvera figure 1.6 une grammaire ambiguë et la table de transition associée. La construction de cette table à partir de la grammaire est détaillée dans l'annexe A, §A.3.2.

Grammaire :
1 : S -> GN + GV
2 : S -> S + GP
3 : GN -> Nom
4 : GN -> Det + Nom
5 : GN -> GN + GP
6 : GP -> Prep + GN
7 : GV -> Verbe + GN

Etat	Det	Nom	Verbe	Prep	$	S	GN	GV	GP
0	Emp 3	Emp 4				1	2		
1				Emp 6	Succès				5
2			Emp 7	Emp 6				8	9
3		Emp 10							
4			Re 3	Re 3	Re 3				
5				Re 2	Re 2				
6	Emp 3	Emp 4					11		
7	Emp 3	Emp 4					12		
8				Re 1	Re 1				
9			Re 5	Re 5	Re 5				
10			Re 4	Re 4	Re 4				
11			Re 6	Re 6,Emp 6	Re 6				9
12				Re 7,Emp 6	Re 7				9

Figure 1.6 : Grammaire ambiguë et table de transition associée

Tomita propose l'utilisation d'une technique de programmation dynamique qui permet de construire toutes les structures associées à une chaîne ambiguë en évitant de dupliquer les sous-structures communes et en factorisant les traitements. L'idée est d'utiliser une technique parallèle, c'est-à-dire de créer autant de processus d'analyse que d'alternatives, mais d'y ajouter un mécanisme de synchronisation qui fusionne les processus se trouvant dans le même état et sur le même symbole de la chaîne à analyser. Cette synchronisation est guidée par les actions d'empilage (qui font avancer dans la chaîne) : un processus ne peut avancer sur un symbole que si les autres processus avancent aussi sur ce symbole. Tous les processus exécutent donc simultanément l'action qui transfère un symbole X de la chaîne sur la pile ; si après ce transfert deux processus sont dans le même état de la table, ils sont fusionnés (puisque leur comportement à venir sera identique), on factorise ainsi les calculs qui restent à faire. Comme les piles des processus fusionnés ne sont pas nécessairement identiques, la pile résultante n'aura pas une structure linéaire mais une structure arborescente avec le sommet à la racine

de l'arbre (ce sommet portant l'état commun). De même, si en dépilant un état on obtient plusieurs piles (les fils de cet état), le processus est dupliqué et chaque copie se voit affecter une des piles obtenues.

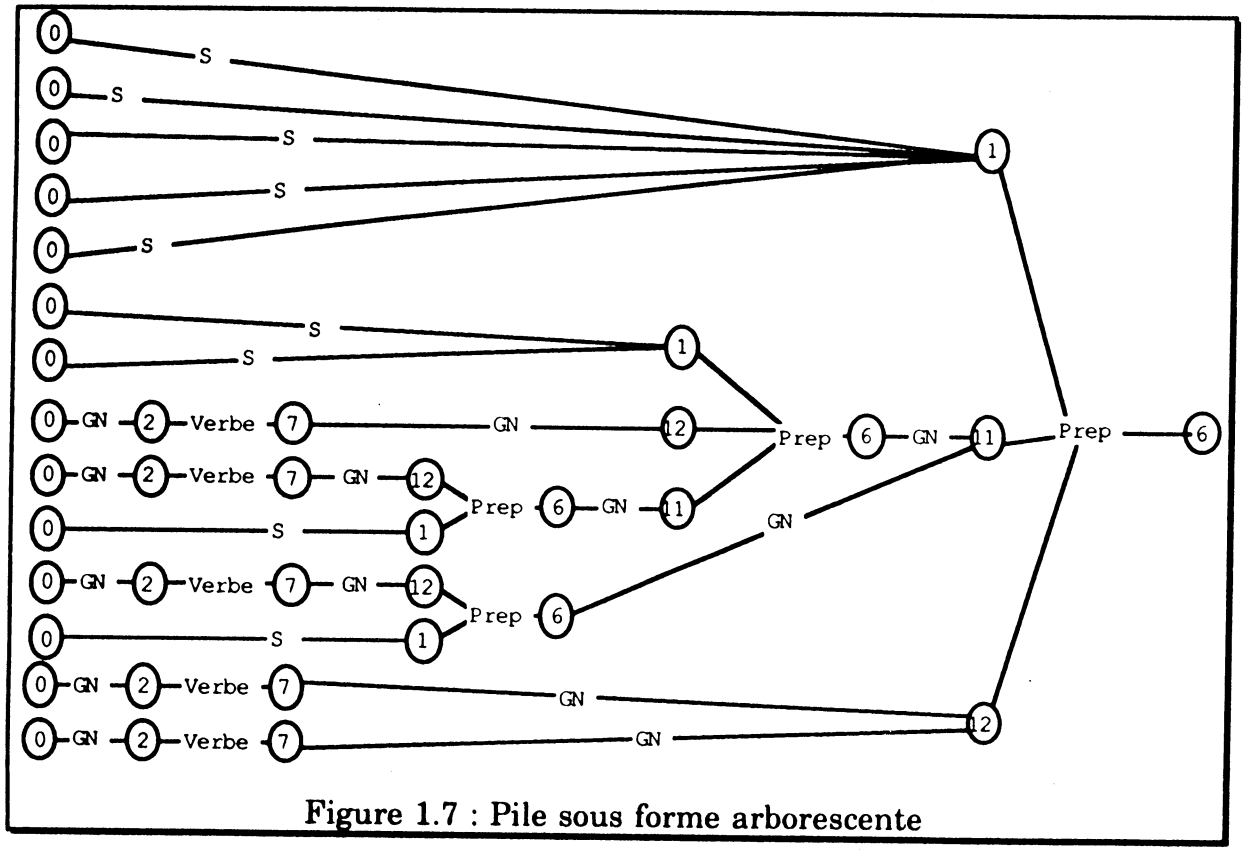


Figure 1.7 : Pile sous forme arborescente

Exemple :

Avec la grammaire et la table de la figure 1.6, on effectue l'analyse de la phrase "Marie étudie l'anglais à l'école de commerce de Paris" correspondant à la séquence de catégories :

`Nom + Verbe + Det + Nom + Prep + Det + Nom + Prep + Nom + Prep + Nom`

Après analyse de "Marie étudie l'anglais" on a la pile :

`0-GN(Nom)-2-Verbe-7-GN(Det, Nom)-12`

avec `Prep` comme symbole courant.

On a alors deux actions possibles (ligne 12, colonne `Prep` de la table) : `Re 7` et `Emp 6`. On duplique donc le processus : une version est chargée d'appliquer `Re 7` et l'autre `Emp 6`, ce dernier est gelé jusqu'à ce que le premier effectue aussi l'action d'empilage de `Prep`. Après la réduction, on a donc les deux piles :

`0-GN(Nom)-2-GV(Verbe, GN(Det, Nom))-8`
`0-GN(Nom)-2-Verbe-7-GN(Det, Nom)-12` (on doit faire Emp 6)

La première est réduite à nouveau (ligne 8, colonne `Prep` de la table) avec la règle 1, on obtient :

`0-S(GN(Nom), GV(Verbe, GN(Det, Nom)))-1`
`0-GN(Nom)-2-Verbe-7-GN(Det, Nom)-12` (on doit faire Emp 6)

Concernant la première pile, l'action suivante est Emp 6, on fusionne donc les deux processus avec une pile unique :

```
0-S(GN(Nom), GV(Verbe, GN(Det, Nom)))-1-\
                                        -Prep-6
0-GN(Nom)-2-Verbe-7-GN(Det, Nom)-12-----/
```

L'analyse continue et on obtient, au moment de l'analyse de "Paris" la pile de la figure 1.7. **NB :** pour simplifier les figures, on a simplement noté le symbole porté par la racine sur les arcs de la pile et non pas les arbres complets.

Cette méthode introduit donc une factorisation par la droite des différentes piles, et Tomita propose aussi une factorisation par la gauche (partage des sous arbres communs à toutes les piles). Plutôt que de dupliquer la pile avec le processus à chaque point de choix, on la partage en deux voies, conservant ainsi commun le chemin depuis l'état 0 jusqu'à l'état où s'est produit le choix. La pile obtenue est partagée par les processus, elle a une structure de graphe sans cycles qui n'est pas sans rappeler la carte du paragraphe précédent. On peut en fait voir l'analyseur de Tomita comme un analyseur utilisant une carte mais guidé par une table de transition LR qui lui donne son efficacité. La figure 1.8 montre la pile de la figure 1.7 sous forme de graphe sans cycle.

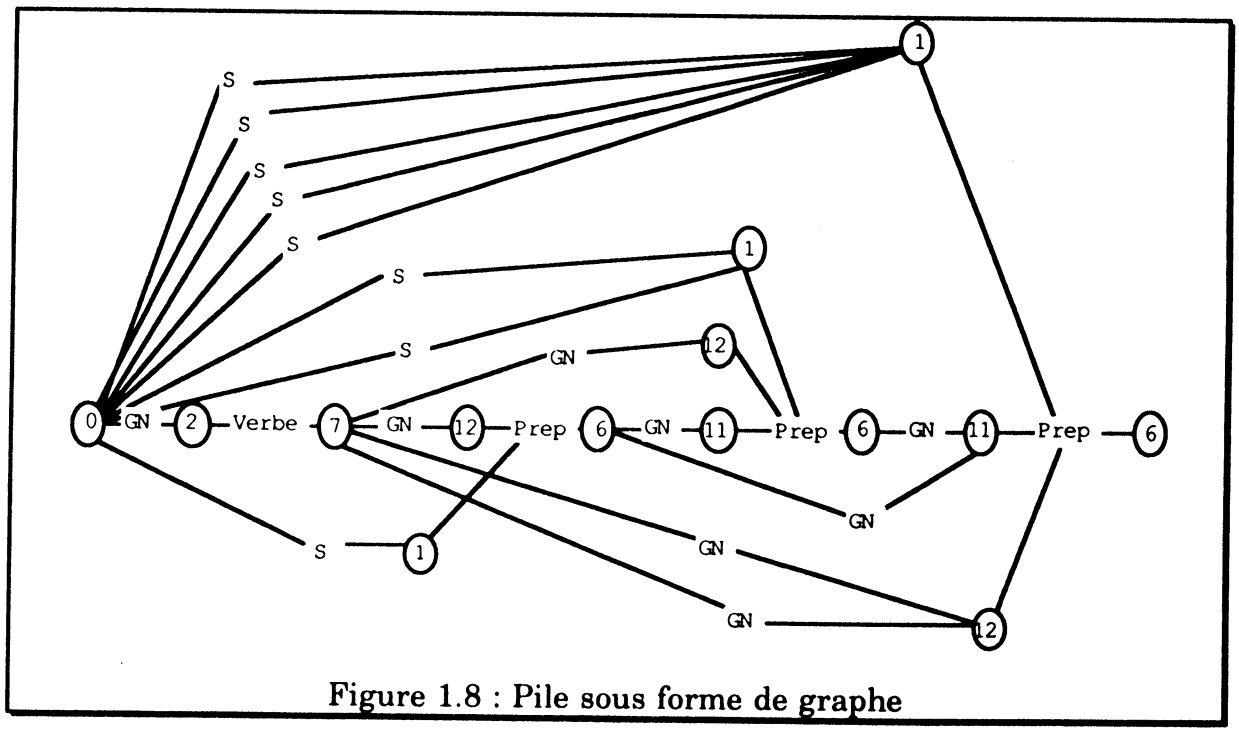


Figure 1.8 : Pile sous forme de graphe

Outre qu'il est très efficace (5 à 10 fois plus rapide que l'algorithme d'Earley [TOMITA 87]), cet algorithme permet également la manipulation d'homographies : il suffit d'appliquer le mécanisme du choix non seulement aux entrées de la table à valeur multiple mais aussi au cas ou plusieurs entrées sont possibles. Cette fonctionnalité autorise le traitement de mots inconnus : on affecte à un mot non reconnu toutes les catégories terminales possibles, le résultat de l'analyse permet de réduire cet ensemble aux seules catégories qui ont permis d'obtenir un résultat. [TOMITA 87] propose

également une méthode de factorisation des arbres syntaxiques produits par mise en commun des sous arbres identiques, on obtient alors un arbre complexe qui est en fait un arbre et/ou. Nous ne détaillerons pas cette technique ici.

Nous avons peu parlé jusqu'ici des phénomènes linguistiques complexes, nous attachant plutôt à la description des mécanismes fondamentaux d'analyse hors contexte. Nous allons voir dans les paragraphes suivants que certains de ces phénomènes ne sont pas (ou difficilement) manipulables par des grammaires purement hors contexte.

1.2. Systèmes transformationnels

1.2.1. Grammaires transformationnelles

Les grammaires transformationnelles sont issues de travaux qui ne visaient pas seulement à décrire la langue naturelle, mais aussi à prendre en compte la capacité humaine à utiliser le langage. Leur but était donc double :

- donner de manière formelle (c'est-à-dire aisément manipulable) une description de toutes les chaînes bien formées d'une langue ;
- modéliser des phénomènes cognitifs tels que la capacité d'une personne à reconnaître et à accepter comme valides des phrases qu'elle n'a jamais entendues, ou la capacité à produire de nouvelles phrases.

Nous donnons ici une description succincte de la théorie connue sous le nom de théorie standard, due à Chomsky ([CHOMSKY 57, 65]). Le but de cette description est l'introduction de la notion de transformation d'arbre et nous verrons au §1.2.2 quelques implications informatiques de cette notion.

Chomsky suppose que pour tout énoncé (toute phrase), il existe une *structure syntaxique profonde* qui porte tout le sens de l'énoncé. Il entend par là que l'interprétation sémantique de l'énoncé est basée sur cette seule structure profonde. La structure de la phrase telle qu'elle apparaît est appelée *structure de surface*. Cette distinction est sensée modéliser les liens sémantiques incontestables entre les 3 phrases suivantes :

"Le bébé mange la bouillie."
"La bouillie est mangée par le bébé."
"Qui mange la bouillie ?"

Ces phrases ont des structures de surface différentes mais auront des structures profondes identiques ou très similaires. Par contre, les deux phrases :

"Jean est habile à convaincre."
"Jean est facile à convaincre."

ont des structures de surface similaires mais auront des structures profondes différentes.

La théorie standard décrit la capacité humaine à utiliser le langage comme un *système transformationnel* (dont la structure est donnée figure 1.9).
Le module de base s'appuie sur une grammaire syntagmatique et est chargé d'engendrer la structure profonde (une grammaire transformationnelle est essentiellement un outil génératif). Cette structure peut se voir attribuer un

sens par le biais d'une interprétation sémantique. La structure profonde subit ensuite un certain nombre de transformations destinées à produire la structure de surface. Cette dernière (arborescente) est linéarisée par un composant phonologique (pour le langage parlé) ou morphologique (pour le langage écrit).

La grammaire du module de base est une grammaire hors-contexte comme celles du §1.1 mais *augmentée* de *traits*, ou *marqueurs*, qui se comportent comme des symboles non terminaux mais qui n'apparaissent que dans la structure profonde. Ces marqueurs sont des indicateurs destinés à guider les transformations.

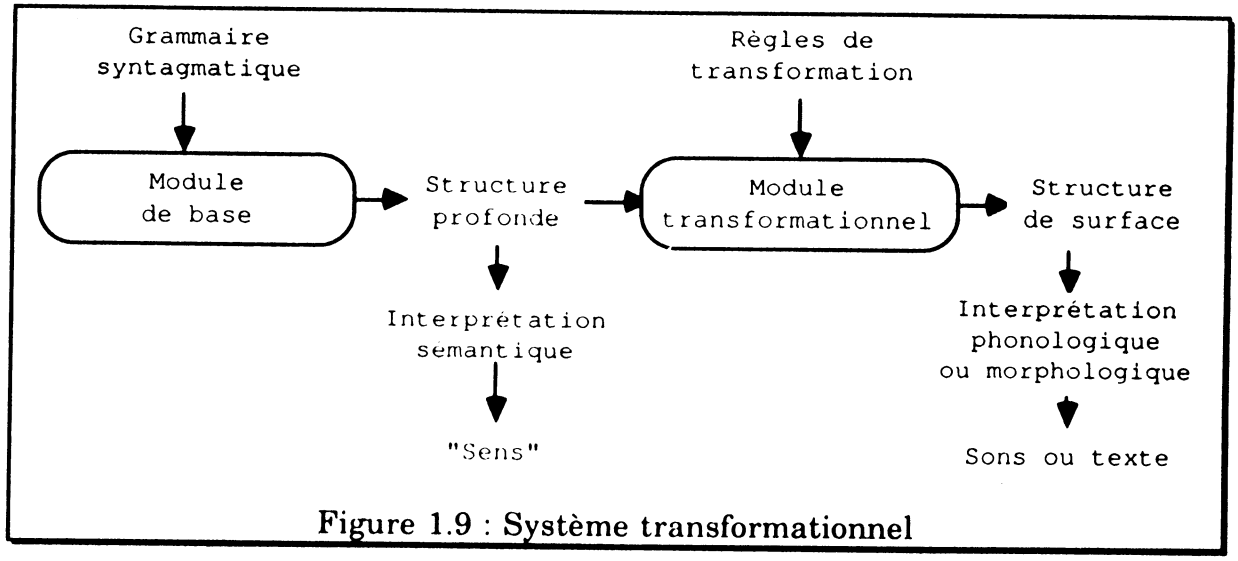


Figure 1.9 : Système transformationnel

Exemple d'une grammaire hors-contexte augmentée :

Le caractère | désigne l'alternative, les éléments entre parenthèses sont facultatifs.

```
S -> Modalité + P
P -> GN + GV
Modalité -> (Actif | Passif) +
            (Assertive | Interrogative | Impérative) +
            (Affirmative | Négative)
GN -> Nombre + GN1
GN1 -> Det + Nom
GV -> (Aux) + Verbe + GN
Nombre -> Singulier | Pluriel
```

La `Modalité` est un marqueur qui permet de sélectionner la transformation à appliquer à la structure profonde qui aura elle toujours l'allure : sujet + verbe + complément (avec cette grammaire très simple). On trouvera figure 1.10 un exemple de structure profonde engendrée par cette grammaire et qui peut correspondre à la phrase "la bouillie n'est pas mangée par le bébé". On remarquera également que la présence du marqueur `Nombre` au niveau du `GN` et non pas au niveau du `Det` et du `Nom` impose un accord global des membres du groupe nominal.

Le module transformationnel applique cycliquement et dans un ordre précis un ensemble de règles de transformation sur la structure profonde. Ces règles

modélisent les phénomènes linguistiques qui sont difficilement manipulables dans un cadre purement hors-contexte (pronominalisation, anaphores, ellipses, accord en genre et en nombre,...). Une règle de transformation est essentiellement une règle de réécriture d'arbre et comprend trois parties :

- une *description structurelle* de la partie de l'arbre qui sera transformée ;
- des *changements structurels* à appliquer à la partie de l'arbre sélectionnée par la description structurelle. Il s'agit d'un ensemble de transformations élémentaires (*suppression* d'un nœud, *substitution* d'un nœud par un autre, *adjonction* d'un nouveau nœud).
- des conditions portant sur les étiquettes des nœuds ; elles permettent de compléter la description structurelle pour la sélection de la partie de l'arbre à transformer ou de préciser quand la règle peut s'appliquer.

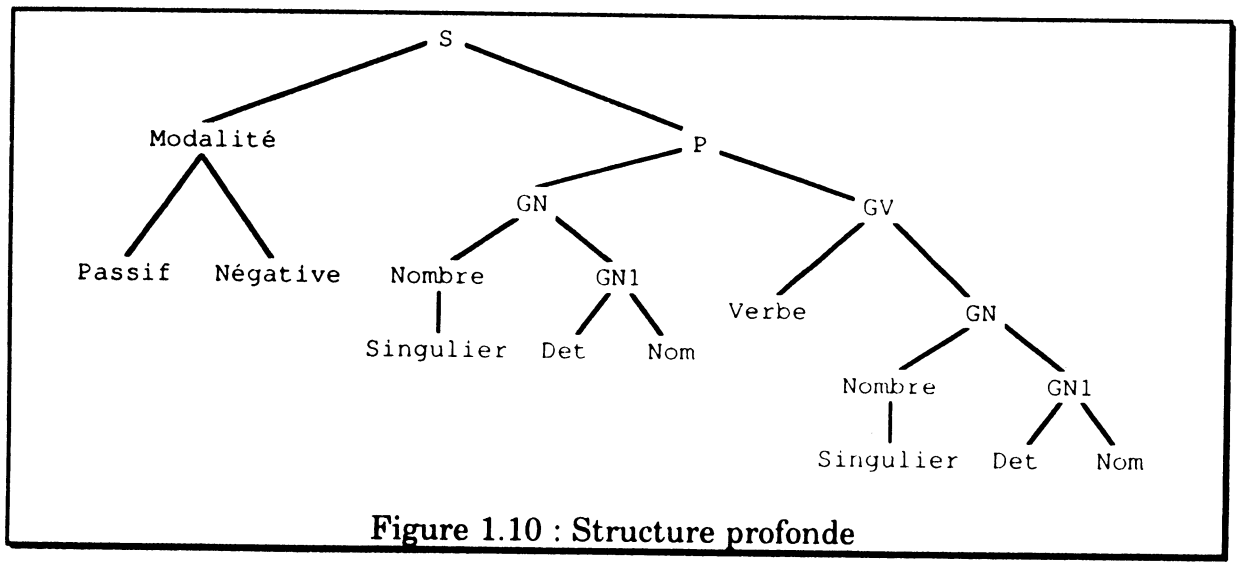


Figure 1.10 : Structure profonde

Exemples de transformations[1] **:**

1) PASSIF (Optionnelle) :

DS	GN	Aux	Verbe	GN
	1	2	3	4
CS	4	2>être+pp	3	par #1

La description structurelle (DS) précise une liste de nœuds (une forêt) numérotés pour pouvoir être référencés dans les changements structurels (CS). Pour que la transformation s'applique, la liste des nœuds de la DS doit apparaître de manière contiguë dans l'arbre mais pas nécessairement au même niveau.

Les CS de cet exemple précisent :

- que le sous arbre dont la racine est 1 est remplacé par le sous arbre dont la racine est 4 (substitution) ;
- qu'au nœud 2 on adjoint deux nouveaux nœuds à droite (>) étiquetés par `être` et `pp` (indicateurs destinés à guider la formation de la forme passive du verbe dans une transformation ultérieure) ;

[1]Empruntés à [SABAH 88]

- que le verbe 3 ne change pas ;
- que le nœud 4 est remplacé (#) par un nœud portant la même catégorie (GN) et ayant deux fils : un noeud étiqueté par par à gauche et le sous arbre dont la racine est 1 à droite.

Cette règle n'a pas de conditions, elle permet la transformation décrite par la figure 1.11.

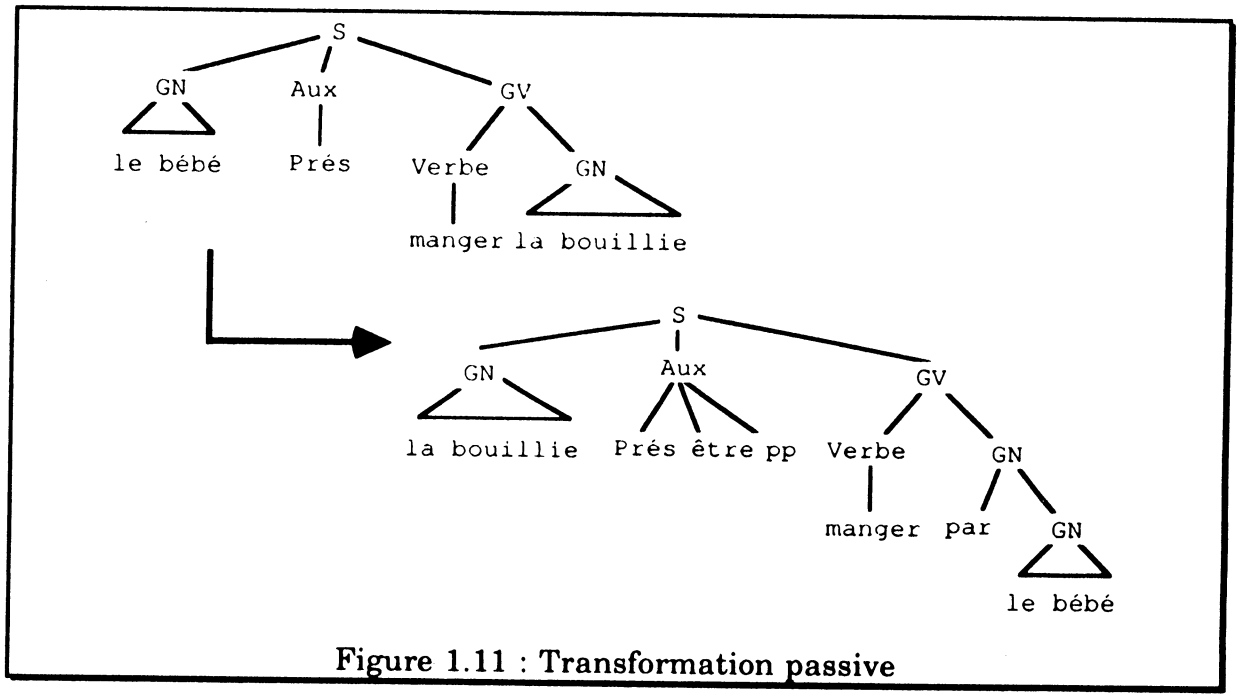


Figure 1.11 : Transformation passive

2) Suppression de GN égaux (Obligatoire) :

DS	GN	Verbe	que	GN	Aux	Verbe
	1	2	3	4	5	6
CS	1	2	0	0	5+Infinitif	6

Conditions : 1) 1 = 4
2) Type(2) = volonté, croyance

Cette transformation produit une structure de surface pour "le bébé veut manger la bouillie" à partir d'une structure profonde dont la traduction littérale serait "le bébé veut que le bébé mange la bouillie". Pour l'interprétation sémantique (basée sur la structure profonde), il est en effet indispensable de savoir que le sujet de "mange" est le même que celui de "veut", alors que c'est implicite dans la structure de surface. Les conditions imposent que les sujets soient effectivement les mêmes (1 = 4) et que le verbe soit d'un type particulier. Cela interdit l'application de cette transformation à des phrases comme "le bébé veut que sa mère goûte la bouillie" ou "le bébé voit que le bébé mange la bouillie". Les deux nœuds 3 et 4 sont supprimés (0) et le verbe est mis à l'infinitif (ajout d'un marqueur sur l'auxiliaire). On notera également que cette transformation, quand elle est applicable, est obligatoire car la structure de surface correspondante ne serait pas correcte (ce n'était pas le cas avec la première transformation).

On voit que l'un des intérêts de la notion de transformation est de faciliter l'écriture des règles du module de base puisqu'on n'y décrit que les éléments très réguliers de la langue (on rend par exemple explicites les éléments ellidés dans la structure de surface), mais cela repose sur l'hypothèse que le sens est entièrement contenu dans la structure profonde. Cette hypothèse a été très critiquée et cela a conduit à des évolutions de la théorie.

Une première évolution a introduit un module d'*insertion lexicale* entre les deux modules du système transformationnel. Ce module permet la manipulation de phénomènes morphologiques complexes (irrégularités, exceptions qui ne rentrent pas dans le cadre des transformations) qui dépendent de la structure profonde et qui ne peuvent donc pas être traités par le module morphologique qui travaille sur la structure de surface. Ainsi la nominalisation peut être traitée par une transformation, comparer : "l'entreprise reçoit une facture" et "la réception de la facture par l'entreprise". Mais cette transformation pose le problème des noms à valeur prédicative n'ayant pas d'équivalents verbaux, comme "oblation"[1] dans : "l'oblation d'un psaume par les fidèles".

L'évolution suivante a conduit à un glissement de l'interprétation sémantique de la structure profonde vers la structure de surface. Pour conserver les qualités sémantique de la structure profonde, la structure de surface a été augmentée de *traces* : marqueurs spéciaux permettant de rendre explicites les déplacements de constituants [CHOMSKY 75]. On trouvera figure 1.12 la structure de surface de la phrase "le bébé veut manger la bouillie".

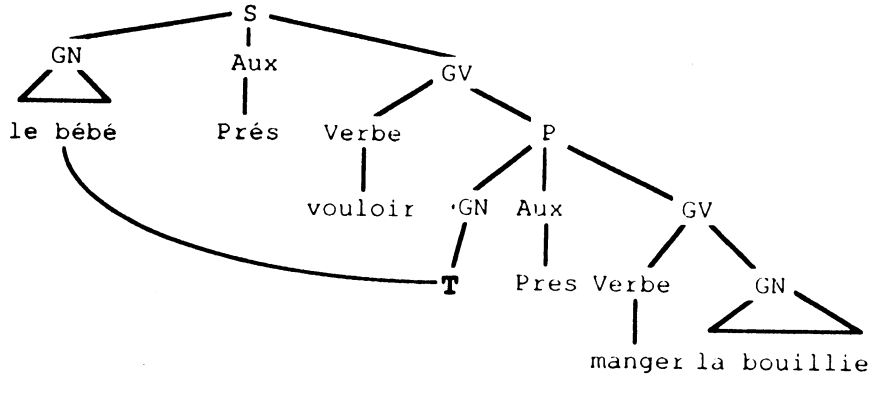


Figure 1.12 : Structure de surface avec trace

Ces théories ont été peu utilisées telles quelles dans la pratique, d'une part à cause des difficultés de mise en œuvre et d'autre part à cause des difficultés à articuler la syntaxe et la sémantique : la réalisation de l'interprétation sémantique sur la base de structures profondes n'est en effet pas immédiate.

Les travaux de Harris, définissant les grammaires en chaînes [HARRIS 68], utilisent la notion de transformations mais sans le passage par la structure profonde ; elles ont eu des applications pratiques plus importantes ([SALKOFF 73], [SAGER 73], [GRISHMAN 73], [JAYEZ 79, 82]).

[1] action d'offrir quelque chose à Dieu.

Mais la théorie standard et ses évolutions constituent les fondements de la linguistique informatique. La dernière évolution (connue sous le nom de théorie du gouvernement et liage [CHOMSKY 82]) intègre plus nettement les aspects sémantiques avec les aspects syntaxiques. Nous donnons au §1.1.3 une description de certaines théories récentes qui intègrent les concepts des théories chomskyiennes sous une forme plus unifiée et qui traitent également la sémantique comme une partie intégrante du processus d'analyse de la langue.

1.2.2. Transformations d'arbres : Robra

L'introduction de la notion de transformation entraîne du point de vue pratique la mise en œuvre de mécanismes capable de transformer une arborescence en une autre. En effet dans les grammaires syntagmatiques (et tout particulièrement dans les grammaires hors-contexte), le processus d'analyse consiste en la mise en correspondance d'une chaîne de symboles terminaux et d'un arbre étiqueté. La nouveauté des grammaires transformationnelles est la mise en correspondance de deux arbres : un arbre objet auquel on applique des transformations et un arbre cible qui est le résultat de l'application de une ou de plusieurs transformations à l'arbre objet. Déjà introduite dans les systèmes-Q [COLMERAUER 70], ce type de transduction arbre-arbre a été systématisé dans Robra.

Robra [CHAUCHE 74] [BOITET 76, 78] est un LSPL (Langage Spécialisé pour la Programmation Linguistique) utilisé au GETA[1] dans le processus de traduction pour les trois phases d'analyse structurale, de transfert structural et de génération structurale.
Robra n'est pas à proprement parler un système transformationnel au sens donné ci-dessus, mais nous avons choisi de le présenter pour deux raisons :

- l'opération de base de Robra est justement la réécriture d'un arbre en un autre, il constitue donc une bonne illustration des mécanismes informatiques induits par la notion de transformation : sélection dans un arbre, déplacement de sous-arbres, ...
- cette opération de base est unique : tout le processus d'analyse est décrit par des réécritures, y compris la construction de la structure de constituants. C'est une méthode qui tranche avec les méthodes du 1.1 qui sont plus proches de méthodes algorithmiques classiques. Cette cassure permet d'introduire de nouveaux concepts qui seront utiles au chapitre 2.

1.2.2.1. Transduction d'arbres

Robra est un *transducteur* d'arbres. Il applique un ensemble de transformations (transductions) à une arborescence étiquetée pour produire une arborescence étiquetée. L'ensemble des transformations et le contrôle de leur application est défini par un *système transformationnel*. Un système transformationnel St est un quadruplet (E, P, G, C) où :

E est un ensemble dénombrable d'*étiquettes* utilisé pour l'étiquetage des *arbres objets* (AO) auxquels s'applique St. On note T_E l'ensemble des arbres finis étiquetés sur E. Les étiquettes dans Robra sont des

[1]Groupe d'Etude pour la Traduction Automatique.

ensembles de *variables* : couples (nom, valeur) ou la valeur peut être un entier, un symbole, un ensemble de symboles, ...

P est un ensemble fini de *règles de production* (voir ci-dessous).

G est un ensemble de *grammaires transformationnelles*, chaque grammaire est un sous-ensemble de P qui doit être muni d'une relation d'ordre.

C est le *graphe de contrôle*. Chaque nœud du graphe porte un élément de G ∪ { &nul } (grammaire vide). Les arcs peuvent porter des conditions de transition.

L'opération de base de Robra est l'application d'une règle de production R à un arbre objet (AO) pour produire un arbre cible. Le format général d'une règle est le suivant :

```
<nom> <schéma de sous arbre> / <conditions d'occurrence>
        == <arbre image> / <fonction de transfert> / <actions>
```

Les *schémas de sous arbre* et *l'arbre image* sont des arbres étiquetés par des *références*. Une référence peut être :

- un entier pour désigner un nœud,
- un symbole précédé d'un $ pour désigner une liste de nœuds frères adjacents (une forêt),
- une * pour désigner la forêt vide.

Les *conditions d'occurrence* sont des expressions booléennes portant sur les valeurs des variables des étiquettes des nœuds du schéma. Elles permettent de restreindre la portée de la règle.
La *fonction de transfert* permet de recoller les sous arbres (ou les sous forêts) de l'arbre objet laissés pendants par le mécanisme de substitution.
Les *actions* sont essentiellement des affectations de valeurs aux variables des étiquettes des nœuds.

L'application d'une règle comporte deux étapes distinctes :

- la *sélection* de la partie de l'AO à modifier, ceci suppose un mécanisme de *filtrage* (*pattern matching*) de l'AO par le schéma de sous arbre donné par la règle ; ceci suppose également un langage de sélection suffisamment précis (constitué ici du schéma d'occurrence et des conditions d'occurrence) ;
- la *substitution* des occurrences du schéma dans l'AO par l'arbre image donné par la règle.

Trois particularités caractérisent l'application d'une règle dans Robra par rapport à un système de réécriture plus classique (comme Prolog) :

1 - le sous arbre substitué n'est pas nécessairement connexe en largeur, ni complet en profondeur, ni complet en largeur. C'est-à-dire qu'on peut toujours avoir, sauf indication contraire, une forêt (dite forêt implicite) entre deux nœuds du schéma, sous une feuille du schéma, à gauche et à droite d'une liste de nœuds du schéma. On trouvera figure 1.13 un exemple de schéma avec les forêts implicites.

2 - la substitution peut s'appliquer en **parallèle** sur plusieurs occurrences d'un schéma dans l'arbre objet.

3 - on peut faire usage du **contexte** du sous arbre substitué.

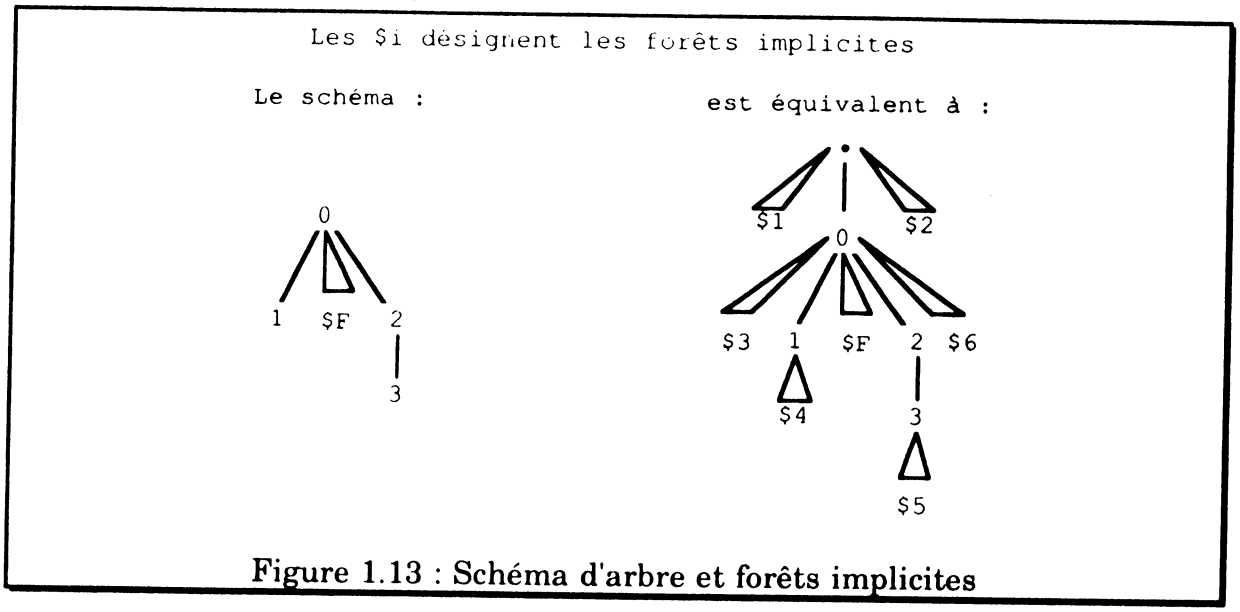


Figure 1.13 : Schéma d'arbre et forêts implicites

Exemple de règle de production :

On donne ci dessous une possibilité de traduction en Robra de la règle de transformation 2) du §1.2.1. Cette règle peut s'appliquer à l'arbre :

```
S(
  GN("le", "bébé"),
  Aux,
  GV(
     "vouloir",
     P(
       "que",
       S(GN("le bébé"), Aux, GV("manger",GN("la bouillie")))
      )
    )
 )
```

pour donner l'arbre :

```
S(
  GN("le", "bébé"),
  Aux,
  GV(
     "vouloir",
     S(Aux(TEMPS=INFINITIF), GV("manger",GN("la bouillie")))
    )
 )
```

```
SUPP-GN : 0(1, 2(3, 4(5, 6(7, 8))))
/ 0 : $S, 1 : $GN, 2 : $GV, 3 : $Verbe, $Modal, 4 : $P
5 : UL -EG- "que", 6 : $S, 7 : 1, 8 : $Aux
== 0(1, 2(6(8)))
/ * <- 4, 5, 7/
8 : 8, TEMPS := INFINITIF.
```

$XXX dénote une procédure, déclarée comme un abrégé de `VAR -EG- XXX`. Ainsi `$Modal` permet de vérifier que le verbe de la principale est bien de type `Modal` (volonté, croyance, ...). La fonction de transfert précise ici que les nœuds 4, 5 et 7 sont éliminés, la seule action affecte l'infinitif à l'auxiliaire du verbe de la relative.

1.2.2.2. Le contrôle dans Robra

Le principal problème posé par la mise en œuvre d'un système basé sur des règles de production est celui du contrôle : si plusieurs règles s'appliquent, laquelle choisir ? Si une règle s'applique plusieurs fois sur le même arbre, faut-il tout appliquer ou choisir une occurrence et n'appliquer qu'une fois ? Dans ce cas comment choisir ? Une méthode possible pour régler ces problèmes est de tout appliquer systématiquement en construisant tous les résultats possibles (en parallèle ou avec retour arrière). Le retour arrière est utilisé par exemple par la résolution de Prolog, avec deux inconvénients :

- efficacité réduite et obligation d'ajouter un mécanisme spécial pour limiter les retours arrières (*cut*) ;
- difficulté pour écrire les règles et maintenir la cohérence de la base de règles car on ne maîtrise pas suffisamment le mécanisme de sélection d'une règle.

Robra dispose d'un mécanisme de contrôle très complet et structuré sur deux niveaux :

- au niveau du **graphe de contrôle** d'un système transformationnel : Robra contient un automate à pile qui parcours le graphe de contrôle en appliquant la grammaire contenue dans chaque nœud à l'arbre objet. L'arbre cible obtenu devient le nouvel arbre objet. En cas d'échec (dans l'application d'une grammaire, ou si l'arbre cible obtenu ne vérifie aucune des conditions de transition porté par les arcs sortants du nœud), l'automate effectue des retours arrières. Le parcours réussit si un nœud de sortie (portant la grammaire vide &nul) est atteint, dans le cas contraire, le résultat est égal à l'arbre objet initial.
  Une règle de la grammaire d'un nœud peut contenir un appel récursif au système transformationnel avec comme arbre objet l'arbre courant et comme nœud initial le nœud contenant la grammaire.
  Notons que les grammaires ont des *modes d'application* qui peuvent modifier le contrôle au niveau du graphe. On peut ainsi interdire le retour arrière sur un nœud (mécanisme semblable au cut de Prolog), ou contraindre l'application d'au moins une règle de la grammaire sous peine d'échec ; on force ainsi le retour arrière et on limite les risques de bouclage.
- au niveau d'une **grammaire** : on peut appliquer une fois la grammaire (mode *unitaire*) ou autant de fois que possible (mode *exhaustif*). Une application élémentaire comprend trois étapes : (1) recherche des schémas d'occurrence des règles dans l'arbre objet, (2) *sélection* du sous ensemble applicable et (3) application en parallèle et *marquage*. La sélection permet de discriminer les règles dont les occurrences partagent des points actifs (en gros celles dont les arbres images ne sont pas disjoints). Cette sélection est fonction du mode global d'application de la grammaire : *coupe* ou *total* (en mode total une occurrence peut dominer strictement une autre, ce qui est interdit en mode coupe). En mode coupe on peut également choisir les occurrences en partant des feuilles de l'arbre objet (mode *bas*) ou de la racine (mode *haut*). Le *marquage* consiste à noter sur les racines

des transformations les règles qui ont été appliquées. On évite ainsi d'appliquer plusieurs fois la même règle à la même racine.
Une règle peut appeler récursivement une sous grammaire ou le système transformationnel complet. Dans chaque cas, les marques des racines de transformation sont modifiées : on efface les marques des règles intervenant dans la récurrence ; au retour on effectue l'union des nouvelles marques et des anciennes.

Nous n'avons pas donné ici tous les détails du contrôle pas plus que du mécanisme de filtrage, mais il est clair que Robra a été conçu pour fournir au linguiste chargé d'écrire des grammaires une grande liberté. On peut regretter que cette liberté se paye par une certaine complexité (notamment dans les mécanismes de contrôle).

Les arguments jouant en faveur d'un mécanisme de transduction d'arbres comme celui de Robra, par opposition aux mécanismes d'analyse chaîne-arbre du §1.1 sont la souplesse et la modularité.
La souplesse car un transducteur peut fournir un résultat (un arbre même incomplet) sur une entrée incorrecte, alors qu'un analyseur pur ne fournit un résultat que si l'analyse réussit. De plus le format de l'arbre issu d'un analyseur est fixé par la grammaire alors que toute liberté est donné au transducteur par le biais des règles de production.
La modularité dérive de l'homogénéité des structures d'entrée et de sortie. Elle permet la composition des grammaires et facilite donc le travail du linguiste qui peut décomposer la description d'une langue en sous ensembles réguliers. Elle favorise également la mise en œuvre de mécanismes de contrôle puissants (graphe).
Mais il est clair que ces qualités se payent en temps humain pour la mise en œuvre et en temps machine à l'exécution.

1.3. Grammaires à structures de traits

Une des principales critiques des grammaires transformationnelles est la difficulté à interpréter sémantiquement la structure profonde construite. D'autre part, les tentatives pour donner une couverture totale de la langue avec ces grammaires ont considérablement compliqué les transformations et les contraintes d'application (dans quel ordre, sous quelles conditions, ...). Certains linguistes ont tenté de résoudre ces deux problèmes en étendant la notion de traits (marqueurs). Chaque nœud de la structure syntaxique est étiqueté par un ensemble complexe de traits grâce auquel :

- la distinction structure profonde / structure de surface disparaît au profit d'un couple de structures construites en parallèle : une structure hors-contexte et une *structure de traits*. Cette manipulation d'informations complexes simultanément à la construction d'une structure hors-contexte a déjà été utilisé par [WOODS 70] dans les grammaires des réseaux de transitions augmentés (ATN) sous forme d'actions associées aux transitions du réseau et agissant sur des registres. Dans les théories décrites dans ce paragraphe, cette manipulation fait partie intégrante du formalisme de description des structures alors que dans les ATN elle fait partie du mécanisme d'analyse.

- les transformations disparaissent et les phénomènes qu'elles permettaient de modéliser sont maintenant traités par la notion de *catégorie dérivée* ;
- l'hypothèse d'autonomie de la syntaxe par rapport à la sémantique, caractéristique des théories chomskyennes, est remise en cause au profit de la construction d'une structure sémantique soit en parallèle avec la structure syntaxique de surface, soit en s'appuyant directement sur cette structure.

La notion de structures de traits est partiellement reprise des travaux liés aux grammaires systémiques [HALLIDAY 61] [WINOGRAD 72, 83] : les traits sont organisés de manière hiérarchique, autorisant un choix plus ou moins fin de la valeur d'un trait en fonction des données présentes. On peut d'autre part avoir plusieurs *systèmes de choix*, introduisant la notion de représentation multi-niveaux. On trouvera figure 1.14 un exemple des choix associés à quatre systèmes de choix pour la phrase "Cet article est écrit par Jean".

	Cet article	est écrit	par Jean
Fonction syntaxique	Sujet	Prédicat	Comp. d'Agent
Causalité	But	Action	Acteur
Thème	Thème	Rhème	Thème
Information	Ancienne		Nouvelle

Figure 1.14 : Systèmes de choix

Le concept de *trait sémantique* (propriété sémantique d'un mot ou fonction sémantique liant plusieurs mots) est repris des travaux sur les grammaires de cas [FILLMORE 68, 71] [SPARCK-JONES 87] dans lesquelles chaque mot ou chaque groupe de mots se voit attribuer un cas (un trait) lié à la fonction sémantique (agent, objet, instrument, ...).

L'attrait des formalismes présentés ci-après réside dans la possibilité qu'ils offrent d'intégrer de manière uniforme les concepts fondamentaux des théories citées ci-dessus. De plus, l'uniformité de représentation de la connaissance (structures de traits) facilite leur mise en œuvre : les opérations de manipulation sont peu nombreuses et les différents niveaux de représentation de la connaissance (morphologique, syntaxique, sémantique) sont bien intégrés, ce qui évite d'avoir à définir de manière précise les frontières entre les différents niveaux.

1.3.1. Grammaires syntagmatiques généralisées

Plus connues sous le nom de GPSG (Generalized Phrase Structure Grammars) [GAZDAR 85, 86], elles sont des descendantes directes des grammaires transformationnelles. Comme ces dernières, elles sont une

extension des grammaires syntagmatiques mais elles n'utilisent pas la notion de transformation.

Les GPSG apportent quatre nouveautés essentielles :

- une théorie des *traits syntaxiques* et la notion de *catégorie dérivée* ;
- la notion de *méta-règles* qui engendrent des règles à partir d'autres règles et qui jouent du point de vue linguistique un peu le rôle des transformations ;
- un mécanisme de transmission des valeurs des traits de constituant à constituant par le biais de *principes d'instanciation* ;
- l'introduction de règles sémantiques associées aux règles syntaxiques.

Traits syntaxiques

Dans une GPSG, les symboles associés classiquement aux catégories syntaxiques (S, GN, GV, ...) sont remplacés par des ensembles non ordonnés de traits (paires <attribut, valeur>). Chaque nœud de l'arbre syntaxique est donc étiqueté par un ensemble de traits représentant les propriétés syntaxiques (genre, nombre, affirmation ou négation,...) mais aussi la catégorie.

Exemple de catégorie :

```
[<N +>, <V ->, <BAR 2>, <PER 3>, <PLU ->]
<N +>, <V -> désigne la catégorie (ici un GN)
<BAR 2> précise le niveau de la catégorie[1]
<PER 3> troisième personne
<PLU -> du singulier                    (<PLU +> aurait dénoté un pluriel)
```

Catégorie dérivée

Un trait particulier, le trait SLASH (/) permet de noter l'absence dans un syntagme du constituant spécifié par le SLASH. La catégorie obtenue est une catégorie dérivée de la catégorie du syntagme complet.

Exemple :

Une relative peut être vue comme une phrase dans laquelle il manque un groupe nominal. Pour décrire une relative, on aura une règle du type :

```
[<N +>, <V ->, <BAR 1>]
    -> [<N +>, <V ->, <BAR 0>] +
       [<S +>, <Q +>, <SLASH [<N +>, <V ->, <BAR 2>] ]
```

ou sous une forme abrégée plus lisible :

```
N1 -> N0 + S / N2 <Q, +>
```

On peut noter que la valeur du trait SLASH est elle-même un ensemble de traits.

---

[1] le BAR fait référence à la théorie X-barre de Chomsky selon laquelle tous les principaux groupes syntagmatiques (verbaux, nominaux, adjectivaux, prépositionnels) ont la même structure profonde qui peut être décrite par les règles :

$\bar{\bar{X}}$ -> Spécificateurs + $\bar{X}$

$\bar{X}$ -> X + Compléments

X -> Tête, [<X, +>]

ou X peut être N, A, V, ou P.

Cette méthode permet de prendre en compte les phénomènes de déplacement de constituants et donc de résoudre un des principaux problèmes qui avaient motivé l'introduction de la notion de transformation. Elle s'apparente à la méthode des traces dans la mesure ou un trou apparaît dans la structure, mais ici aucun lien explicite n'est établi entre le trou et le constituant, la correspondance sera établie par les principes d'instanciation et les règles sémantiques.

Méta-règles

Pour introduire des catégories dérivées dans une grammaire de façon systématique (et sans être obligé de multiplier le nombre de règles), Gazdar introduit des méta-règles qui produisent des règles par transformation d'autres règles. Cette transformation est facilitée par le fait que les règles sont partagées en deux types : des règles de dominance immédiate et des règles de précédence linéaire. Les premières déterminent la composition d'un syntagme mais sans préciser l'ordre des constituants qui est défini par les secondes. Ceci n'apporte rien à la puissance théorique du formalisme mais facilite son utilisation pratique (écriture des grammaires).

Principes d'instanciation

Le module de base des GPSG est un module fondamentalement hors contexte produisant des structures syntagmatiques. La dernière étape de l'analyse est donc le passage des structures produites dans un filtre qui élimine les structures incorrectes grâce aux principes d'instanciation des traits et qui construit la représentation sémantique grâce aux règles sémantiques.
Citons ici sans entrer dans les détails :

- le principe du contrôle de l'accord qui précise comment les traits AGR (pour agreement) et SLASH sont transmis d'un nœud à un autre pour vérifier les accords en genre et en nombre ou pour déterminer le genre et le nombre d'un syntagme absent.
- le principe du trait de tête : certains traits sont dits *traits de tête* et se transmettent automatiquement du constituant père (partie gauche de la règle) au constituant fils (un des constituant de la partie droite désigné par le symbole spécial `H` dans les règles). Ainsi dans la règle : `[<V +> <BAR 2>] -> H, [<N +>, <BAR 2>]`, le symbole `H` désigne un nœud dont la valeur n'est pas précisée mais qui sera nécessairement un verbe car le trait `V` est un trait de tête. Ce principe permet de distinguer dans un syntagme un constituant particulier, dit tête de syntagme : on peut rapprocher cette notion de la notion de gouverneur (voir §1.1.4).
- le principe du trait de pied, qui concerne essentiellement le trait SLASH, permet d'établir les liens entre le trou laissé par le trait SLASH dans un syntagme et le référent associé (pronom par exemple).

Règles sémantiques

A chaque règle syntaxique est associée une règle sémantique (sous forme d'une formule du lambda calcul). En fin de dérivation, ces formules sont combinées pour chaque arbre construit et les arbres ne donnant pas une formule bien formée sont éliminés. Ces règles construisent donc une représentation

sémantique de la phrase mais permettent aussi de contrôler la grammaticalité des structures produites par le composant de base. Cela donne plus de liberté dans l'écriture de ce composant qui peut engendrer plus que les seules phrases de la langue. La structure sémantique joue le rôle dévolu à la structure profonde dans les grammaires transformationnelles : deux phrases sémantiquement proches auront des structures syntaxiques différentes mais une même formule sémantique ; au contraire, deux phrases de structures syntaxiques identiques pourront avoir des formules sémantiques différentes.

Notons pour conclure que l'introduction de structures de traits complexes entraîne un glissement de la compétence linguistique de la grammaire vers le lexique. Cet aspect, déjà présent dans les GPSG, est à la base du formalisme décrit au paragraphe suivant.

1.3.2. Grammaires lexicales fonctionnelles

Très proches des grammaires fonctionnelles [KAY 81, 84], les LFG (pour Lexical Functional Grammars), développées par Kaplan et Bresnan [KAPLAN 72, 73] et [BRESNAN 81b] unifient la représentation des entrées lexicales et des structures de phrases.

La description d'une phrase est composée de deux structures : une structure de constituants classique (appelée C-structure) et une structure fonctionnelle (F-structure). Une grammaire lexicale fonctionnelle comporte deux types de règles :

- des règles hors contextes qui permettent d'engendrer la C-structure. La grammaire contexte associée a une couverture très large : elle peut engendrer des structures incorrectes, mais chaque règle hors contexte porte des équations fonctionnelles qui permettent de filtrer ces structures.
- des règles lexicales qui factorisent les redondances des entrées lexicales. Elles complètent une entrée lexicale avec des valeurs de traits déduites des valeurs déjà présentes. Ainsi si on a le trait <Humain +>, on peut déduire les traits <Animé +> et <Abstrait ->. Mais certaines de ces règles permettent aussi de modifier la structure d'une entrée lexicale. Ainsi dans les LFG, la passivation est obtenue par modification de l'entrée lexicale du verbe. Cette méthode facilite le traitement des exceptions (verbes qui n'ont pas de passif par exemple).

Exemple simple de LFG :

Règles :

Les équations fonctionnelles portent sur les traits associés aux nœuds, une flèche vers le haut permet de désigner le père tandis qu'une flèche vers le bas désigne le fils. Tout se passe comme si une équation était portée par l'arc qui relie deux catégories dans l'arbre syntaxique. Dans la première règle ci-dessous, le sujet de S est le GN et les traits du GV remontent sur le S.

```
S ->      GN      +      GV
     (↑ Sujet) = ↓    ↑ = ↓
```

```
GN -> Det + Nom
GV -> Verbe      +    ( GN )
                      (↑ Objet) = ↓
```

Entrées lexicales :

```
le          Det      (↑ Définition) = Défini
                     (↑ Nombre) = Singulier
                     (↑ Genre) = Masculin
la          Det      (↑ Définition) = Défini
                     (↑ Nombre) = Singulier
                     (↑ Genre) = Féminin
bébé        Nom      (↑ Prédicat) = 'bébé'
                     (↑ Nombre) = Singulier
                     (↑ Genre) = Masculin
bouillie    Nom      (↑ Prédicat) = 'bouillie'
                     (↑ Nombre) = Singulier
                     (↑ Genre) = Féminin
mange       Verbe    (↑ Mode) = Indicatif
                     (↑ Temps) = Présent
                     (↑ Personne) = 3
                     (↑ Nombre) = Singulier
                     (↑ Prédicat) = 'Manger <(↑ Sujet) (↑ Objet)>
```

Ces exemples très simples montrent l'uniformité de notation des différents niveaux de connaissance du lexique : morpho-syntaxique (genre, nombre, temps,...) et sémantique (prédicat). On peut de plus décrire les compléments autorisés pour la structure prédicative d'un verbe, les règles sont chargées d'associer à chaque élément de cette structure une réalisation concrète (via les composants de la C-structure).

Une phrase est engendrée en trois étapes :

1 - construction de la C-structure par génération avec la partie hors contexte des règles en ignorant les équations fonctionnelles, qui sont simplement attachées aux nœuds créés. On peut donc utiliser pour cette étape les techniques efficaces décrites au §1.1.

2 - quand toutes les feuilles de l'arbre portent une catégorie lexicale, un mot de cette catégorie est choisi dans le dictionnaire, les équations fonctionnelles associées sont attachées au nœud.

3 - les équations portées par chaque nœud sont *instanciées* : on associe à chaque nœud un nom de variable unique et on remplace les flèches montantes par le nom de la variable associée au père du nœud qui porte la flèche et les flèches descendantes par la variable associée au nœud qui porte la flèche. L'ensemble des équations ainsi obtenu est résolu pour donner la F-structure de la phrase. Si aucune solution n'est possible la structure hors contexte est éliminée, si plusieurs solutions sont possibles c'est que la phrase est ambiguë.

Exemple :

Avec la phrase "le bébé mange la bouillie"on obtient par génération la C-structure déjà vue plus haut (figure 1.2). On associe au GN "le bébé" la variable x_1, au S la variable x_2, au GV la variable x_3, au GN "la bouillie" la variable x_4 (les feuilles ne sont pas référencées dans les équations).

Après instanciation, on obtient les équations :

```
le          (x1 Définition) = Défini
            (x1 Nombre) = Singulier
            (x1 Genre) = Masculin
bébé        (x1 Prédicat) = 'bébé'
            (x1 Nombre) = Singulier
            (x1 Genre) = Masculin
mange       (x3 Mode) = Indicatif
            (x3 Temps) = Présent
            (x3 Personne) = 3
            (x3 Nombre) = Singulier
            (x3 Prédicat) = 'Manger <(↑ Sujet) (↑ Objet)>
la          (x4 Définition) = Défini
            (x4 Nombre) = Singulier
            (x4 Genre) = Féminin
bouillie    (x4 Prédicat) = 'bouillie'
            (x4 Nombre) = Singulier
            (x4 Genre) = Féminin
```

La règle `S -> GN + GV` donne :

$(x_2$ Sujet$) = x_1$

$x_2 = x_3$

et la règle `GV -> Verbe + ( GN )` :

$(x_3$ Objet$) = x_4$

La F-structure résultant de la résolution de ces équations est la suivante :

$$x2 = x3 = \begin{bmatrix} \text{Sujet} = x1 \\ \text{Objet} = x4 \\ \text{Prédicat} = \text{'Mange<bébé,bouillie>'} \\ \text{Mode} = \text{Indicatif} \\ \text{Temps} = \text{Présent} \\ \text{Personne} = 3 \\ \text{Nombre} = \text{Singulier} \end{bmatrix}$$

$$x1 = \begin{bmatrix} \text{Définition} = \text{Défini} \\ \text{Genre} = \text{Masculin} \\ \text{Nombre} = \text{Singulier} \\ \text{Prédicat} = \text{'bébé'} \end{bmatrix}$$

$$x4 = \begin{bmatrix} \text{Définition} = \text{Défini} \\ \text{Genre} = \text{Féminin} \\ \text{Nombre} = \text{Singulier} \\ \text{Prédicat} = \text{'bouillie'} \end{bmatrix}$$

On voit sur cet exemple comment l'accord en genre et en nombre entre le déterminant et le nom est simplement vérifié au moment de la résolution des équations : le déterminant "transmet" son genre et son nombre au GN, le nom fait de même et donc les valeurs doivent être identiques.

L'absence d'un constituant, que nous avons illustrée jusqu'ici avec la phrase "le bébé veut manger la bouillie", est traitée dans les LFG grâce aux équations. On aura dans l'entrée lexicale de "veut" :

(↑ Prédicat) = 'Veut < (↑ Sujet) (↑ Vcomp)>'

(↑ Sujet) = (↑ Vcomp Sujet)

et la règle :

GV -> Verbe + GV1

(↑ Vcomp) = ↓

On précise ainsi que le sujet de la complétive (↑ Vcomp Sujet) est le même que le sujet de la principale (↑ Sujet).

Les LFG permettent également de représenter le déplacement de constituants comme dans la phrase "Combien de cuillerées le bébé mange-t-il le soir ?". Le mécanisme utilisé est assez semblable au trait SLASH des GPSG : le formalisme permet l'introduction d'un constituant se réécrivant vide (∅) auquel est associé un trait dit de *dépendance lointaine* étiqueté par le constituant ($\Uparrow_{GN}$). Ce trait va de pair avec un unique trait $\Downarrow_{GN}$, lié au constituant déplacé. Pour traiter l'exemple ci-dessus on aura les règles :

GN -> ∅

↑ = $\Uparrow_{GN}$

S -> GN + GN + GV1

↓ = $\Downarrow_{GN}$ (↑ Sujet) = ↓ ↑ = ↓

La résolution des équations effectuera le lien entre les deux éléments de la paire.

L'utilisation d'un module purement hors contexte permet l'emploi des techniques d'analyse du §1.1. L'utilisation d'une carte, grâce aux facilités qu'il donne sur l'ordre d'application des règles et sur le contrôle en général, peut autoriser la résolution sur des structures hors contextes partielles obtenues en cours d'analyse. On peut ainsi éliminer des ambiguïtés locales et éviter de multiplier des solutions qui seraient de toutes façons annulées par les contraintes fonctionnelles.

1.3.3. Grammaires catégorielles d'unification

Les grammaires catégorielles d'unification ou UCG (Unification Categorial Grammars) [USZKOREIT 86] [CALDER 88] [BOUMA 88] sont nées du mariage de la théorie syntaxique des grammaires catégorielles et de la théorie sémantique de représentation du discours de Kamp [KAMP 81]. Elles ont en commun avec les deux théories présentées ci-dessus une très forte utilisation du lexique (liée à l'usage de catégories complexes), et des liens très étroits entre les représentations sémantiques et syntaxiques. Elles s'en distinguent essentiellement par la partie syntaxique, qui ne s'appuie pas sur un modèle hors contexte mais sur les grammaires catégorielles.

Dans ces grammaires, une catégorie est soit :

- une catégorie élémentaire : leur nombre est très limité, nous considérons ici n (noms) s (phrases) et np (groupes nominaux) ;
- A/B ou A et B sont des catégories.

Chaque entrée du lexique possède évidemment une catégorie et la catégorie d'un groupe de mot est calculée à partir des mots qui la composent grâce à un mécanisme très simple de réduction.

Exemple :

Un nom a la catégorie n, un adjectif la catégorie des objets qui, combinés avec un nom donnent un nom, c'est-à-dire n/n :

```
 n/n     n       ==> n                  (comme la réduction d'une fraction)
(beau, bébé)   ==>(beau bébé)
```

De même un adverbe comme "très" peut qualifier un adjectif, il aura la catégorie `(n/n)/(n/n)` indiquant que "très beau" a la catégorie d'un adjectif :

```
(n/n)/(n/n)  (n/n)  ==>  (n/n)
```

Une UCG permet la manipulation de variables désignant des composants non spécifiés. Ces variables seront affectées par unification (à la Prolog) de la catégorie avec d'autres pendant une réduction.

L'objet linguistique de base d'une UCG est le *signe*.Un signe est composé de trois parties et a le format général W:C:S où :

W est le composant morphologique (ou phonologique) ;
C est le composant catégoriel ;
S est le composant sémantique.

Exemples :

Dans ces exemples, les variables sont notés avec des majuscules.

```
bébé:n:[x]bébé(x)
beau+W:n/(W:n:[x]P):[x][beau(x), P]
```

On notera sur le deuxième exemple que les composants d'un signe peuvent contenir des signes. En effet, la règle de réduction présentée succinctement ci-dessus pour les catégories est étendue aux signes sous la forme de deux règles, *application droite* (pre) et *application gauche* (post) :

(pre) : (W:C/A:S A) ==> W:C:S
(post) : (A W:C/A:S) ==> W:C:S[1]

L'application d'une de ces règles implique la mise en correspondance de deux signes (désignés par A dans les règles et appelés *partie active*). Cette mise en correspondance est réalisée par une unification à la Prolog.

Le langage utilisé pour noter la partie sémantique d'un signe est appelé InL (pour Indexed Language) car toute formule possède un index (une variable) typé, dont la formule explicite les propriétés (sont état, sa modification,...).

Exemples :

Index	Formule	Expression	Type
[e]	PROMENE(e, x)	"Pierre se promène"	événement
[x]	BEBE(x)	"un bébé"	objet dénombrable
[m]	BOUILLIE(m)	"de la bouillie"	objet non dénombrable
[s]	DORT(s)	"le bébé dort"	état

L'introduction de cet index dans les formules logiques permet de manipuler une formule comme une instance particulière d'un objet sémantique. On peut

[1]L'application gauche est parfois notée avec un \, on écrit : (A W:C\A:S) ==> W:C:S

ainsi distinguer un événement de sa description et attribuer à l'événement des propriétés particulières ou expliciter des relations entre événements.

Exemples :

"Pierre se promène s'il fait beau."

```
[e][TEMPS(e, BEAU) ==> PROMENE(e, Pierre)]
```

On pourrait se contenter de la description :

`TEMPS(BEAU) ==> PROMENE(Pierre)` mais ce n'est que la description de l'événement, on n'a aucun moyen de faire référence explicite à l'événement lui-même.

Exemple d'analyse de la phrase "le bébé mange la bouillie" :

Dans cet exemple, le composant morphologique n'est pas toujours spécifié. On ajoute par contre un composant *ordre* après le composant sémantique de la partie active afin de préciser si on peut utiliser l'application droite (pre), l'application gauche (post) ou les deux (on ne précise pas d'ordre).

Lexique :

```
a.    le:np/(n:[a]S:pre):[a][S, DEFINI(a)]
b.    bébé:n:[x]BEBE(x)
c.    la:np/(n:[a]S:pre):[a][S, DEFINI(a)]
d.    bouillie:n:[m]BOUILLIE(m)
e.    mange:
      ((s/np:[a]A:post)/np:[b]B:pre):
      [e][[a]A, [b]B, MANGE(e, a, b)]
```

Par unification de la partie active de `le : (n:[a]S:pre)` et du signe de bébé (`x` s'unifiant avec `a` et `S` avec `BEBE(x)`), on obtient :

```
le bébé:np:[x][BEBE(x), DEFINI(x)]
```

de même pour `la` et `bouillie` :

```
la bouillie:np:[m][BOUILLIE(m), DEFINI(m)]
```

On peut ensuite unifier la partie active de `mange` avec `la bouillie` et obtenir :

```
((s/np:[a]A:post):
[e][[a]A, [m][BOUILLIE(m), DEFINI(m)], MANGE(e, a, m)]
```

Enfin, par unification avec `le bébé`, on a :

```
s:
[e][[x][BEBE(x), DEFINI(x)],
    [m][BOUILLIE(m), DEFINI(m)],
    MANGE(e, x, m)]
```

Le mécanisme de réduction présenté ci-dessus peut également être vu comme l'application d'une fonction (une catégorie) à un argument (une autre catégorie), c'est-à-dire qu'un mot est considéré comme un foncteur pouvant prendre comme arguments d'autres mots. Cette façon de considérer les groupes de mots est appelée par les anglo-saxons "head-modifier approach", dans le sens où le foncteur est la tête de la phrase et où ses arguments sont des modificateurs. Nous reparlons plus en détail de cette approche au paragraphe suivant, signalons simplement que les grammaires catégorielles peuvent être vues comme une formalisation mathématique de cette approche, dans le sens ou elles permettent de décrire une catégorie par une formule et qu'elles donnent les moyens de combiner ces formules pour en obtenir de nouvelles.

Les UCG ont été utilisées dans des applications d'envergure dans le cadre d'un projet ESPRIT de la Communauté Economique Européenne. D'importantes

grammaires ont été développées pour l'anglais et le français. Leur principal intérêt est l'intégration simple des différents niveaux de représentation et la construction des structures associées en parallèle grâce à une opération unique : l'unification. Mais ce parallélisme strict ne correspond pas toujours à une réalité linguistique : il peut être parfois difficile d'associer une interprétation sémantique à une opération syntaxique.

1.4. Grammaires de dépendances

Parallèlement aux développements importants des grammaires de constituants (et principalement du niveau hors-contexte), une autre famille de grammaires connaissait un essor plus confidentiel : les grammaires de dépendances que les anglo-saxons qualifient de "head-modifiers grammars". Nous avons choisi d'utiliser ce modèle pour la construction de structures syntaxiques. Après avoir présenté les grands principes de cette théorie et souligné les avantages et les inconvénients par rapport aux structures de constituants, nous présentons le constructeur de structures de dépendance dont nous disposons en mettant en valeur ses qualités et ses limites.

1.4.1. Présentation

La principale caractéristique de ces grammaires par opposition aux grammaires de constituants est que la structure d'une phrase est vue comme un mot (par exemple le verbe), appelé la *tête*, auquel sont attachés des *modificateurs* (les noms sujet et complément). Les modificateurs peuvent à leur tour posséder des modificateurs et la structure d'une phrase est donc une arborescence. On peut constater la similitude de la théorie X-barre de Chomsky avec ce modèle : dans cette théorie, tous les syntagmes principaux sont supposés avoir la même structure constituée de modificateurs et d'une tête, elle-même pouvant se décomposer en une nouvelle tête suivie de modificateurs (voir §1.3.1).

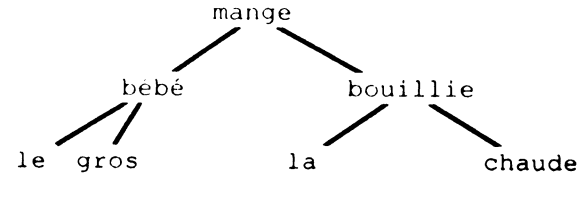


Figure 1.15 : Structure de dépendances

Le concept fondamental associé aux structures de dépendances [TESNIERE 59] est celui de *relation* entre les mots. Etant donnés deux mots du langage, on établit entre eux une relation de dominé (*dépendant*) à dominant (*gouverneur*) et on peut représenter cette relation par un arc entre deux nœuds, chaque nœud étant étiqueté par un mot.

Exemple :

Dans la phrase "le gros bébé mange la bouillie chaude ", "mange" est le gouverneur et "bébé" et "bouillie" sont ses dépendants. Eux-mêmes ont des dépendants, d'où la structure de la figure 1.15 qui peut également s'écrire sous forme parenthésée : `((le gros) bébé) mange ((la) bouillie (chaude))`

On peut également considérer que le gouverneur est un *foncteur* appliqué à des *arguments* (les dépendants), ce qui revient à considérer une relation comme une fonction à valeur booléenne. La structure de la figure 1.15 pourrait aussi s'écrire : mange(bébé(le, gros), bouillie(la, chaude)), avec l'inconvénient de perdre l'ordre initial des mots de la phrase.
On s'attache en effet à conserver cet ordre dans l'arborescence : cela permet de retrouver la phrase par un simple parcours infixé gauche de l'arbre, et évite de nombreux calculs dans la construction de cet arbre (voir §1.4.2). On dit que l'arborescence est *projective* ; on verra que cette propriété ne peut être maintenue qu'au prix d'une perte de pouvoir d'expression des grammaires (toutes les phrases d'une langue n'ont pas une structure projective).
On définit donc, pour un gouverneur donné, ses dépendants droits (Dd) et ses dépendants gauches (Dg). Seront Dg (resp. Dd) d'un gouverneur tous les mots situés à gauche (resp. à droite) de ce gouverneur dans la phrase, le mot le plus à gauche (resp. le plus à droite) devenant bien sûr le dépendant le plus à gauche (resp. le plus à droite).

Une *grammaire de dépendances* sur un vocabulaire V est constituée : [VEILLON 70]

- D'une famille de parties C_i de V telle que l'union des C_i soit égale à V.
- D'un ensemble de règles des deux formes suivantes :
  i) *(X)
  ii) $X(X_1, ...X_i, *, X_{i+1}, ..., X_n)$

Les C_i sont les *catégories lexicales* des paragraphes précédents : chaque mot du vocabulaire appartient à une catégorie qui caractérise son comportement syntaxique. Les X_i sont des noms de catégories (Det, Nom, Adjectif, Verbe,...). L'astérisque sert à indiquer la place du gouverneur par rapport à ses dépendants ; ainsi dans une règle de type ii), $X_1...X_i$ sont les dépendants gauches du gouverneur X et $X_{i+1}...X_n$ ses dépendants droits. Si n=0, la règle s'écrit X(*) et est une règle terminale ; les règles de type i) sont les règles initiales.

Exemple de grammaire :

C_1=Verbe={boit, mange} ; C_2=Nom={bébé, chat, bouillie, lait}
C_3=Det={le,la} ; C_4= Adjq={gros,roux,chaude,sucré}

Règles :

```
*(Verbe)
Verbe(Nom,*,Nom)
Nom(Det,*)
Nom(Det,*,Adjq)
Det(*)
Verbe(*)
Adjq(*)
Nom(*)
```

Cette grammaire permet de construire la structure de dépendance de la figure 1.15.

Les grammaires de dépendances sont des grammaires génératives, le principe de génération est le suivant :

1- On choisit une règle de type i) (choix du gouverneur principal),

2- On choisit une règle de type ii) dont le gouverneur est une feuille de la structure courante et on remplace cette feuille par la structure associée à la règle (ce qui revient à ajouter des fils à la feuille sélectionnée). On obtient une structure correcte si la structure courante ne contient que des "*" aux feuilles. On trouvera figure 1.16 un exemple de dérivation pouvant correspondre à la phrase "le bébé mange la bouillie" dans la grammaire de l'exemple ci-dessus.

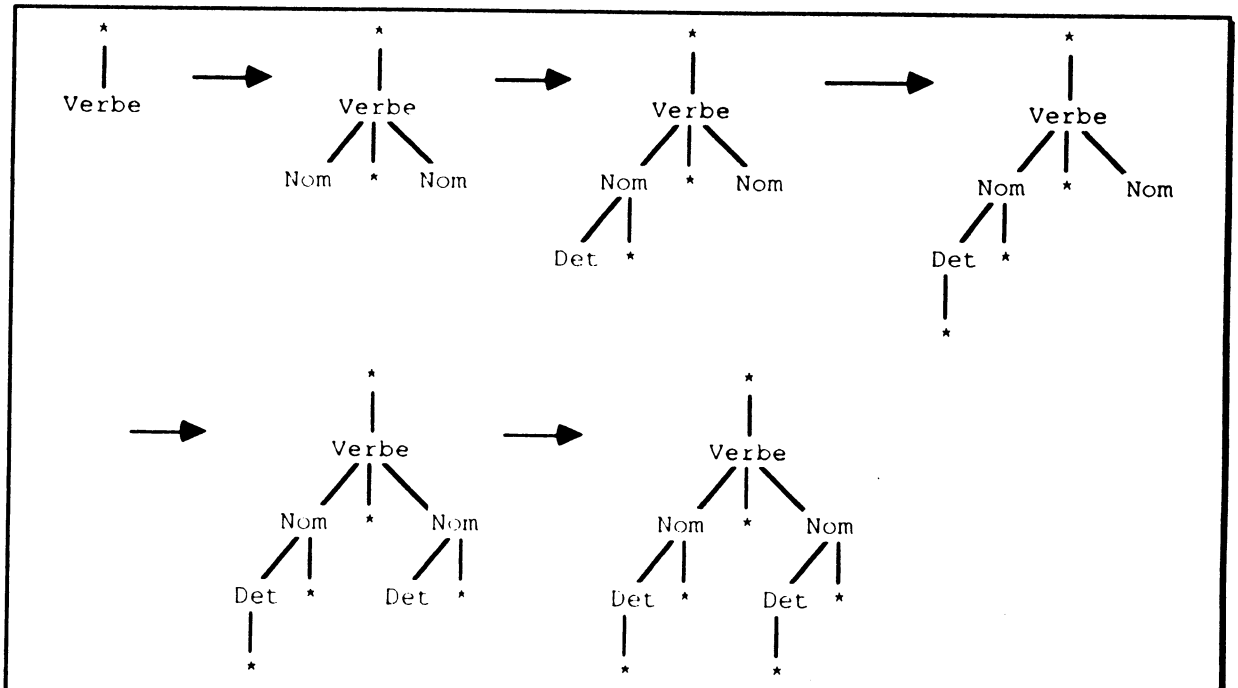


Figure 1.16 : Dérivation dans une grammaire de dépendances

Remarques :

- Un inconvénient de cette forme d'écriture des grammaires de dépendances est que pour un gouverneur donné la grammaire doit comporter autant de règles qu'il y a de combinaisons possibles de dépendants sous ce gouverneur. Si on veut par exemple engendrer des groupes nominaux comprenant au moins un nom et un déterminant, mais 0,1, ou 2 adjectifs avant le nom et 0 ou 1 adjectif après le nom, on aura la grammaire :

```
Nom(Det,*)                    Nom(Det,*,Adjq)
Nom(Det,Adjq,*)               Nom(Det,Adjq,Adjq,*)
Nom(Det,Adjq,*,Adjq)          Nom(Det,Adjq,Adjq,*,Adjq)
```

  On verra au §1.4.2 un formalisme permettant d'éviter cette multiplicité.
- Cette forme d'écriture ne peut s'appliquer qu'aux grammaires projectives et impose que le nombre de dépendants d'un gouverneur donné soit borné (puisque le nombre de règles de la grammaire est fini). Bien que cette limite existe toujours plus ou moins dans les phrases d'une langue, elle n'est pas toujours facile à déterminer a priori.

Les grammaires de dépendances sont à placer, en termes de puissance d'expression, au même rang que les grammaires hors contextes. [GAIFMAN 65] démontre leur équivalence faible (tout ce qui peut être décrit par une grammaire hors contexte peut être décrit par une grammaire de dépendances et vice-versa, mais les structures construites ne sont pas identiques).

Certaines propriétés syntaxiques attachées aux groupes syntagmatiques ne peuvent pas facilement être attachées à un mot particulier d'une structure de dépendances ; ainsi un adverbe qui modifie toute une phrase peut difficilement être considéré comme un modificateur du verbe qui gouverne la phrase. Plus généralement, les structures de dépendances ne distinguent pas les propriétés d'un gouverneur d'un groupe de mots et les propriétés du groupe de mots pris comme un tout. C'est une des principales critiques du modèle gouverneur-dépendant.

Mais d'autres propriétés linguistiques sont au contraire plus facile à noter sur une structure de dépendances que sur une structure de constituants : notons notamment l'accord en genre et en nombre. Dans un groupe nominal, les dépendants du nom (déterminants, adjectifs, quantificateurs) s'accordent en genre et en nombre avec le nom qui les gouverne directement. Les structures fonctionnelle et logique sont également assez immédiates à coder sur une structure de dépendances : il suffit dans la plupart des cas d'ajouter un arc étiqueté entre le verbe et ses dépendants principaux (on trouvera un exemple très simple figure 1.17).

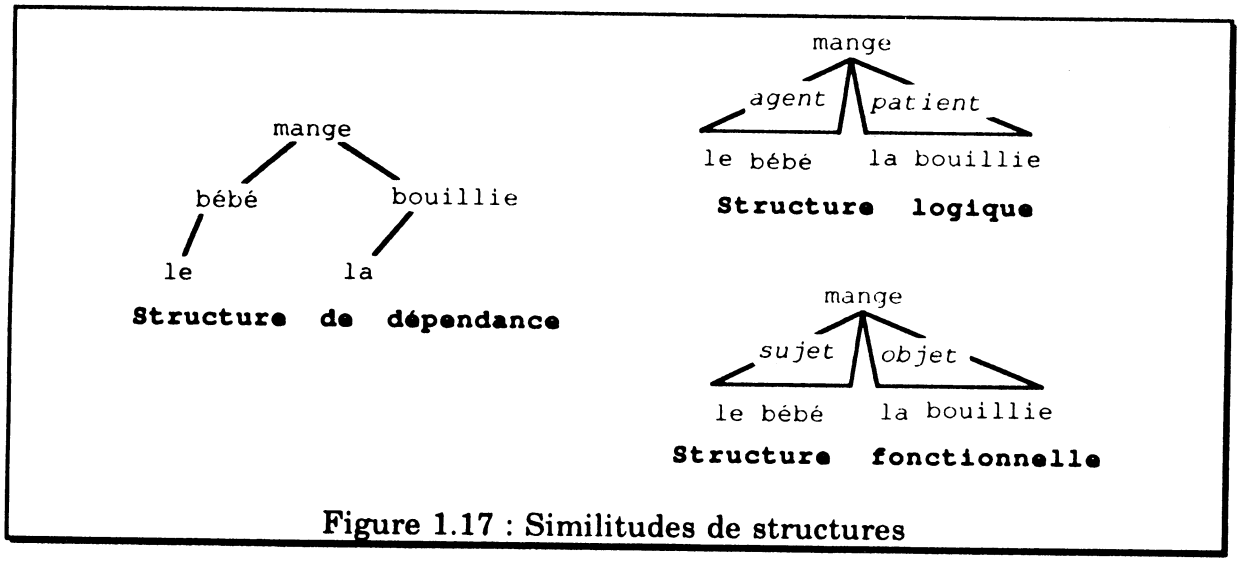


Figure 1.17 : Similitudes de structures

Enfin, une structure de dépendances, grâce aux aspects de hiérarchisation des mots d'une phrase, correspond bien à la notion intuitive qu'on a d'une description : on a un mot principal, le verbe par exemple, et des modificateurs qui précisent l'action, l'événement, l'état décrit par le verbe. De même le nom indique un objet, un sentiment, un lieu et les adjectifs et compléments de nom affinent sa description. On peut ainsi considérer une information minimale, par exemple (bébé) mange (bouillie) et la compléter par ajout de modificateurs pour donner ((le beau) bébé (de ((la) voisine)) mange (goulûment (la) bouillie (de (légumes (frais))).

Tous ces critères sont assez subjectifs mais il faut souligner l'importance de la notion de gouvernement dans toutes les théories syntaxiques récentes basées sur le modèle hors contexte : la théorie X-barre, déjà citée, systématise la notion de tête de syntagme ; la théorie du gouvernement et liage [CHOMSKY 82] s'appuie sur la notion de gouvernement ; les GPSG utilisent un symbole spécial H pour désigner la tête d'un syntagme et lui confère un rôle particulier par le biais de la convention du trait de tête ; les LFG, dans le système d'instanciation des variables, associent une variable unique au constituant principal d'un syntagme ; enfin, comme nous l'avons déjà dit, les UCG peuvent être considérées comme relevant de l'approche gouverneur-dépendant.

1.4.2. Le constructeur de structure de dépendances de PILAF

Les structures de dépendances ont été beaucoup utilisées par les écoles russe [KULAGINA 67], finlandaises [JÄPPINEN 88] [VOLKONEN 87], tchécoslovaque [HAJICOVA 88], ainsi qu'en Allemagne [KUNZE 75] [HELLWIG 80, 86]. En France, le CETA [VEILLON 70] [VAUQUOIS 75] a, depuis le début des travaux en traduction automatique, refusé de trancher entre dépendances et constituants et choisi de construire une structure multi-niveaux intégrant les deux. Les motivations du choix d'une telle méthode sont liées aux limitations des grammaires de dépendances citées ci-dessus. Pour construire une telle structure, on utilise un noyau hors contexte dans lequel chaque partie droite de règle porte un indicateur pour indiquer quel est le gouverneur. Cet indicateur est reporté sur l'arc correspondant de la structure de constituants et on peut obtenir la structure de dépendances associées simplement en faisant remonter les feuilles le long des arcs ainsi marqués.
Exemple : (voir figure 1.18)

Grammaire (l'étoile matérialise l'indicateur) :
S -> GN + GV(*)
GN -> Det + Nom(*)
GV -> Verbe(*) + GN
Avec la structure de constituants (un trait plus épais matérialise les arcs marqués) : on obtient la structure de dépendance :

Figure 1.18 : Passage constituants -> dépendances

Depuis toujours, l'équipe TRILAN a utilisé les grammaires de dépendances pour construire des structures syntaxiques. L'analyseur de dépendance de PILAF est dû à [COURTIN 77]. Il transforme directement la chaîne d'entrée en une structure de dépendances sans utiliser de structure intermédiaire. L'algorithme est donc basé sur une grammaire de dépendances mais n'utilise pas directement cette grammaire comme le font les analyseurs basés sur des grammaires hors contexte. En effet, comme on l'a vu au §1.4.1, ces

grammaires imposent la description de toutes les configurations possibles des dépendants pour un gouverneur donné. Pour pallier cet inconvénient on introduit la notion de *relation de dépendances* entre deux catégories syntaxiques.

1.4.2.1. Relations de dépendances

Les relations de dépendances (RD) représentent non seulement la relation gouverneur-dépendant mais également :

- la position du dépendant par rapport au gouverneur (à gauche ou à droite),
- la possibilité pour un gouverneur d'avoir plusieurs dépendants du même côté et les positions relatives de ces dépendants.

Exemple :

Dans "Le beau bébé brun", l'analyse morphologique donne les catégories `Det Adjq Nom Adjq`. Les relations doivent préciser que l'`Adjq` peut se trouver à gauche ou à droite du `Nom`, et que le `Det` est plus à gauche que l'`Adjq`.

On attache à chaque relation un vecteur de poids (entiers positifs ou négatifs) et on écrit `GOUV*DEP:=(`$X_1, \ldots, X_n$`)`, ce qui signifie qu'il peut y avoir 0,1,..n dépendants de la catégorie `DEP` du gouverneur de la catégorie `GOUV`.
La valeur des poids permet également, pour deux catégories dépendant d'un même gouverneur, de déterminer les positions relatives de ces deux catégories sous le gouverneur.

Exemple :

```
Nom*Adjq:=(-15,-14,+18)
Nom*Det:=(-16)
```

Les poids positifs désignent les dépendants droits et les poids négatifs les dépendants gauches. Le poids dans la deuxième relation précise que le déterminant sera toujours placé avant les adjectifs. Dans la première, on voit qu'on pourra avoir deux adjectifs avant le nom, mais que rien ne peut s'insérer entre les deux (aucun entier entre -15 et -14).
Les deux relations ci-dessus sont donc équivalentes à la grammaire de dépendances :

```
Nom(*)                          Nom(Det,*)
Nom(*,Adjq)                     Nom(Det,*,Adjq)
Nom(Adjq,*,Adjq)                Nom(Det,Adjq,*,Adjq)
Nom(Adjq,Adjq,*,Adjq)           Nom(Det,Adjq,Adjq,*,Adjq)
Nom(Adjq,Adjq,*)                Nom(Det,Adjq,Adjq,*)
Nom(Adjq,*)                     Nom(Det,Adjq,*)
```

1.4.2.2. Fonctionnement de l'analyseur

Les entrées de cet algorithme sont :

- la suite $X_1 \ldots X_n$ des catégories syntaxiques calculées par l'analyseur morphologique. On ajoute en tête de cette suite la catégorie `PHRA` afin de déterminer les gouverneurs possibles de la phrase (pour "lancer" l'algorithme).
- l'ensemble des relations avec les vecteurs de poids associés. Si les gouverneurs possibles d'une phrase sont par exemple la conjonction de coordination et le verbe, on aura les deux relations de lancement :
  `PHRA*VERB:=(+1)`

PHRA*COCO:=(+1)
Comme on ne peut avoir qu'un seul gouverneur pour une phrase, ces deux relations sont exclusives ; on le signale en mettant le même poids aux dépendants possibles de la catégorie PHRA.

L'algorithme construit toutes les structures de dépendances projectives compatibles avec ces relations. La projectivité n'est pas absolument obligatoire avec ce formalisme mais elle simplifie notablement l'algorithme et contribue à son efficacité. En effet, sans cette contrainte, l'analyseur serait amené à essayer d'accrocher "le" à "bouillie" dans la phrase : "le bébé mange la bouillie".
La construction est récursive et descendante :

1- Pour chaque gouverneur possible de la phrase, c'est-à-dire pour chaque mot dont la catégorie apparaît comme dépendante de la catégorie PHRA :
   2- construire tous les sous arbres gauches puis tous les sous arbres droits (récursivement)
   3- bâtir les structures finales avec les structures partielles obtenues : rattacher les sous arbres au gouverneur.

Exemple :

```
Le    bébé   brun   mange  la   bouillie   de    légumes.
Det   Nom    Adjq   Verbe  Det  Nom        Prep  Nom
```

On a les relations :

```
PHRA*Nom:=(+1)              PHRA*Verbe:=(+1)
Verbe*Nom:=(-20,+20)        Verbe*Prep:=(+30)
Nom*Det:=(-16)              Nom*Adjq:=(-15,-14,+18)
Nom*Prep:=(+20)             Prep*Nom:=(+7)
```

Etape 1 - : Verbe est le seul gouverneur de phrase compatible avec ce tableau (les autres conduiraient à des échecs), en effet Verbe ne peut dépendre que de PHRA
Etape 2 - : Comme on ne veut que des structures projectives, on peut calculer les sous arbres droits indépendamment des sous arbres gauches. Prenons le cas des dépendants droits $X_{j+1}...X_n$ d'un gouverneur X_j donné (ici "mange"). On détermine toutes les listes de dépendants possibles de X_j pris parmi $X_{j+1}...X_n$. On pourra avoir des listes de 1, 2, ... k dépendants. Pour pouvoir construire ces listes, on doit s'assurer qu'il existe une liste correspondante de poids strictement croissante. Avec "mange", on aura les listes :
1 dépendant : (Nom(+20)) (Prep(+30)) (Nom(+20))
2 dépendants : (Nom(+20), Prep(+30))
Ces listes permettent de construire 3 arbres partiels :

```
() mange (bouillie)                    (a)
() mange (de)                          (b)
() mange (légumes)                     (c)
() mange (bouillie, de)                (d)
```

Comme il reste des mots non raccordés, on appelle récursivement l'analyse sur les tranches de la phrase déterminées par les listes. Ainsi pour (a) (b) et (c) on appelle l'analyse avec "la bouillie de légumes", seul le gouverneur change ; pour (d) on l'appelle avec "la bouillie" puis avec "de légumes".
Les arbres (b) et (c) conduisent à des échecs : on ne parviendra pas à placer le nom "bouillie". L'arbre (a) permet d'obtenir :

() mange ((la) bouillie (de (légumes)))
et l'arbre (d) :
() mange ((la) bouillie, de (légumes))
qui est syntaxiquement correcte mais sémantiquement fausse, on peut manger de bon appétit mais pas de légumes.
L'analyse des dépendants gauches de "mange" permet d'obtenir l'unique structure : (le) bébé (brun)
Etape 3 - : On construit toutes les combinaisons possibles avec les structures obtenues à l'étape 2, on obtient donc :
((le) bébé (brun)) mange ((la) bouillie (de (légumes)))
((le) bébé (brun)) mange ((la) bouillie, de (légumes))

Cet algorithme, de par sa simplicité, donne de bons résultats en terme d'efficacité sur des phrases simples. L'analyse de la phrase ci-dessus est instantanée, même sur un micro-ordinateur bas de gamme. Il permet également la construction de structures incorrectes comme : (bébé) mange (bouillie). De telles structures peuvent suffire en entrée d'un modèle sémantique qui serait uniquement intéressé par l'information minimale que véhicule la phrase.

Par contre, le langage d'écriture des règles (ici des relations) a un très gros défaut qui le rend quasi inutilisable sur des grammaires plus complexes : il ne permet pas de tenir compte du contexte dans la construction des structures. On ne peut que décrire la position relative d'un dépendant sous un gouverneur et interdire la présence simultanée de certains dépendants en jouant sur les valeurs des poids.
On ne peut pas imposer la présence d'un dépendant sous un gouverneur, cela a pour conséquence de produire une grande quantité de structures incohérentes. Ainsi si on veut coordonner les noms, les verbes et les adjectifs on aura les règles de coordination :

Coco*Nom := -1, +1
Coco*Verbe := -1, +1
Coco*Adjq := -1, +1

Comme la coordination de deux catégories peut remplacer la catégorie, on aura également :

Verbe*Nom:=(-20,+20) et Verbe*Coco:=(-20,+20)
PHRA*Coco := +1
Nom*Adjq:=(-15,-14,+18) et Nom*Coco:=(-15,-14,+18)

Avec ces règles, l'analyse de "le chien et le chat mangent et boivent" produit 19 arbres dont un seul peut être considéré correct.
Il est vrai que peu de formalismes manipulent correctement la coordination, mais ces problèmes se retrouvent pour toutes les catégories dont la configuration de dépendants est particulière (préposition, pronom relatif,...)[1].

[1]Pour pouvoir utiliser l'analyseur dans un protoype de vérificateur syntaxique [STRUBE DE LIMA 89, 90], nous avons réglé ce problème en adjoignant à chaque catégorie une information précisant la géométrie des arbres dont elle est le gouverneur. Ainsi nous imposons que la préposition n'ait qu'un seul dépendant à droite, que la conjonction de coordination ait un dépendant à droite et un à gauche,... Cette technique permet d'éliminer bon nombre d'arbres incohérents mais n'est pas homogène avec la méthode des relations de dépendances.

Cette faiblesse de pouvoir d'expression des règles rend de plus difficile l'extension de l'analyseur vers le niveau sémantique : il n'est pas évident d'exprimer des relations sémantiques entre mots ou groupes de mots en ayant pour seule information un dépendant et son gouverneur.

Enfin, l'algorithme utilisé est un analyseur, c'est-à-dire qu'il ne produit aucun résultat sur une entrée incorrecte (c'est rarement le cas car il est très tolérant, mais la structure obtenue n'est pas souvent interprétable). Cet aspect est gênant compte tenu de nos objectifs en matière de détection et correction d'erreurs. Il nous paraît plus souhaitable d'avoir un outil plus strict mais capable de fournir des résultats partiels sur lesquels on puisse bâtir une stratégie de correction.

Pour ces raisons, nous avons choisi de redéfinir le langage d'écriture des règles de dépendance (cf chapitre 2) et d'utiliser un formalisme de représentation des connaissances qui puisse intégrer des informations sémantiques (cf chapitre 3). Cette redéfinition entraîne l'écriture d'un nouvel algorithme d'analyse.

1.5. Conclusion

Nous donnons en guise de conclusion quelques remarques sur les aspects qui nous paraissent les plus marquants dans les évolutions récentes des théories syntaxiques.

Importance du lexique

Les évolutions récentes en traitement automatique de la langue naturelle (TALN) accordent une importance de plus en plus grande au contenu des items lexicaux. On assiste à un glissement prononcé de la compétence linguistique de la grammaire vers les dictionnaires. Ce glissement nous paraît résulter de plusieurs facteurs :

- l'accroissement de la compétence des systèmes de TALN induit des tentatives de plus en plus ambitieuses vers des systèmes pratiques, voire industriels. De tels systèmes supposent des lexiques de taille importante dont la construction fait apparaître la multitude de cas particuliers et d'exceptions qui caractérisent les langues naturelles. Les grammaires n'étant censées modéliser que les régularités de ces langues, il est normal que la représentation des cas particuliers soit dévolue au lexique.
- l'intégration de connaissances de type sémantique relève plus naturellement d'un lexique. La sémantique est soit propre à un mot soit décrite par les relations que ce mot peut entretenir avec d'autres.

Catégories et sous catégories

Un rôle très important est accordé à la catégorie d'un mot. Cet aspect est lié au précédent : un mot du lexique est principalement caractérisé par sa catégorie (du moins dans ses liens avec la grammaire), et nombre de régularités qui étaient représentées par des règles (ou des transformations), se trouvent, dans les formalismes modernes, modélisées par des mécanismes de manipulation de catégories complexes.

Le rôle de la tête

Tous les formalismes récents qui n'utilisent pas l'approche "head-modifiers" ont reconnu l'importance de la hiérarchie entre les différents composants d'un même syntagme. Il y a toujours un composant (la tête de syntagme) qui joue un rôle particulier et qu'il faut pouvoir distinguer pour établir clairement toutes les propriétés syntaxiques et (surtout) sémantiques d'un syntagme. Ajoutons que là encore le rôle important du lexique, entraînant la présence d'un grande quantité d'informations sur les mots (donc sur les feuilles des structures de constituants) , n'est pas étranger à la prépondérance accordée au rôle de la tête. La théorie des dépendances fait de ce rôle la base de la représentation syntaxique d'une phrase, mais elle doit être quelque peu enrichie pour permettre vraiment l'analyse d'une langue.

Sémantique et unification

Enfin, les théories récentes intègrent toutes des aspects sémantiques dans leur traitements :

- par ajout d'informations sémantiques dans les lexiques ;
- par utilisation au niveau de la grammaire de mécanismes qui permettent de manipuler ces informations. La représentation sémantique finale d'une phrase est construite pendant l'analyse syntaxique (UCG) ou après (GPSG, LFG) mais sur la base de structures calculées par le module syntaxique.

Cette intégration repose sur la fusion dans une même structure des informations de toutes natures (morphologique, syntaxique et sémantique) et sur l'utilisation quasi exclusive d'une opération unique de manipulation de ces structures : l'unification.

2

Etude d'un constructeur de structures de dépendances

Les formalismes modernes de description de la langue naturelle accordent une importance de plus en plus grande au contenu des items lexicaux. Il s'ensuit un appauvrissement de la grammaire qui n'est plus qu'un outil de guidage dans la construction de structures qui sont plutôt de nature sémantique. Dans certains cas même, la grammaire disparaît peu ou prou au profit d'un mécanisme très général d'assemblage des items lexicaux : adjonction et substitution dans [SCHABES 88] pour les structures de constituants, application et unification dans les UCGs, et unification dans les grammaires dépendancielles d'unification de [HELLWIG 86].

Nous pensons qu'il est important de conserver un module grammatical pour deux raisons essentielles :

- d'un point de vue informatique la description de structures syntaxiques par le biais de règles simples permet de conserver un certain contrôle sur le déroulement de l'analyse. Ce contrôle autorise la mise en œuvre de stratégies d'analyse variées : on peut ainsi décider d'effectuer certaines vérifications à un moment donné de l'analyse (par exemple dès la reconnaissance d'un groupe nominal complet on pourra s'assurer de la bonne réalisation des règles d'accord en genre et en nombre) ; on peut également lancer la construction d'une partie de la structure sémantique de la phrase sur la base d'une structure syntaxique partielle.
- d'un point de vue linguistique, s'il est vrai qu'on peut de manière élégante décrire les compléments essentiels d'un verbe grâce à une sous catégorisation ou par ajout de structures complexes à l'entrée lexicale du verbe (cf [HELLWIG 86]), il nous paraît difficile de décrire tous les modificateurs (adverbes, compléments circonstanciels) qui peuvent lui être adjoints autrement qu'avec des règles de grammaire d'une portée générale.

Nous avons donc choisi de définir un langage qui permette de décrire des structures syntaxiques simples : des arbres de dépendances et d'augmenter ces arbres avec un modèle de représentation de la connaissance qui puisse rendre compte de phénomènes linguistiques complexes. Dans ce chapitre, nous décrivons le langage de réécriture d'arbres que nous avons défini et nous renvoyons au chapitre suivant la description du modèle de représentation de la connaissance.

2.1. Un langage de réécriture d'arbres

Notre but est la construction de structures de dépendances. Il ne s'agit pas bien sûr d'écrire un programme qui construise directement ces structures mais de définir un langage de description de ces structures qui soit indépendant de toute considération de mise en œuvre et d'écrire un interprète de ce langage. Nous utilisons donc une approche déclarative avec ses avantages (modularité, souplesse, construction incrémentale des grammaires, ...) et son principal inconvénient (moins efficace que l'approche procédurale).

2.1.1. Présentation de l'idée de base

Le problème posé peut s'énoncer comme suit : étant donné un ensemble C de catégories lexico-syntaxiques (CLS), comment décrire les rapports de gouverneur à dépendant existant entre ces catégories de manière à construire, pour une phrase codée par une liste de CLS[1], l'arbre de dépendances correspondant à la structure syntaxique de la phrase. La description de ces rapports, sous quelque forme que ce soit, constitue une grammaire de dépendance.
Nous avons vu au §1.4 deux langages d'écriture de grammaires de dépendances : le langage des grammaires originelles de [TESNIERE 59] et celui des relations de dépendances de [COURTIN 77]. Dans les deux cas il s'agit de grammaires génératives : on part de la racine de l'arbre et on essaye récursivement de décrire les dépendants jusqu'à obtenir un arbre correspondant à la phrase. Nous avons souligné les inconvénients de ces deux langages : dans le premier l'obligation de décrire toutes les configurations possibles de dépendants sous un gouverneur et dans le deuxième la difficulté à contraindre l'utilisation d'une relation en fonction du contexte.

Notre idée est que la relation gouverneur-dépendant existant entre deux catégories ne peut que difficilement être fixée a priori pour deux CLS données et doit être fixée pour deux **instances** de ces catégories dans une phrase donnée. Ainsi, s'il est vrai que le déterminant est toujours gouverné par le nom, il n'est pas correct de rattacher "la" à "vélo" dans la phrase : "la voiture et le vélo". La relation fixant un déterminant comme dépendant d'un nom ne peut donc être exploitée que si le déterminant et le nom apparaissent dans un certain contexte (le déterminant doit, par exemple, apparaître avant le nom et les catégories qui les séparent ne sont pas quelconques).

Cette idée était déjà prise en compte dans les deux langages cités (par la description des configurations de dépendants et par les valeurs de poids) mais nous l'érigeons en principe de base en la généralisant aux arbres de dépendances : la description de la structure de dépendances d'une phrase n'est plus la mise en correspondance d'une chaîne de catégories et d'un arbre mais la mise en correspondance d'une chaîne d'arbres et d'un arbre. Ainsi, un arbre D dominé par un déterminant ne pourra se rattacher à un arbre N dominé par un nom que si D précède immédiatement N et si les dépendants du nom ne comportent pas déjà un déterminant. La méthode d'analyse est donc ascendante : on construit des arbres par assemblage d'autres arbres et la

[1]Nous disposons d'un analyseur morphologique capable de transformer une phrase (une chaîne de caractères) en une telle liste (voir Chapitre 5).

phrase initiale est codée par une liste d'arbres à un nœud étiqueté par la CLS du mot correspondant. L'élément de base du langage est une règle de réécriture dont la partie gauche est une liste d'arbre (les *arbres objets*) et la partie droite l'arbre construit à partir de cette liste (l'*arbre résultat*).

Exemple :

En notant les arbres sous forme parenthésée infixée, on pourra écrire les règles :

```
Det, Nom ==> (Det)Nom                                    (1)
(Det)Nom, Coco, (Det)Nom ==> ((Det)Nom)Coco((Det) Nom)   (2)
```

En appliquant ces règles à la liste d'arbres :

```
Det, Nom, Coco, Det, Nom
```

on pourra dans un premier temps obtenir (règle (1) appliquée 2 fois) la liste :

```
(Det)Nom, Coco, (Det)Nom
```

puis avec la règle (2) :

```
((Det)Nom)Coco((Det)Nom)
```

Le langage obtenu permet d'écrire des grammaires de dépendances (ensemble de règles de réécriture) mais n'est pas à proprement parler un langage de description de structures mais plutôt un langage de construction de structures.

2.1.2. Description du langage

Pour qu'une telle approche soit utilisable, il faut bien sûr donner plus de souplesse au langage dans la partie qui décrit les arbres objets sous peine de retomber sur la verbosité des grammaires de dépendances de Tesnière (voir §1.4.1). La description des arbres objets doit notamment rendre facultative la description de tout ce qui n'est pas significatif pour l'application d'une règle. La description de l'arbre résultat doit quant à elle être très précise et basée sur des références exactes aux nœuds des arbres objets. Afin de pouvoir tester le contexte d'une configuration sans le modifier, les règles ne produisent pas seulement un arbre résultat mais une forêt résultat.

Exemple :

On pourra écrire `a, b, c, d ==> a, (b)c, d` pour exprimer que `b` ne se rattache à `c` que dans un contexte `a-d`.

Enfin, il est indispensable d'autoriser la sélection d'une liste d'arbres (une *forêt*) pour permettre des rattachements répétitifs ou lointains.

Exemple :

Si l'étoile dénote la répétition, on pourra écrire `a, b*, c ==> (a, b*)c` pour exprimer qu'un `a` suivi d'une liste de `b` se rattache au `c` qui suit.

Syntaxe commentée des règles :

Nous donnons ci-dessous la syntaxe BNF (Backus Naur Form) des règles. Nous nous limitons à la partie géométrique (construction des arbres), la syntaxe complète est donnée au §4.4. Les éléments terminaux sont notés en gras, le | dénote l'alternative et les éléments entre [ ] sont facultatifs.

`<Règle>` ::= `<Symbole>` [`<Gauche>` **=>** `<Droite>`]

Le symbole est le nom de la règle, il n'a qu'un rôle documentaire. La partie gauche décrit la liste des arbres objets à laquelle la règle peut s'appliquer et la partie droite décrit les arbres résultats.

`<Droite>` ::= `<Forêt>`

```
<Gauche> ::= ( <LSchema> )
<LSchema> ::= <Schema> [ , <LSchema> ]
```

La seule différence entre les deux parties est que les arbres résultats ne comportent pas de conditions d'occurrence mais seulement une description de la structure arborescente.

```
<Schema> ::= <Arbre> [ <Limite> ] | <VarForêt>
```

Un schéma décrit les arbres objets, on distingue le premier niveau (la racine) pour introduire une limite dont on décrira plus loin le rôle particulier.

```
<Arbre> ::= [ <Forêt> ] <Noeud> [ <Forêt> ]
```

C'est un arbre de dépendances classique, sous forme infixée afin de distinguer les dépendants droits des dépendants gauches. Le nœud désigne bien sûr la racine de l'arbre.

```
<Forêt> ::= ( ) | ( <LssArbre> )
<LssArbre> ::= <ssArbre> [ , <LssArbre> ]
<ssArbre> ::= <Arbre> | <VarForêt>
```

Nous avons pris comme principe directeur que tout ce qui n'est pas précisé dans un schéma est considéré comme quelconque, il n'est donc pas obligatoire de décrire les dépendants d'un nœud particulier. Si on veut imposer une liste de dépendants vide, on doit l'indiquer.

```
<VarForêt> ::= $<Symbole> [ : <EnsCat> ] | 0
```

Les variables de forêt permettent de sélectionner toute une liste d'arbres dans un arbre objet ou toute une liste d'arbres objets. La référence spéciale **0** permet d'imposer une forêt vide là ou une forêt implicite pourrait apparaître (cf §2.2.2). L'ensemble de catégories (EnsCat) indique quelles catégories peuvent étiqueter les **racines** des arbres de la forêt.

```
<Noeud> ::= <Entier> [ : <EnsCat> ]
```

Un nœud particulier est désigné avec une référence unique dans les arbres objets et résultats. Cette référence est un entier positif. L'ensemble de catégories indique quelles catégories peuvent étiqueter le nœud. Ces ensembles constituent les conditions d'occurrence.

```
<EnsCat> ::= { <LCateg> }
<LCateg> ::= <Symbole> [ ; <LCateg> ]
```

Le symbole doit représenter une catégorie lexico-syntaxique valide.

```
<Limite> ::= ? <EnsCat> | / <EnsCat>
```

Voir §2.2.4 pour une description des limites.

Un <Symbole> est un identificateur au sens des langages de programmation.

Exemple simple de grammaire :

```
Det_Nom [
    (1:{Det}, $A:{Adj}, 2:{Nom}) => ( ( 1, $A ) 2 ) ]
Adj_Nom [
    (1:{Adj}, (0, $A:{Adj}) 2:{Nom}) => ( ( 1, $A ) 2 ) ]
Nom_Adj [
    (1:{Nom}, 2:{Adj}) => ( 1 ( 2 ) ) ]
Nom_Verbe [
    (1:{Nom}, () 2:{Verbe}) => ( ( 1 ) 2 ) ]
Verbe_Nom [
    (1:{Verbe} (), 2:{Nom}) => ( 1 ( 2 ) ) ]
Prep_Nom [
    (1:{Prep} (), 2:{Nom}) => ( 1 ( 2 ) ) ]
Nom_Prep [
    (1:{Nom}, 2:{Prep} (3)) => ( 1 ( 2 ( 3 ) ) ) ]
```

```
Verbe_Prep [
    (1:{Verbe}, 2:{Prep} (3)) => ( 1 ( 2 ( 3 ) ) ) ]
```

2.1.3. Hiérarchisation des catégories

Ces règles présentent l'avantage sur de simples relations binaires d'autoriser l'expression du contexte de chaque catégorie mise en jeu. On peut ainsi imposer qu'un gouverneur n'ait qu'un ou deux dépendants, interdire qu'un dépendant d'une certaine catégorie apparaisse plusieurs fois,... On peut également définir des relations binaires doubles, comme pour la conjonction de coordination :

```
Nom_coco [
    (1:{Nom}, 2:{coco}, 3:{Nom}) => ( ( 1 ) 2 ( 3 )) ]
```

Par contre, elles ont l'inconvénient des premières grammaires de dépendances : on devra avoir une règle quasiment pour chaque paire de CLS. Pour éviter ce problème, nous avons choisi de munir l'ensemble de CLS d'une relation d'ordre partiel `sorte-de`, on obtient donc une *hiérarchie* de CLS (figure 2.1).

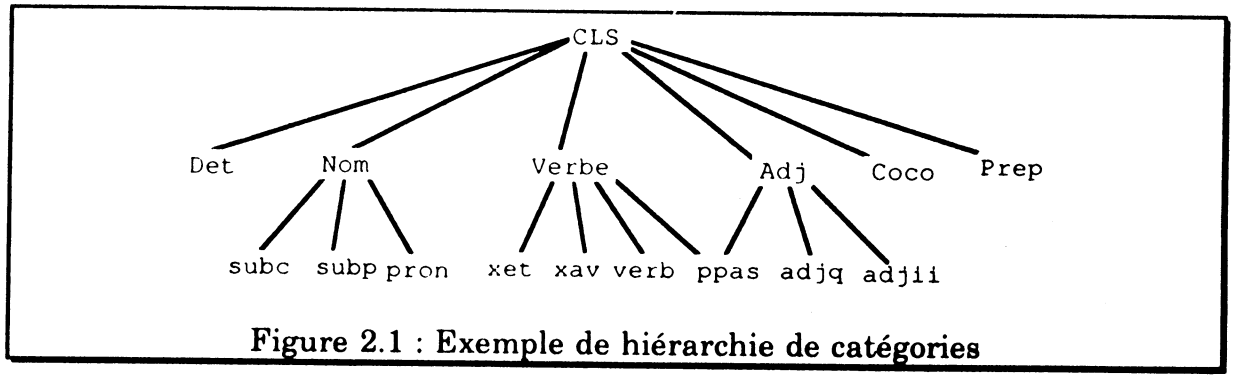


Figure 2.1 : Exemple de hiérarchie de catégories

On a les significations suivantes : Det = déterminants, subc = substantifs communs, subp = substantifs propres, pron = pronoms, xet = auxiliaire être, xav = auxiliaire avoir, ppas = participes passés, adjq = adjectifs qualificatifs, adjii = adjectifs indéfinis, Coco = conjonctions de coordination, Prep = prépositions.

Une telle hiérarchie s'interprète comme suit : `xet`, par exemple, est une `sorte-de Verbe` ; en termes ensemblistes on peut dire que toute unité lexicale qui appartient à la catégorie `xet` appartient aussi à la catégorie `Verbe`. On définit l'unification de deux catégories X et Y comme la catégorie la plus générale qui soit à la fois une `sorte-de` X et une `sorte-de` Y[1]. L'interprétation de l'unification est l'intersection de deux ensembles. Ainsi l'unification de `Verbe` et `Adj` est `ppas`, alors que `subc` et `Verbe` ne s'unifient pas.

[1]Pour que cette catégorie soit unique dans tous les cas, il faut que la hiérarchie ait une structure de demi-treillis inférieur, l'unification de deux catégories est alors définie comme la borne inférieure de ces deux catégories dans la relation d'ordre. Si la hiérarchie n'a pas une telle structure, l'unification peut être un ensemble de catégories (on trouvera plus de détails au chapitre 3 et notamment au §3.1.4).

Si une règle s'applique à une catégorie C, elle s'applique aussi à toute catégorie x qui s'unifie avec C. On peut ainsi écrire des règles très générales du type des règles de la grammaire de l'exemple ci-dessus ou des règles plus spécifiques comme :

```
aux_ppas [
    (1:{xet ; xav}, 2:{ppas}) => ( ( 1 ) 2 ) ]
```

La règle Nom_Verbe par exemple, peut s'appliquer à toutes les paires de catégories suivantes :

```
(subc, xet)     (subp, xet)     (pron, xet)
(subc, xav)     (subp, xav)     (pron, xav)
(subc, verb)    (subp, verb)    (pron, verb)
(subc, ppas)    (subp, ppas)    (pron, ppas)
```

et s'appliquer à des phrases comme "Pierre aime", "il a une voiture", "le bébé mange",... On revient en détail sur cette hiérarchie au chapitre 3.

L'utilisation d'un ensemble ordonné de catégories, plutôt que l'ensemble linéaire classique, donne donc une certaine souplesse pour l'écriture de grammaires de dépendances. Imaginons-nous en présence d'un corpus de texte pour lequel on veut construire un analyseur (ce qui est l'approche la plus courante pour une application de traitement de la langue naturelle).

On commence par définir un ensemble de catégories de base décrivant les mots de manière très précise (en caricaturant on pourrait avoir une catégorie pour chaque unité lexicale du corpus) et on utilise ces catégories pour la construction du lexique.

On regroupe ensuite ces catégories en "sur"-catégories plus générales en fonction des structures que l'on veut construire. Si on ne veut qu'une analyse approximative d'une phrase, on pourra se contenter d'un ensemble très général de "grosses" catégories, regroupant de nombreuses catégories de bases (comme dans l'exemple de grammaire donné au §2.1.2). Si on veut au contraire une analyse plus détaillée on utilisera des regroupements plus fins.

L'idéal est une approche mixte et incrémentale : en effet, le but de la définition d'une hiérarchie est l'exploitation de cette hiérarchie pour l'écriture de règles de construction. On peut donc commencer avec des catégories très générales et écrire les règles appropriées : on obtiendra par exemple le jeu de règles très général donné au §2.1.2. Le test de ces règles sur quelques phrases du corpus mettra en évidence les lacunes : certaines phrases ne peuvent pas être analysées, d'autres donnent trop d'interprétations différentes. On affinera alors le jeu de règles de manière à corriger ces imperfections.

Exemple :

On ajoutera certainement au jeu précédent une règle du type :

```
xet_adjq [ (1:{xet}, 2:{adjq}) => (1(2)) ]
```

pour traiter le rattachement d'un attribut du sujet au verbe être.

L'utilisation de la hiérarchie permet aussi la mise en œuvre d'une certaine forme de sous-catégorisation. On pourra, pour un mot donné, lui appliquer une règle générale ou une règle précise. Ainsi, si on décompose l'ensemble des formes finies des verbes (`verb`) en 4 ensembles :

`VAction`	verbes qui ne sont dans aucune des catégories suivantes
`VEtat`	verbes d'état (être, sembler, paraître)

VPerc verbes de perception (regarder, voir, sentir,...)
VModal verbes modaux (aimer, vouloir, pouvoir,...)

On aura une règle générale pour le rattachement du sujet à un verbe quelconque :

```
Nom_verb [
  (1:{Nom}, () 2:{verb}) => ( ( 1 ) 2 ) ]
```

Mais on aura des règles spécifiques pour le rattachement des groupes susceptibles de suivre chaque type de verbe :

```
objet_direct [
     (1:{VAction; VPerc}(), 2:{Nom}) => (1 ($F, 2)) ]
Verb_Infi [                                  -- regarde passer, doit faire,...
     (1:{VModal; VPerc}, 2:{Infi}) => (1(2)) ]
```

Et si CoSub est la catégorie des conjonctions de subordination :

```
Verbe_CoSub [                        -- je vois qu'il est là, je veux qu'il vienne,...
     (1:{VModal; VPerc}, 2:{CoSub}(3)) => (1(2(3))) ]
```

Cette méthode évite donc le compromis classique entre un ensemble très fin de catégories (qui multiplie les ambiguïtés morphologiques et le nombre de règles syntaxiques) et un ensemble très général (qui multiplie les ambiguïtés syntaxiques).

Cette méthode permet également une certaine robustesse de l'analyse syntaxique. On peut attribuer à un mot inconnu une catégorie : la plus générale (CLS), à laquelle toute règle peut s'appliquer. Le contexte du mot inconnu seul sélectionnera l'application de telle ou telle règle, laquelle déterminera la catégorie que devrait avoir ce mot.

Exemple :

Avec une forêt du type :

```
( 1:{subc}, 2:{coco}, 3:{CLS} )
```

on obtiendra :

```
( (1:{subc}) 2:{coco} (3:{Nom}) )
```

après application de la règle :

```
Nom_coco [
  (1:{Nom}, 2:{coco}, 3:{Nom}) => ( ( 1 ) 2 ( 3 )) ]
```

En effet, l'application d'une règle remplace chaque paire de catégories mises en jeu par le résultat de leur unification (voir §2.2.3). Ainsi l'application de Nom_Coco à la séquence subc - Coco - CLS met en jeu les paires (Nom, subc), (coco, coco) et (Nom, CLS). Elle produit un nouvel arbre où :

subc (résultat de l'unification de Nom et subc) remplace subc,
coco (résultat de l'unification de coco et coco) remplace coco,
Nom (résultat de l'unification de Nom et CLS) remplace CLS.

Les concepts concernant la robustesse de l'analyse font l'objet du chapitre 6, nous ne nous étendons donc pas plus ici.

2.1.4. Quelques problèmes

L'utilisation de ce langage pose deux types de problèmes : ceux liés au langage lui-même et qui sont ceux des langages à base de règles (comment sélectionner une règle, dans quel ordre les appliquer, ...) et ceux liés à la langue naturelle : ambiguïtés lexicales et structurales. Nous allons illustrer ces problèmes à travers l'application de la grammaire de l'exemple du §2.1.2 à un exemple

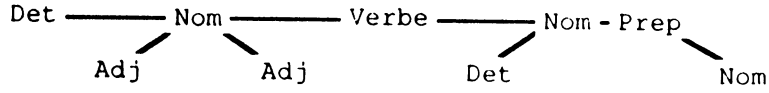
simple de phrase : "le beau bébé blond mange la bouillie de légumes". Nous considérons ici que la liste des CLS associée est (Det, Adj, Nom, Adj, Verbe, Det, Nom, Prep, Nom), on aura donc la liste d'arbres à un nœud :

Det — Adj - Nom — Adj — Verbe — Det - Nom - Prep — Nom

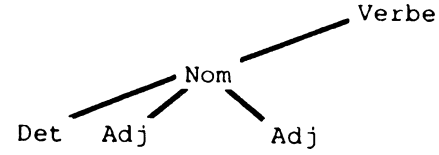
Deux nœuds contigus dans un schéma de règles doivent également être contigus dans les arbres objets, on pourra donc appliquer les règles Nom_Adj, Adj_Nom sur la séquence "beau bébé blond", la règle Prep_Nom sur "de légumes" et la règle Det_Nom sur "la bouillie". La règle Nom_Prep :

```
Nom_Prep [
    (1:{Nom}, 2:{Prep} (3)) => ( 1 ( 2 ( 3 ) ) ) ]
```

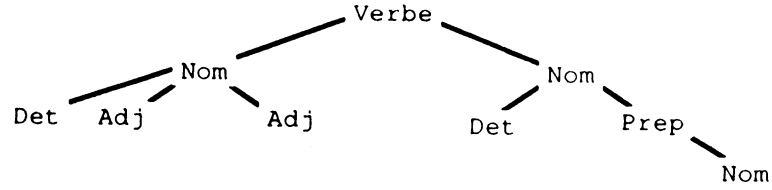
ne s'applique pas car elle précise que la préposition doit avoir un dépendant à droite (nœud 3). Un premier problème se pose sur la séquence "beau bébé blond" : dans quel ordre appliquer les règles ? en parallèle : il faut gérer le recouvrement, en séquence : dans quel ordre et quel résultat intermédiaire garder ? Nous supposons pour l'instant une application en parallèle dont le résultat est un seul arbre. On obtient donc :



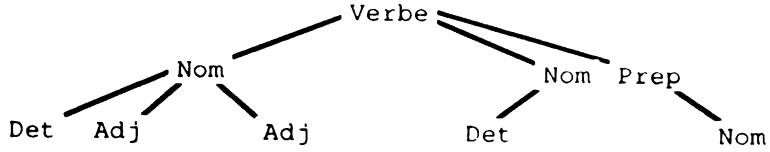
Avec cette liste, le problème se corse : on peut appliquer deux règles à gauche du verbe : Det_Nom et Nom_Verbe, mais si on applique d'abord la deuxième, on ne peut plus appliquer la première (le nom est "descendu" sous le verbe) ; on peut là encore effectuer une application en parallèle donnant :



On peut de même appliquer deux règles à droite du verbe : Nom_Prep et Verbe_Nom, mais cette fois l'application en parallèle donnerait :



et masquerait une autre solution :



qui n'est pas correcte dans ce cas mais qui pourrait l'être avec "le beau bébé blond mange la bouillie avec appétit".

Ces problèmes relèvent tous de la stratégie d'application des règles : dans quel ordre les appliquer, quels résultats intermédiaires conserver ? Si ces problèmes apparaissent avec une phrase et une grammaire aussi simples, ils deviennent énormes avec des phrases contenant des ambiguïtés lexicales (mots ayant plusieurs catégories, appelés homographes). Un exemple bien connu est "la belle ferme le voile" pour lequel notre analyseur morphologique fournit :

$$\begin{Bmatrix}\text{Det}\\\text{pron}\end{Bmatrix}\begin{Bmatrix}\text{subc}\\\text{adjq}\end{Bmatrix}\begin{Bmatrix}\text{subc}\\\text{adjq}\\\text{verb}\\\text{adv}\end{Bmatrix}\begin{Bmatrix}\text{Det}\\\text{pron}\end{Bmatrix}\begin{Bmatrix}\text{subc}\\\text{verb}\end{Bmatrix}$$

Sur cette séquence, six règles de la grammaire sont directement applicables.

Nous donnons ci-dessous la description des solutions que nous avons adoptées pour résoudre ces problèmes.

2.2. Transduction d'arbres

La construction d'un arbre de dépendances à partir d'une liste de CLS (éventuellement ambiguë) et d'une grammaire nécessite l'application des règles de la grammaire jusqu'à obtention d'un résultat. L'entrée et la sortie d'un module chargé de cette construction sont des listes d'arbres, l'algorithme général est donc celui d'un transducteur.

Chaque règle transforme une forêt en une nouvelle forêt à laquelle on peut appliquer une nouvelle règle,... L'application d'une grammaire (liste de règles) à une forêt F_0 peut être schématiquement décrite par l'algorithme suivant :

1 - initialiser i à 0
    et construire la liste L des règles de la grammaire applicables à F_0
2 - Tant que L n'est pas vide :
    3 - choisir une règle R parmi la liste L
    4 - transformer F_i en F_{i+1} en utilisant R
    5 - ajouter 1 à i,
        construire la liste L des règles de la grammaire applicables à F_i

La forêt obtenue est le résultat de la transduction.

La figure 2.2 donne l'architecture logicielle d'un système utilisant cet algorithme. Le module `Application` transforme une forêt en une autre forêt en lui appliquant une règle de la grammaire (voir §2.2.3). Cette règle est sélectionnée parmi les règles dont la partie gauche a satisfait au filtrage (voir §2.2.2) avec la forêt courante. Le système est initialisé avec une forêt et le résultat est la forêt obtenue quand aucune règle n'est applicable.

2.2.1. Choix fondamentaux

Deux problèmes se posent pour mettre en œuvre l'algorithme ci-dessus : déterminer des critères de choix à l'étape 3 et représenter les ambiguïtés inhérentes au langues naturelles.

2.2.1.1. Représentation de l'ambiguïté

Les langues naturelles étant de toutes façons ambiguës, il est nécessaire qu'un système d'analyse soit capable de représenter cette ambiguïté et ce à deux niveaux :

- lexical : il faut pouvoir prendre en compte les multiples interprétations d'un même mot (cf "la belle ferme le voile") ;

- structural : il faut pouvoir représenter les multiples interprétations d'une même phrase (cf "le bébé mange la bouillie de légumes" ou "avec appétit").

Cette représentation se traduit dans le schéma simple donné ci-dessus par le remplacement de la forêt F_i par une liste de forêts (F_{i1}, F_{i2}, ... F_{in}) où chaque F_{ij} est non ambiguë et représente une alternative, que ce soit au niveau lexical ou au niveau structural.

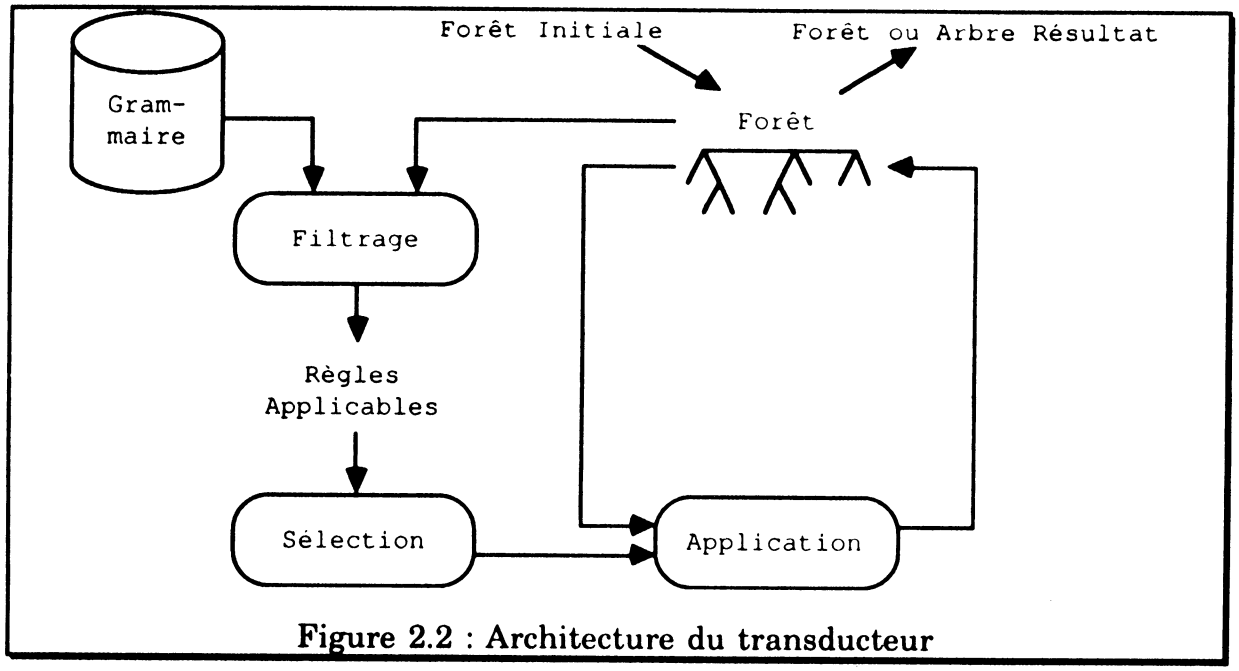


Figure 2.2 : Architecture du transducteur

2.2.1.2. Traitement de l'ambiguïté

Avec la représentation par liste de forêts, on peut imaginer un traitement en parallèle de chaque forêt de la liste, mais d'une part on retrouve les problèmes liés à ce type de technique que nous avons présenté au §1.1 :

- risque d'explosion combinatoire au niveau de l'encombrement mémoire : la représentation de "la belle ferme le voile" donne lieu à une liste de 64 forêts non ambiguës ;
- risque de calculs redondants : on peut être amenés à analyser plusieurs fois de manière identique un même morceau de la phrase.

D'autre part, cela ne règle pas le problème du choix de l'étape 3 de l'algorithme ci dessus.

Notre solution à ces problèmes passe par la définition d'une stratégie d'analyse simple et précise : procéder à une analyse gauche-droite progressive de la phrase plutôt que de la considérer directement dans son ensemble. On introduit les mots les uns après les autres dans le transducteur en commençant par la gauche et on n'introduit un nouveau mot que si l'analyse de la portion de phrase déjà présente est terminée. Le schéma d'algorithme d'analyse est donc le suivant (une version détaillée est donnée au §2.2.6) :

0 - Créer une forêt F à un arbre avec le premier mot de la phrase
1 - Répéter les étapes 2 et 3 jusqu'au dernier mot de la phrase
2 - Appliquer la grammaire à F (algorithme donné au début du §2.2)

3 - Ajouter le mot suivant à la fin de la forêt obtenue

La forêt résultat sera considérée comme une structure de dépendances correcte de la phrase si elle ne comporte qu'un seul arbre.

Exemple :

Avec la séquence `Det + Nom + Verbe + Det + Nom`, on a l'évolution suivante de l'analyse : (on ne détaille pas les règles appliquées)

Forêts	Reste à analyser
()	Det + Nom + Verbe + Det + Nom
(Det)	Nom + Verbe + Det + Nom
(Det, Nom)	Verbe + Det + Nom
((Det)Nom)	Verbe + Det + Nom
((Det)Nom, Verbe)	Det + Nom
(((Det)Nom)Verbe)	Det + Nom
(((Det)Nom)Verbe, Det)	Nom
(((Det)Nom)Verbe, Det, Nom)	
(((Det)Nom)Verbe, (Det)Nom)	
(((Det)Nom)Verbe((Det)Nom))	

Le premier intérêt de cette construction incrémentale et ascendante des arbres de dépendances est qu'elle est bien adaptée au langage des règles de réécriture d'arbres puisque ces règles ont en fait une portée limitée à quelques arbres et qu'elles bâtissent un arbre à partir de plusieurs.

Ensuite, ça permet de limiter l'espace de recherche pour l'application d'une règle puisque toute une partie de la phrase (donc toute une sous forêt de la forêt à traiter) est supposée déjà complètement analysée. Les forêts seront considérées comme des *piles* où seul le sommet est actif (on détaille cette technique au §2.2.2.2 et 2.2.5).

Enfin, le choix du parcours gauche-droite est motivé par le fait que l'analyseur morphologique dont nous disposons fonctionne de gauche à droite, on peut donc mettre en œuvre en parallèle les deux analyses et imaginer des transferts d'informations de la morphologie vers la syntaxe (mots reconnus) mais aussi de la syntaxe vers la morphologie (prédictions sur la catégorie du mot suivant).

Dans les paragraphes suivants, nous donnons une description détaillée des opérations de base du transducteur et de l'algorithme principal tels qu'ils ont été mis en œuvre, nous renvoyons au §2.3 quelques propositions pour l'amélioration des performances.

2.2.2. Filtrage

Une règle est applicable à une forêt si on peut mettre en correspondance le schéma de forêt de la partie gauche de la règle avec la forêt objet. Un schéma de forêt est en fait la description d'un ensemble de forêts et la mise en correspondance suppose la réponse à la question : est-ce que cette forêt appartient à l'ensemble des forêts décrit par ce schéma ? Si c'est le cas on dit que la forêt est une *instance* du schéma. La mise en correspondance est appelée *instanciation*.

Nous donnons d'abord quelques définitions générales avant la définition de l'instanciation, puis nous présentons l'algorithme utilisé par le transducteur.

2.2.2.1 Définitions

Soit C_0 un ensemble muni d'une relation d'ordre partiel notée «. On étend C_0 en ajoutant deux éléments spéciaux, notés _TOUT_ et _RIEN_ tels que, si C = $C_0 \cup$ {_TOUT_,_RIEN_}, on ait les propriétés :

$\forall x \in C$, x « _TOUT_ et $\forall x \in C$, _RIEN_ « x

On note $\mathcal{P}(C)$ l'ensemble des parties de C.

Minorants :

On appelle ensemble des *minorants* d'un élément x de C et on note $\mathcal{M}in(x)$ le sous-ensemble de C défini par :

$\mathcal{M}in(x) = \{ y \in C / y « x \}$

Remarque : $\mathcal{M}in(x)$ n'est jamais vide puisqu'il contient au moins _RIEN_.

Maximaux :

On appelle ensemble des *maximaux* d'un élément A de $\mathcal{P}(C)$ et on note Max(A) le sous ensemble de A défini par :

$Max(A) = \{ x \in A / \forall y \in A, x « y \Rightarrow x = y \}$

Remarques : si _TOUT_ $\in$ A, Max(A) = {_TOUT_}

si $A = B \cup$ {_RIEN_}, $B \neq \emptyset$ alors Max(A) = Max(B)

Unification sur C :

On définit l'*unification simple* de deux éléments x et y de C comme une fonction notée $\amalg_C$:

$\amalg_C : C \times C \rightarrow \mathcal{P}(C)$

$(x, y) \rightarrow Max(\mathcal{M}in(x) \cap \mathcal{M}in(y))$

Remarques : $\forall x \in C, \amalg_C(\text{\_TOUT\_}, x) = \amalg_C(x, \text{\_TOUT\_}) = \{x\}$

$\forall x \in C, \amalg_C(\text{\_RIEN\_}, x) = \amalg_C(x, \text{\_RIEN\_}) = \{\text{\_RIEN\_}\}$

Unification sur $\mathcal{P}(C)$:

On peut étendre $\amalg_C$ à $\mathcal{P}(C)$ et obtenir une loi de composition interne dans $\mathcal{P}(C)$, appelée *unification* et notée $\amalg$ de la façon suivante :

$\amalg : \mathcal{P}(C) \times \mathcal{P}(C) \rightarrow \mathcal{P}(C)$

$(\{x_i\}_{1 \leq i \leq n}, \{y_j\}_{1 \leq j \leq m}) \rightarrow Max(\bigcup_{i=1}^{n}(\bigcup_{j=1}^{m}(\amalg_C(x_i, y_j))))$

Remarques : Cette loi est commutative et possède un élément neutre, en effet :

$\forall A \in \mathcal{P}(C), \amalg(\{\text{\_TOUT\_}\}, A) = A$

De plus {_RIEN_} est un élément absorbant :

$\forall A \in C, \amalg(\{\text{\_RIEN\_}\}, A) = \{\text{\_RIEN\_}\}$

Exemples :

En prenant comme ensemble partiellement ordonné C_0 l'ensemble des symboles de la figure 2.1 avec la relation « définie par "est relié par un arc montant avec" on a :

```
xet « xet, xet « Verbe, subc « Nom, ...
⊔({Verbe}, {xav, xet}) = {xav, xet}
⊔({Verbe}, {Adj}) = {ppas}
```

Forêts et arbres de dépendances :
On appelle A_N l'ensemble des *arbres de dépendances* sur un ensemble N fini de *nœuds*. Un élément x de A_N est décrit par :

$(g_1, g_2, \ldots, g_n)\ r\ (d_1, d_2, \ldots, d_m)$

où $r \in N, n, m \geq 0, g_i, d_j \in A_{N-\{r\}}$

Une *forêt de dépendances* est une liste éventuellement vide d'arbres de dépendances. La forêt $(g_1, g_2, \ldots, g_n)$ est dite *forêt gauche* ou liste des dépendants gauches de r. La forêt $(d_1, d_2, \ldots, d_m)$ est dite *forêt droite* ou liste des dépendants droits de r. La forêt vide est notée ().
On suppose qu'il existe toujours une bijection de N dans un intervalle de $\mathbb{N}^+$, on désigne donc les nœuds avec des entiers positifs.

Forêts et arbres de dépendances étiquetés :
Un *arbre de dépendances étiqueté sur E* est un couple (a, f_E) ou $a \in A_N$ et f_E est une fonction de N dans un ensemble E d'*étiquettes*. Une *forêt de dépendances étiquetée* est une liste éventuellement vide d'arbres de dépendances étiquetés.
On utilise $\mathcal{P}(C)$ pour étiqueter les arbres et les forêts.
Exemples :
En prenant pour C l'ensemble des catégories définies par la figure 2.1, augmenté des symboles _TOUT_ et _RIEN_, l'arbre associé au groupe nominal "le beau petit bébé blond" s'écrira :
(1, 2, 3) 4 (5) avec $f_{\mathcal{P}(C)}(1)$ = {Det}, $f_{\mathcal{P}(C)}(2)$ = {adjq}, $f_{\mathcal{P}(C)}(3)$ = {adjq}, $f_{\mathcal{P}(C)}(4)$ = {subc}, $f_{\mathcal{P}(C)}(5)$ = {adjq}.

Schémas d'arbres et de forêts :
On appelle S_{NV} l'ensemble des *schémas d'arbres de dépendances* sur un ensemble fini N de *nœuds* et sur un ensemble V de symboles. Un élément x de S_{NV} est soit :

- $s où $s \in V$
- $(g_1, g_2, \ldots, g_n)\ r\ (d_1, d_2, \ldots, d_m)$ où $n, m \geq 0, g_i, d_j \in S_{NV}, r \in N$

Un *schéma de forêt de dépendances* est une liste éventuellement vide de schémas d'arbres de dépendances.
Les références $s sont appelées *variables de forêt*.
Comme pour les arbres de dépendances, on désigne les nœuds avec des entiers positifs.

Schémas d'arbres et de forêts contraints :
Un *schéma d'arbre de dépendances contraint sur E* est un couple (s, c_E) où $a \in S_N$ et c_E est une fonction de $N \cup V$ dans un ensemble E d'*étiquettes*. Un *schéma de forêt de dépendances contraint* est une liste éventuellement vide de schémas d'arbres de dépendances contraints. On utilise $E = \mathcal{P}(C)$.
Exemples :
Le schéma de forêt :
(1:{Verbe}, 2:{Prep} (3))
donné dans la règle Verbe_Prep est noté :
($A, ($B)1($C), ($D) 2 (($E)3($F))) avec :

$c_{\mathcal{P}(C)}(1)$ = {Verbe}, $c_{\mathcal{P}(C)}(2)$ = {Prep}, $c_{\mathcal{P}(C)}(3)$ = {_TOUT_}, $c_{\mathcal{P}(C)}(x)$ = {_TOUT_} pour tout x dans V = {A, B, C, D, E, F}. Comme on va le voir ci-dessous, on rend explicites les éléments laissés implicites par la syntaxe des règles.

2.2.2.2. Instanciation

Elle est à la base de l'application d'une règle de la grammaire à une forêt. Dans notre système, une forêt est considérée comme une pile d'arbres dont le sommet est l'arbre le plus à droite. Une instance d'un schéma de forêt n'est cherchée que sur les arbres situés au sommet de la pile. Pour établir une correspondance exacte entre un schéma de forêt et une forêt donnée, on ajoute dans le schéma les variables de forêts laissées implicites dans la description du schéma dans la règle. Les principes sont les suivants :

- toute liste de schémas d'arbres ne précisant pas un 0 à gauche se voit ajouter une variable de forêt à gauche ; ainsi le schéma :
  `(1, 2(3))` est équivalent à `($A, 1, 2($B, 3))` alors que :
  `(0, 1, 2(3))` est équivalent à `(1, 2($A, 3))`.
- tout nœud dont une liste de dépendants n'est pas précisée se voit ajouter une variable de forêt comme liste de ces dépendants ; ainsi :
  `($A, 1, 2($B, 3))` est équivalent à
  `($A, ($C)1($D), ($E)2($B, ($F)3($G)))`
  alors que :
  `(0, ()1, ()2(3))` est équivalent à `(1($A), 2($B, ($C)3($D)))`.

Avec ces équivalences on a les définitions récursives suivantes :

Un arbre de dépendances étiqueté a = (ga_1, ga_2, ..., ga_n) ra (da_1, da_2, ..., da_m), n, m ≥ 0 est une instance d'un schéma d'arbre avec contraintes s = (gs_1, gs_2, ..., gs_p) rs (ds_1, ds_2, ..., ds_q), p, q ≥ 0 si et seulement si :

1- (ga_1, ga_2, ..., ga_n) et (da_1, da_2, ..., da_m) sont respectivement des instances de (gs_1, gs_2, ..., gs_p) et (ds_1, ds_2, ..., ds_q)

2- la condition d'occurrence sur rs est vérifiée c'est-à-dire si l'étiquette de la racine de l'arbre s'unifie avec la contrainte de la racine du schéma, ce qui s'écrit :
$\sqcup(f_{\mathcal{P}(C)}(ra), c_{\mathcal{P}(C)}(rs)) \neq$ {_RIEN_}

Une forêt de dépendances étiquetée f_a = (a_1, a_2, ..., a_n), n ≥ 0, est une instance d'un schéma de forêt de dépendances avec contraintes f_s si et seulement si :

1- f_s = $x, x ∈ V, et ∀i, 1 ≤ i ≤ n, si ra_i désigne la racine de l'arbre a_i, on a : $\sqcup(f_{\mathcal{P}(C)}(ra_i), c_{\mathcal{P}(C)}(x)) \neq$ {_RIEN_}

2- f_s = (s_1, s_2, ..., s_m), m ≥ 0, et soit :

- m = n = 0
- m, n ≠ 0, $s_m \neq$ $x, x ∈ V, a_n est une instance de s_m et (a_1, a_2, ..., a_{n-1}) est une instance de (s_1, s_2, ..., s_{m-1})
- m ≠ 0, s_m = $x, x ∈ V, et ∃j, 1 ≤ j ≤ n tel que : (a_1, ..., a_j) est une instance de (s_1, s_2, ..., s_{m-1}) et (a_{j+1}, ..., a_n) est une instance de s_m.

2.2.2.3. Algorithme d'instanciation

Les définitions ci-dessus permettent de répondre à la question : cette forêt appartient-elle à l'ensemble des forêts décrit par ce schéma ? Mais pour construire l'arbre résultat, on doit pouvoir attribuer à chaque élément (nœud ou variable de forêt) du schéma de forêt f_s un élément indépendant de la forêt f_a. Autrement dit on doit pouvoir construire une fonction, dite *fonction de correspondance*, qui à chaque nœud de f_s associe un nœud de f_a et à chaque variable de forêt de f_s associe une sous-forêt de f_a. Cette fonction doit être telle que les ensembles de nœuds images dans N soient disjoints deux à deux. Cette contrainte est indispensable à la bonne construction de l'arbre résultat car elle assure que les différents arbres à assembler sont disjoints.

Dans la dernière définition ci-dessus, il peut exister plusieurs indices j vérifiant les contraintes imposées, ce qui veut dire qu'une même forêt pourra simultanément se voir attribuer plusieurs fonctions de correspondance avec un schéma de forêt.

L'algorithme d'instanciation donné ci-dessous reprend la définition mais ajoute la construction des fonctions de correspondance. Chaque fonction est représentée par une liste de couples (élément de f_s, élément de f_a).

Prédicat InstancierListe(fa, fs, res) :
Entrées :
fa : forêt de dépendances étiquetée, notée $(a_1, a_2, ..., a_n)$, $n \geq 0$
fs : schéma de forêt contraint, noté $(s_1, s_2, ..., s_m)$, $m \geq 0$
Sorties :
res : une liste de listes de couples (élément de fs, élément de fa) représentant une liste de fonctions de correspondance entre fs et fa.
une valeur booléenne vrai si fa est une instance de fs et faux sinon (cette valeur est la valeur du prédicat).

Si m = n = 0 alors InstancierListe := vrai sinon
  Si m et n $\neq$ 0 et $s_m \neq \$x$, $x \in V$ alors
    Si InstancierEléments(a_n, s_m, f_un)
    et InstancierListe($(a_1, a_2, ..., a_{n-1})$, $(s_1, s_2, ..., s_{m-1})$, f_liste) alors
      InstancierListe := vrai
      res := combinaison de f_un et f_liste (voir ci-dessous)
    sinon InstancierListe := faux
  sinon
    Si m $\neq$ 0, $s_m = \$x$, $x \in V$ alors
      res := ()
      pour tout j, $1 \leq j \leq n$ faire
        Si InstancierListe($(a_1, ..., a_j)$, $(s_1, s_2, ..., s_{m-1})$, f_liste)
        et $\forall i, j+1 \leq i \leq n$, $\amalg(f_{\mathcal{R}(C)}(\text{racine}(a_i)), c_{\mathcal{R}(C)}(x)) \neq$ {_RIEN_}
        alors
          Combiner(f_liste, $(((x, (a_{j+1}, ..., a_n))))$, F)
          res := concaténation de res et F
      InstancierListe := (res $\neq$ ())
    sinon InstancierListe := faux

Prédicat InstancierEléments(a, s, res) :
Entrées :
a : arbre de dépendances étiquetée, noté
(ga_1, ga_2, ..., ga_n) ra (da_1, da_2, ..., da_m), n, m ≥ 0
s : schéma d'arbre contraint, noté
(gs_1, gs_2, ..., gs_p) rs (ds_1, ds_2, ..., ds_q), p, q ≥ 0
Sorties :
res : une liste de listes de couples (élément de s, élément de a) représentant une liste de fonctions de correspondance entre a et s.
une valeur booléenne vrai si a est une instance de s et faux sinon (cette valeur est la valeur du prédicat).
Si InstancierListe((ga_1, ga_2, ..., ga_n), (gs_1, gs_2, ..., gs_p), gauche)
et InstancierListe((da_1, da_2, ..., da_m), (ds_1, ds_2, ..., ds_q), droit)
et ⊔($f_{\mathcal{R}(C)}$(ra), $c_{\mathcal{R}(C)}$(rs)) ≠ {_RIEN_} alors
Combiner(gauche, droit, res)
Combiner(res, (((rs, ra))), res)
InstancierEléments := vrai
sinon InstancierEléments := faux

Algorithme Combiner(l1, l2, res) :
Entrées :
l1, l2 : listes de listes de couples (élément de schéma, élément d'arbre)
Sorties :
res : liste de listes de couples (élément de schéma, élément d'arbre)
res := ()
Pour chaque liste de couples x de l1
Pour chaque liste de couples y de l2
ajouter la concaténation de x et y à res

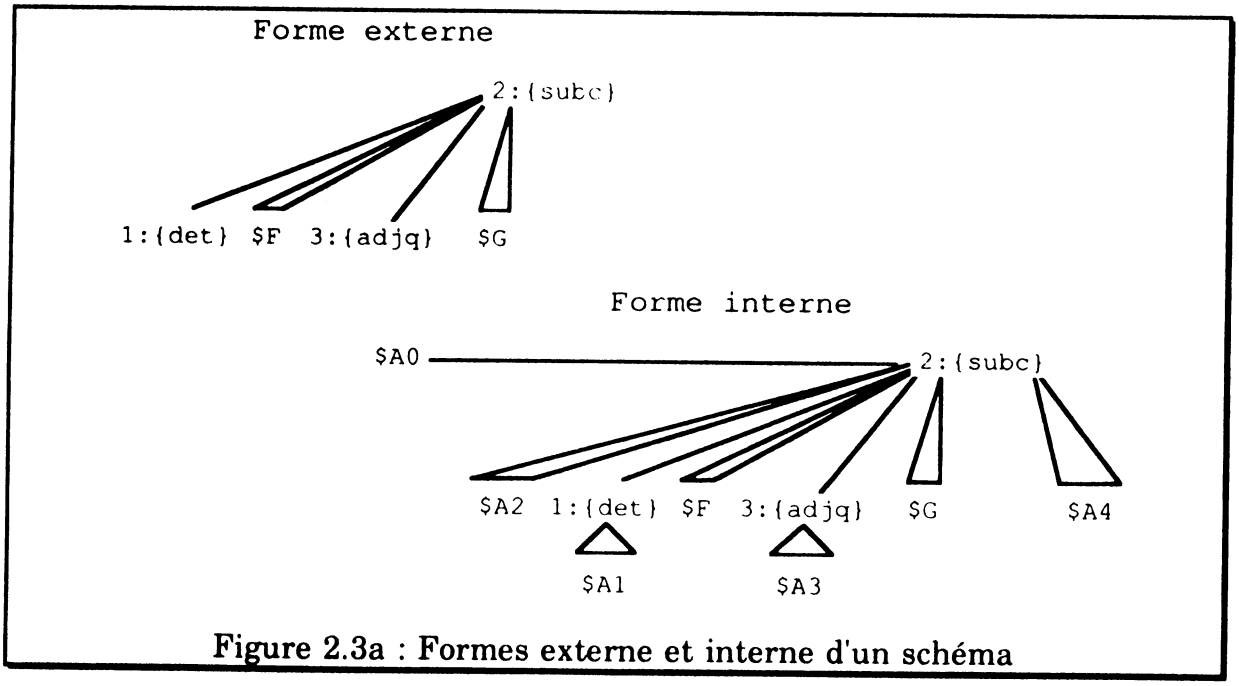


Figure 2.3a : Formes externe et interne d'un schéma

On trouvera sur la figure 2.3a un exemple de schéma sous forme externe (telle qu'elle est notée dans les règles) et la forme interne correspondante (utilisée pour l'instanciation). La figure 2.3b donne un arbre et les différentes fonctions de correspondance entre le schéma de la figure 2.3a et cet arbre.

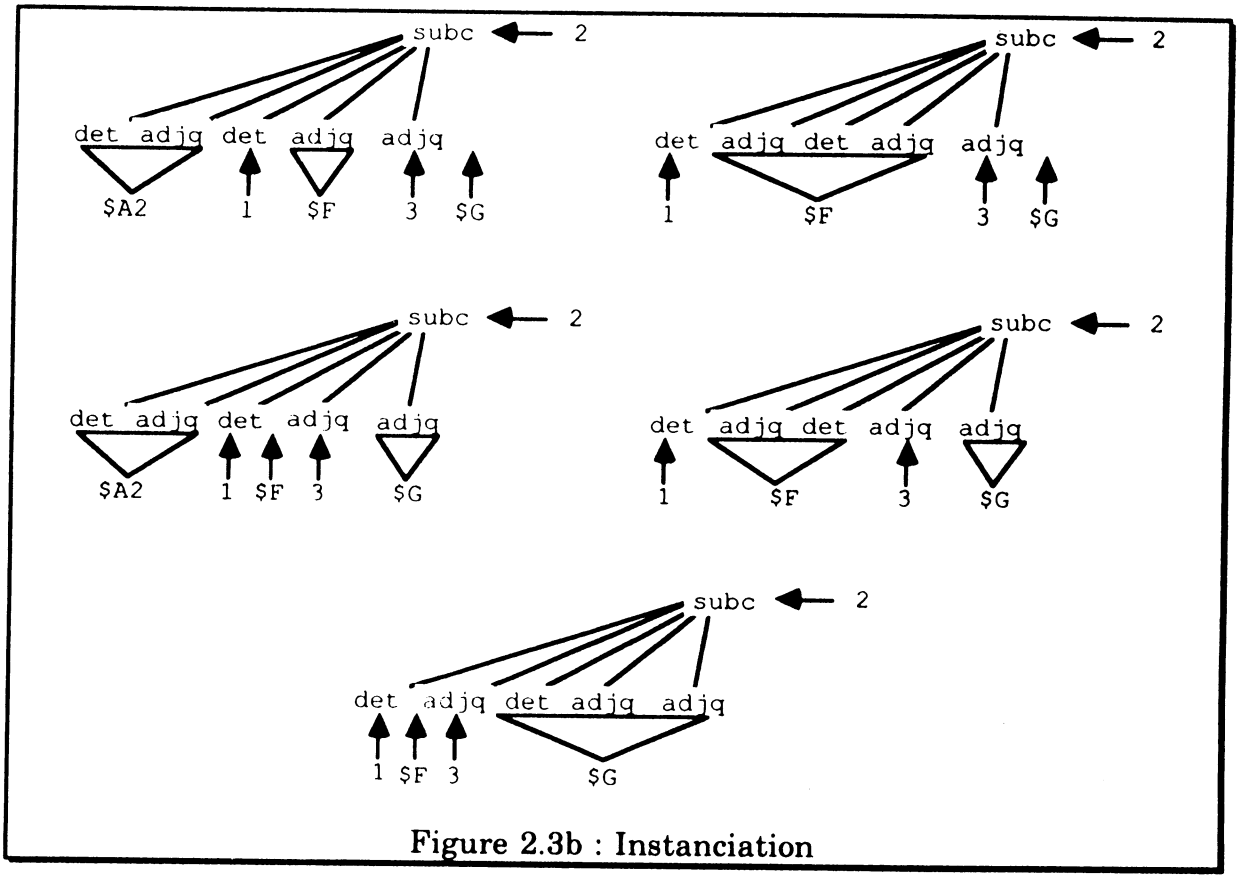


Figure 2.3b : Instanciation

2.2.3. Construction de l'arbre résultat (application)

La partie droite d'une règle de réécriture est constituée d'un schéma de forêt de dépendances non contraint. Ce schéma décrit la géométrie d'une forêt dont les éléments (nœuds et sous-forêts) sont les instances des éléments correspondants du schéma de la partie gauche de la règle.

La construction de cette forêt s'appuie sur la fonction de correspondance calculée par l'instanciation : si on considère cette fonction comme un découpage de la forêt objet, la construction de la forêt résultat est un simple réassemblage des éléments obtenus. Il est clair que tout nœud ou toute forêt apparaissant dans le schéma de forêt résultat doit également apparaître dans le schéma de la partie gauche. Par contre, il n'est pas nécessaire que tous les éléments de la partie gauche apparaissent en partie droite : les éléments absents seront simplement effacés dans la forêt résultat. Cette fonctionnalité permet entre autre d'écrire des règles qualifiées de *règles de désambiguïsation* dans lesquelles le schéma de la partie droite est vide. Leur rôle est d'éliminer des configurations de mots qui n'ont aucun sens, rendant ainsi l'analyse plus efficace.

Exemple :

On pourra avoir la règle :

```
Det_Verb [($D:{Det ; adjq}, 1:{verb}) => () ]
```
qui permet d'éliminer 22 des 64 combinaisons de catégories non ambiguës associées à la phrase "la belle ferme le voile".

Le seul problème dans cette construction est le rattachement des forêts implicites, c'est-à-dire de toutes les forêts qui apparaissent dans la forme interne mais pas dans la forme externe des schémas d'instanciation. La fonction chargée de ces rattachements est classiquement appelée *fonction de transfert* car c'est elle qui a la charge du transfert des éléments implicites de la forêt objet vers la forêt résultat.

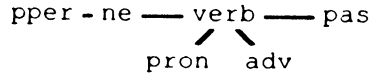
Nous n'avons pas (contrairement à ROBRA) rendu cette fonction explicite dans les règles, nous avons simplement défini son fonctionnement à l'aide des principes suivants (qualifiés de *principes du moindre effort*) :

- toute forêt implicite reste attachée à son père et le suit dans la construction ;
- les éléments explicites ajoutés comme frère d'une forêt implicite sont toujours ajoutés à droite de cette forêt si c'est une sous-forêt droite et à gauche si c'est une sous forêt gauche. Ce principe permet dans la majorité des cas rencontrés pour l'analyse du français de conserver la projectivité des structures construites.

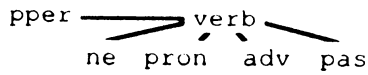
Exemple :

La règle :

```
Ne_Verbe_Pas [ (1:{ne}, 2:{Verbe}, 3:{pas}) => ((1) 2 (3)) ]
```
appliquée à la forêt (qui peut correspondre à "je ne le vois généralement pas ") :



donne trois forêts implicites non vides : `({pper}) ({pron}) et ({adv})` et construit la forêt :



L'application d'une règle à une forêt suppose la vérification des contraintes d'instanciation du schéma. La vérification de ces contraintes apporte une information sur les étiquettes des nœuds mis en jeu. Pour ne pas perdre cette information, on remplace les étiquettes des nœuds par le résultat de l'unification de l'ancienne valeur avec la contrainte correspondante. Ce mécanisme n'apporte pas grand chose par rapport à une simple recopie des étiquettes puisque dans la plupart des cas l'étiquette du nœud est une simple spécialisation de la contrainte portée par la règle. Mais dans le cas de mots inconnus ou dont la catégorie n'est pas complètement déterminée, il permet de la spécialiser et donc d'ajouter de l'information. Nous reviendrons sur ce mécanisme dans le chapitre 6.

La construction d'une forêt résultat sur la base d'une fonction de correspondance peut être décrite par l'algorithme suivant :

Algorithme ConstruireRésultat(fc, sc, ao, s, res) :
Entrées :
fc : fonction de correspondance calculée par l'instanciation : liste de couples (élément de sc, élément de ao)
sc : schéma de forêt contraint dont ao est une instance
ao : forêt de dépendances étiquetée
s : schéma de forêt non contraint, noté $(s_1, s_2, ..., s_p)$ $p \geq 0$
Sorties :
res : forêt de dépendances étiquetée, instance du schéma s.
res := ()
pour chaque i, $1 \leq i \leq p$ faire
Si $s_i \in V$ alors
soit $fc(s_i)$ la forêt de a associée à s_i dans fc
soit $c_{\mathcal{R}(C)}(s_i)$ la contrainte associée à s_i dans sc
pour chaque racine r des arbres de $fc(s_i)$ faire
$f_{\mathcal{R}(C)}(r) := \amalg(f_{\mathcal{R}(C)}(r), c_{\mathcal{R}(C)}(s_i))$
concaténer les forêts res et $fc(s_i)$
sinon
ConstruireArbre(fc, a, s_i, unarbre)
concaténer les listes res et (unarbre)

Algorithme ConstruireArbre(fc, sc, ao, s, un) :
Entrées :
fc : fonction de correspondance calculée par l'instanciation : liste de couples (élément de sc, élément de ao)
sc : schéma de forêt contraint dont ao est une instance
ao : forêt de dépendances étiquetée.
s : schéma d'arbre non contraint, noté
$(gs_1, gs_2, ..., gs_p)$ rs $(ds_1, ds_2, ..., ds_q)$, p, $q \geq 0$
Sorties :
un : arbre de dépendances étiqueté, instance du schéma s.
Soit fc(rs) le nœud de a associé à rs dans fc
et $c_{\mathcal{R}(C)}(rs)$ la contrainte associée à rs dans sc
$f_{\mathcal{R}(C)}(fc(rs)) := \amalg(f_{\mathcal{R}(C)}(fc(rs)), c_{\mathcal{R}(C)}(rs))$
un := rs avec les deux forêts implicites $G et $D qui lui sont attachées, on a
un = ($G)rs($D)
Si $p \neq 0$ alors
ConstruireRésultat(fc, sc, ao, $(gs_1, gs_2, ..., gs_p)$, gauche)
concaténer gauche et $G, on a un = (gauche, $G)rs($D)
Si $q \neq 0$ alors
ConstruireRésultat(fc, sc, ao, $(ds_1, ds_2, ..., ds_q)$, droite)
concaténer $D et droite, on a un = (gauche, $G)rs($D, droite)

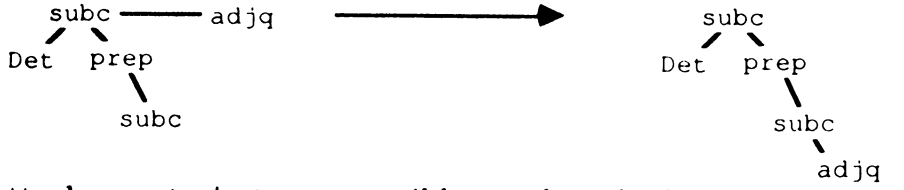
2.2.4. Les limites basses et hautes

Les algorithmes présentés ci-dessus pour l'instanciation et la construction de la forêt résultat ne permettent pas de résoudre tous les problèmes exposés au §2.1.4 : le problème des rattachements des groupes prépositionnels subsiste. Ce problème est en fait lié à la stratégie gauche-droite que nous avons choisie : au moment où le groupe prépositionnel est reconnu, tout ce qui se trouve à gauche

est déjà analysé et donc le point de rattachement peut être autre que la racine de l'arbre qui précède le groupe. On retrouve le même problème partout où un mot est susceptible d'être un dépendant droit d'un autre.

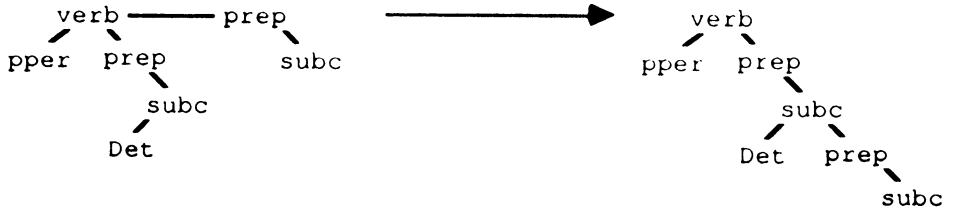
Exemples :

"l'avion de couleur jaune", après analyse de "couleur",
on aura : et on voudrait qu'une règle permette de construire :

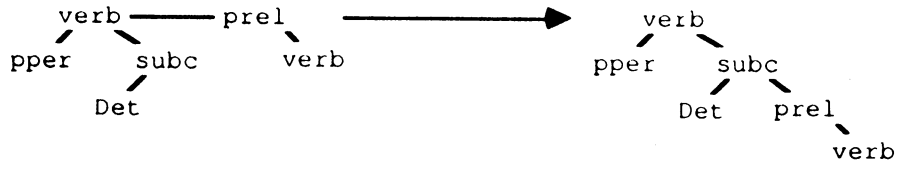


Mais ce rattachement n'est pas possible car le subc "couleur" n'apparaît pas à la racine de l'arbre.

"elle travaille à l'école de commerce", après avoir rattaché "commerce" à "de", on aura : et on voudrait obtenir :



"j'entends les lions qui rugissent", après avoir rattaché "rugissent" à "qui", on aura : et on voudrait obtenir :



Pour régler ce problème, nous avons ajouté à l'instanciation la possibilité de sélectionner un sous-arbre d'un arbre avec pour contrainte que la racine du sous-arbre soit sur le chemin le plus à droite de l'arbre objet. Cette possibilité est notée sur la forme externe des schémas par une *limite*, introduite par un ? ou un / après la racine d'un arbre suivi d'un ensemble L de catégories (élément de $\mathcal{P}(C)$), dont l'interprétation est la suivante : une instance de ce schéma peut être cherchée sur le chemin le plus à droite de l'arbre objet en partant du haut (/) ou du bas (?) mais sans dépasser un nœud dont l'étiquette s'unifie avec L. Un tel nœud pourra toutefois être la racine de l'instance.

Exemple :

Les exemples ci dessus peuvent être traités avec les règles :

```
Nom_Adj [ (1:{Nom}?{prel},  2:{Adjq} ) => (1(2)) ]
Nom_prel [ (1:{Nom}?{verbe},  2:{prel}(3)) => (1(2(3))) ]
Nom_prep [ (1:{Nom}?{prel},  2:{prep}(3)) => (1(2(3))) ]
```

La limite permet d'interdire des rattachements trop lointains comme dans "l'avion qui sort du hangar de couleur jaune" ("jaune" ne peut pas s'attacher à "avion" et "de couleur jaune" non plus). Elle empêche donc la construction de multiples structures incohérentes. En effet, l'introduction de ces limites multiplie les instances possibles d'un même schéma dans un arbre objet : pour

rattacher  à l'arbre de la figure 2.4a, on aura 4 possibilités avec la règle Nom_prep ci-dessus mais seulement une pour rattacher  à l'arbre de la figure 2.4b.

Figure 2.4a

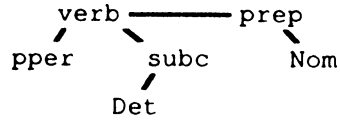
Ce mécanisme introduit un autre type de structure implicite : le *chapeau implicite*. En effet, l'instance du schéma d'arbre avec limite peut être un sous-arbre d'un arbre objet, et la racine de cet arbre peut ne pas apparaître dans la fonction de correspondance.

Figure 2.4b

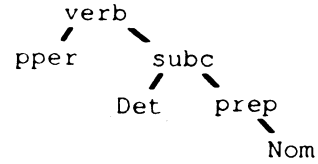
Nous avons donc imposé qu'un seul arbre avec limite apparaisse dans le schéma de forêt d'une règle et nous avons ajouté à la fonction de transfert le principe suivant : la racine de l'instance d'un schéma avec limite reste attachée à son chapeau implicite ; si cette racine est attachée à un autre élément du schéma comme sous-arbre alors cet élément est attaché au chapeau de la racine.

Exemple :

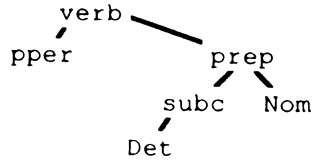
La règle Nom_prep appliquée à la forêt :



donne :



Si la règle précisait comme schéma de forêt résultat : `((1)2(3))`, on obtiendrait :



2.2.5. Piles d'arbres

Nous avons vu dans les paragraphes précédents les mécanismes d'application d'une règle à une forêt. Nous allons détailler ici la structure de données et les algorithmes mis en œuvre pour gérer les ambiguïtés.

Les ambiguïtés peuvent prendre deux formes :

- un mot a de multiples interprétations (ambiguïté lexicale ou homographie) ;
- une forêt sur laquelle plusieurs règles s'appliquent ou la même règle s'applique plusieurs fois.

Les règles ne pouvant manipuler directement que des forêts non ambiguës, il faut dans les deux cas pouvoir énumérer toutes les alternatives. Nous représentons ces alternatives dans une simple **liste** de forêts non ambiguës où, comme on l'a vu ci-dessus, chaque forêt est considérée comme une pile d'arbres dont le sommet est à la fin.

L'application d'une règle à une forêt conduit au remplacement de cette forêt par la forêt calculée par la règle. Si plusieurs règles s'appliquent, alors la forêt est remplacée par la liste des forêts résultant de l'application des règles. Nous allons illustrer l'évolution de cette liste avec un exemple simple.

Exemple :

On ne considère que 4 catégories : D, A, N, V (pour déterminant, adjectif, nom et verbe) et on se donne les règles simples :

```
D_N [(1:{D}, 2:{N}) => ((1) 2) ]
A_N [(1:{A}, 2:{N}) => ((1) 2) ]
N_A [(1:{N}, 2:{A}) => (1 (2)) ]
N_V [(1:{N}, 2:{V}) => ((1) 2) ]
V_N [(1:{V}, 2:{N}) => (1 (2)) ]
```

La figure 2.5 illustre l'évolution de la liste pour l'analyse du groupe nominal : "la belle ferme" qui correspond à la liste de catégories :

$$D \begin{Bmatrix} A \\ N \end{Bmatrix} \begin{Bmatrix} A \\ N \\ V \end{Bmatrix}$$

On introduit d'abord le mot "la" sous forme d'un arbre à un nœud étiqueté par D. Comme aucune règle ne s'applique à cet arbre, on introduit le mot

"belle"dont l'ambiguïté produit deux forêts qui sont décrites par la liste (1). L'application de la règle D_N à la deuxième forêt produit la liste (2).

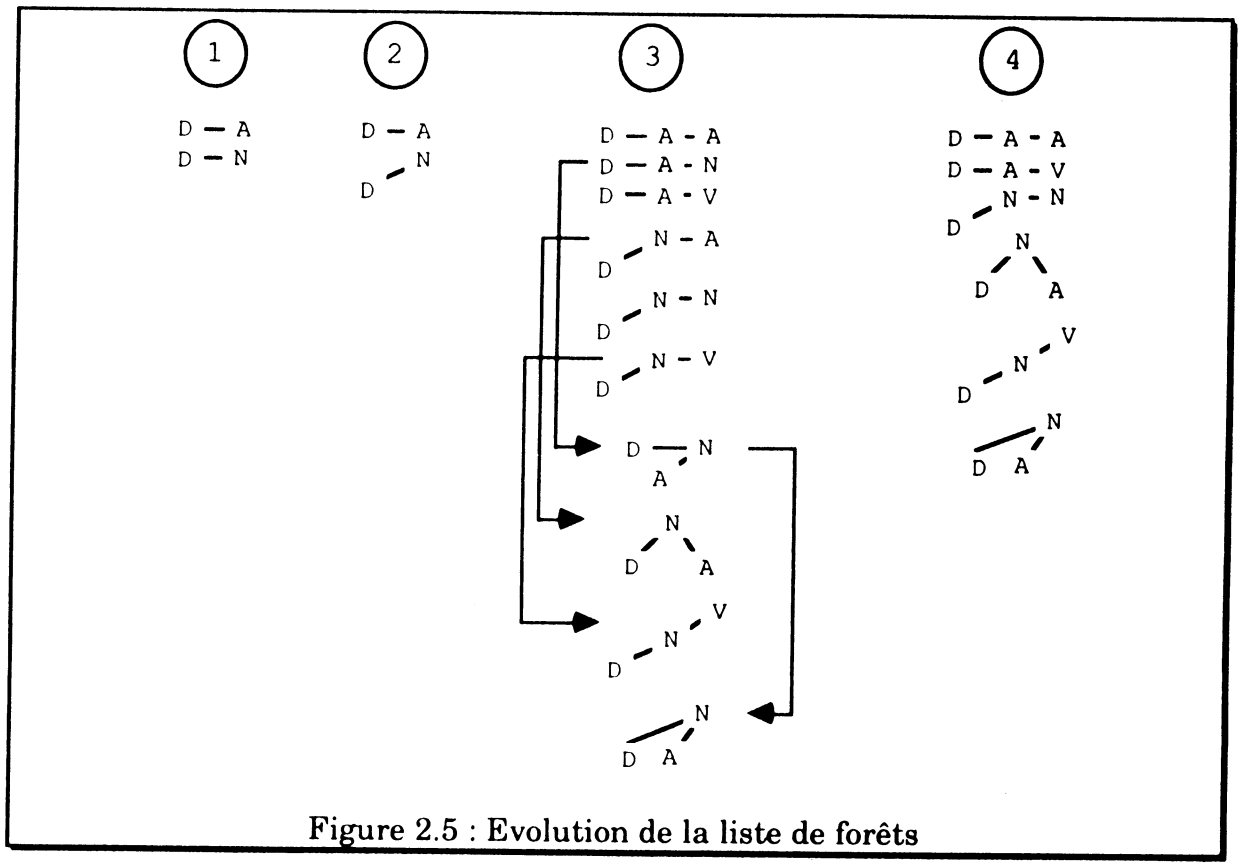


Figure 2.5 : Evolution de la liste de forêts

Sur la liste (3), obtenue après introduction du mot "ferme", on détaille les applications des règles. Ainsi l'application de la règle A_N à la deuxième forêt de cette liste produit une nouvelle forêt qui est ajoutée en fin de liste. L'intérêt de cet ajout est de pouvoir continuer à appliquer les règles de la grammaire aux forêts produites. Ainsi la règle D_N sera appliquée à cette forêt. On arrête l'application quand on atteint la fin de la liste. On a alors la liste (4).

Pour une phrase donnée, les seules forêts correspondant à des interprétations correctes sont les forêts qui ne contiennent qu'un seul arbre. Pour l'exemple on aura trois structures correctes :
`(la)belle (ferme)`, `((la)belle)ferme` et `(la, belle)ferme`

2.2.6. Algorithme

Le moteur du transducteur peut être décrit par l'algorithme suivant :

Algorithme Analyse(Enoncé, Règles, Liste)
Entrées :
    Enoncé : la chaîne à analyser
    Règles : la liste des règles de la grammaire
Sorties :
    Liste : la liste des forêts produites
Liste := ()

```
Tant que NouveauMot(Enoncé, Liste) faire
    Forêt := première forêt de Liste
    Tant que Forêt existe faire
        ApplicationRègles(Règles, Forêt, UneListe)
        Si UneListe ≠ () alors
            ajouter UneListe en fin de Liste
            Supprimer Forêt de Liste
        Forêt := forêt suivante de Liste
```

Algorithme ApplicationRègles(Règles, Forêt, Liste)

Entrées :
    Règles : la liste des règles de la grammaire
    Forêt : la forêt objet à laquelle les règles vont s'appliquer
Sorties :
    Liste : la liste des forêts résultant de l'application des règles à Forêt

```
Liste := ()
pour chaque règle R de Règles faire
    Calculer la liste L des forêts résultant de l'application de R à Forêt
    ajouter L en fin de Liste
```

Prédicat NouveauMot(Enoncé, Liste)

Entrées :
    Enoncé : la chaîne à analyser
    Liste : une liste de forêts
Sorties :
    Liste : la liste des forêts à laquelle le prochain mot de l'énoncé a été ajouté
    un booléen qui est la valeur du prédicat, et qui indique si la fin de l'énoncé est atteinte

```
obtenir le prochain mot M de l'énoncé (appel de l'analyseur morphologique)
Si la fin de l'énoncé est atteinte alors NouveauMot := faux
Sinon
    NouveauMot := vrai
    Copie := copie de Liste
    Liste := ()
    pour chaque interprétation possible I de M
        créer un arbre à un nœud A étiqueté par I
        Pour chaque forêt f de Copie
            empiler a sur f et ajouter le résultat en fin de Liste
```

Remarques :

L'arrêt de cet algorithme n'est pas garanti : rien n'empêche en effet l'écrivain de grammaire de créer une règle identité qui recopie simplement la forêt objet sur une forêt résultat. Une solution possible est d'imposer dans la grammaire que le nombre d'arbres d'un schéma de forêt résultat soit toujours strictement inférieur au nombre d'arbres du schéma d'instance de la forêt objet. Une telle contrainte garantit l'arrêt puisque le nombre d'arbres des forêts ne peut que diminuer.

On peut faire une évaluation de cet algorithme en fonction du nombre de mots m de l'énoncé et du nombre de règles r de la grammaire. L'application des règles à une forêt non ambiguë est en O(r) puisque toutes les règles sont

essayées. L'évaluation de cet algorithme consiste donc à déterminer le nombre d'utilisations de ApplicationRègles. Ce nombre est lié à m par la formule :

$AR(m, r) = \sum_{i=1}^{m} ar_i$ où ar_i est le nombre d'appels à ApplicationRègles après introduction du $i^{ème}$ mot.

Le nombre d'interprétations différentes d'un mot est borné par une constante h (dans la version actuelle du dictionnaire h = 4). Pour l'évaluation, on considère que h est le nombre moyen d'interprétations différentes d'un mot. On a un appel par forêt de la liste, on peut donc écrire :

$ar_i = f_{i-1} * h + a_i$, où f_{i-1} est le nombre de forêts de la liste après l'étape i-1 et a_i est le nombre de forêt ajoutées en fin de liste par application des règles aux $f_{i-1} * h$ forêts de base.

L'évaluation de la somme nécessite une évaluation de f_i et a_i. Le nombre de règles applicables à une liste est en général très inférieur à r, on considère donc qu'il existe une constante n telle que :

$a_i = n * f_{i-1} * h$

Les a_i forêts ne sont pas ajoutées à la liste puisque les forêts objets dont elles sont issues sont effacées. On constate que le nombre de règles pouvant s'appliquer plusieurs fois à une même forêt et le nombre de règles différentes s'appliquant à une forêt sont en moyenne bornés par une autre constante k, on a donc :

$f_i = f_{i-1} + k * f_{i-1}$ ou plus simplement $f_i = k * f_{i-1}$, $k \geq 1$ (on ne considère que les forêts ajoutées et on ne tient pas compte des éventuelles règles de désambiguïsation). Comme $ar_1 = h + a_1$, on a $f_0 = 1$ et on peut donc écrire $f_i = k^i$,

et donc $AR(m, r) = \sum_{i=1}^{m} f_{i-1} * h + n * f_{i-1} * h$ ou plus simplement :

$$AR(m, r) = h*(n+1) \sum_{i=0}^{m-1} f_i = h*(n+1) \sum_{i=0}^{m-1} k^i$$

On a donc $AR(m, r) = h*(n+1)\frac{k^m - 1}{k - 1}$ et l'algorithme est donc en $h*nO(k^m) * O(r)$ c'est-à-dire $h*nO(r*k^m)$.

Il est donc exponentiel sur le nombre de mots et loin des performances polynomiales des algorithmes cités au §1.1.

2.3. Améliorations

Pour améliorer les performances du transducteur, nous proposons de réduire le nombre de structures incohérentes construites grâce à l'ajout de règles de désambiguïsation dont nous avons déjà parlé. Nous avons utilisé les travaux de [MERLE 82] sur ce sujet. L'idée de base est d'éliminer les séquences de catégories qui ne pourront jamais produire une analyse correcte et qui n'existent qu'à cause de la combinatoire liée aux ambiguïtés lexicales.

Une autre méthode est de restreindre l'application des règles grâce à l'ajout de traits sémantiques associés aux mots. Ces aspects seront développés au chapitre 3.

Mais ces deux méthodes ne portent pas sur l'algorithme lui-même mais sur les données qu'on lui fournit. Pour améliorer l'algorithme, nous proposons deux techniques qui font l'objet des deux paragraphes suivants.

2.3.1. Partage de structures

Une première idée est de limiter l'encombrement grâce à la factorisation des sous-forêts communes à plusieurs listes. Cette technique s'inspire de celle utilisée par Tomita et entraîne également une certaine factorisation des calculs.

Nous allons l'illustrer en reprenant l'exemple très simple du §2.2.5. La figure 2.6 reprend la figure 2.5 en factorisant les sous-forêts communes, donnant un graphe d'arbres dans lequel tout chemin constitue une forêt non ambiguë. A l'étape (3) nous n'avons pas fait apparaître toutes les évolutions mais seulement l'état du graphe après application de la règle A_N à "belle ferme". L'arc reliant N à A a disparu laissant la place à un nouvel arbre, toujours connecté au D de "la".

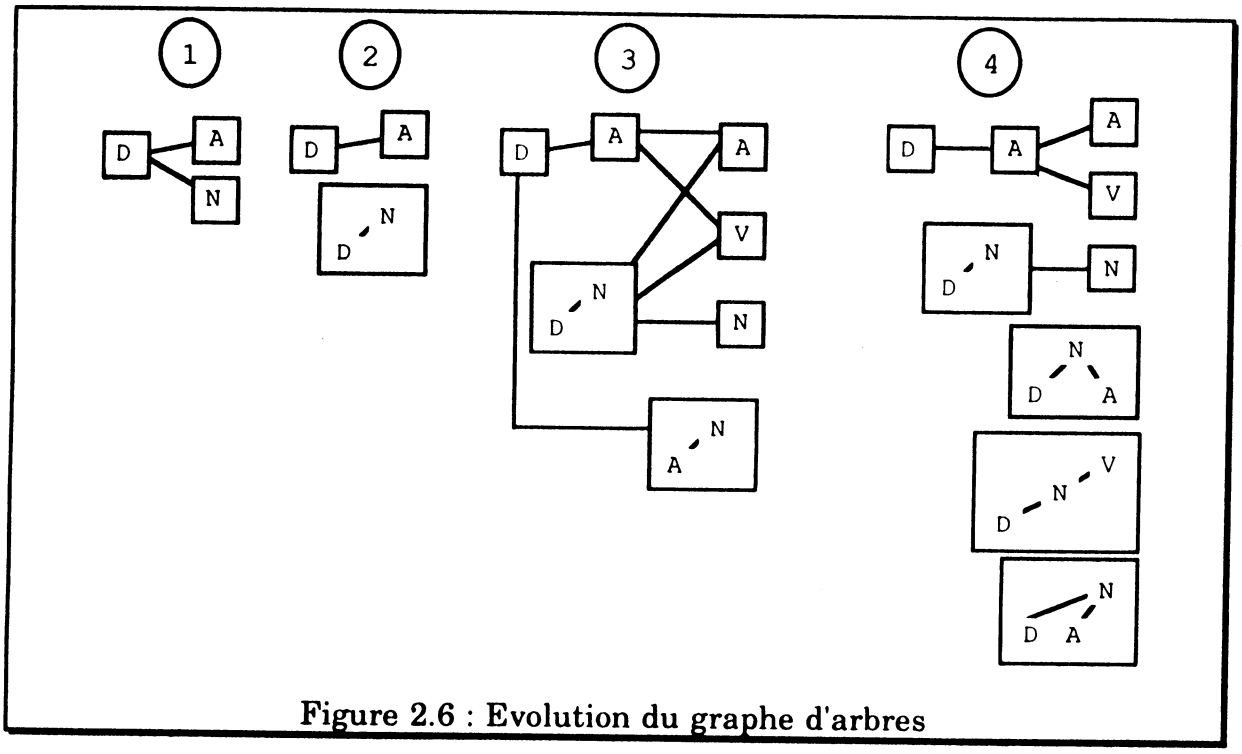


Figure 2.6 : Evolution du graphe d'arbres

La factorisation des calculs n'apparaît pas sur cet exemple mais devient claire si on ajoute les deux mots "le voile" à "la belle ferme" avec les interprétations D et $\begin{Bmatrix} V \\ N \end{Bmatrix}$. En effet la construction du groupe nominal (D)N ne sera faite qu'une fois alors qu'avec la structure de liste elle était faite pour chaque élément de la liste de l'étape (4).

2.3.2. Table d'accès aux règles

Une deuxième amélioration consiste à tenir compte de la structure des règles afin de limiter le nombre de tentatives d'application de règles à une forêt. Un

critère de sélection très simple et très efficace est de considérer pour chaque règle la (ou les) catégories de la contrainte associée au dernier schéma d'arbre d'un schéma de forêt. En effet le choix de l'analyse gauche-droite qui mène à l'introduction progressive des mots à droite des forêts induit une structure des règles du type : (forêt d'arbres partiels, arbre à un nœud) qui produit un rattachement de l'arbre à un arbre de la forêt.

Pour chaque catégorie x de C, on construit la liste des noms des règles dont le schéma d'instanciation (f_1, f_2,..., f_n) est tel que c s'unifie avec la contrainte de la racine r_n de f_n, c'est-à-dire tel que : $\sqcup(\{c\}, c_{\mathcal{P}(C)}(r_n)) \neq$ {_RIEN_}. Cette table peut aisément être construite une fois pour toutes à la compilation avec un parcours de la table des règles dans lequel on ne considère que les catégories fondamentales, c'est-à-dire les catégories x de C telles que si $\exists\, y \in C$ / y « x alors y = _RIEN_. Les listes de règles associées aux autres catégories sont construites par concaténation des listes de règles des catégories fondamentales : si x est une catégorie non fondamentale, on considère M_x = Max($\mathcal{M}in$(x)) et la liste des règles applicables à x est l'union des listes associées aux éléments de M_x.

On utilise cette table pour associer à chaque arbre situé en fin de forêt (en sommet de pile), c'est-à-dire tout arbre qui est un point d'entrée dans le graphe des arbres, une liste des règles applicables à cet arbre. Ces arbres proviennent de deux sources :

- l'introduction d'un nouveau mot : la liste des règles est alors la liste des règles associées à la catégorie du mot ;
- l'application d'un règle : une simple consultation de la racine de l'arbre situé en fin de la forêt produite donne une catégorie ou plus généralement un ensemble de catégories et la liste des règles est alors l'union des listes des différentes catégories.

Exemple :

Avec la grammaire très simple donnée ci-dessus, on aura les listes de règles :

D → ()

A → (N_A)

N → (D_N, A_N, V_N)

V → (N_V)

Bien que trivial, cet exemple montre que l'introduction d'un D ne produit aucune tentative d'application de règles et ce quel que soit le nombre d'arbres situés avant le D dans le graphe.

Le nombre de tentatives d'application de règles avec cette méthode n'est plus égal au nombre de règles mais proportionnel à ce nombre divisé par le nombre de catégories. Ce nombre dépend de la grammaire et la répartition des règles par catégories n'est évidemment pas uniforme. On trouvera en annexe B la grammaire que nous avons utilisée pour les tests et la table associée.

3

Représentation de la connaissance

Le transducteur présenté au chapitre 2 permet de construire des arbres de dépendances à partir d'une liste de mots (ou d'unités lexicales) considérée comme une forêt d'arbres à un nœud étiqueté par la catégorie lexicale du mot. Mais les structures obtenues sont trop pauvres pour faire l'objet d'une exploitation automatique ultérieure ; le traitement de phénomènes linguistiques complexes comme l'accord en genre et en nombre, les anaphores, la quantification, la coordination,... nécessite l'enrichissement des étiquettes des arbres avec des informations de divers niveaux :

- morphologique : le genre et le nombre des déterminants, noms, adjectifs,... le nombre, la personne, le mode et le temps des verbes, etc...
- syntaxique : la fonction d'un mot ou d'un groupe de mots dans la phrase (sujet, objet, ...), la modalité de la phrase (affirmative/interrogative, active/passive), etc...
- sémantique : la présence de traits sémantiques attachés aux mots peut permettre la levée de certaines ambiguïtés lexicales ou syntaxiques et la construction d'une structure sémantique est indispensable à l'exploitation du "sens" de la phrase.

Nous présentons dans ce chapitre le modèle unique que nous avons choisi pour représenter toutes ces connaissances et nous définissons les outils permettant de les manipuler.

3.1. Les Ψ-termes

Comme on l'a dit au §1.3, les structures de traits ont l'avantage de permettre la représentation avec un même formalisme de connaissances de divers niveaux (morphologique, syntaxique, sémantique) et cette uniformité de représentation induit une grande économie en terme d'outils de manipulation. Nous avons choisi de représenter la connaissance avec des structures de traits *typées* : les *Ψ-termes* [AIT-KACI 86B, 88, 89a]. C'est la notion de type qui caractérise les Ψ-termes par rapport aux structures de traits des GPSG ou aux f-structures des LFG. Alors que dans les GPSG par exemple un élément de connaissance est noté par une liste de paires <attribut valeur>, cette même liste sera regroupée dans un Ψ-terme sous la forme : type(liste de <attribut valeur>). Dans le premier cas, la valeur d'un attribut peut être une autre liste de paires ou une valeur booléenne (notée - ou +). Dans le second, la valeur d'un attribut est elle

même un Ψ-terme et la liste de paires associées peut être vide. Les valeurs "terminales" ne sont donc plus limitées à un ensemble booléen mais à un ensemble de types dit *types atomiques* ou *types simples*.

Nous donnons ci-dessous la syntaxe que nous utilisons pour décrire les Ψ-termes (les éléments terminaux sont en gras, le | dénote l'alternative, les éléments entre [ ] sont optionnels) :

```
<PsiTerme> ::= [ <Etiq> : ] <Tete> [ ( <LPaires> ) ]
             | <Etiq>
<LPaires> ::= <Paire> [ ; <LPaires> ]
<Paire> ::= <Label> => <PsiTerme>
<Etiq> ::= @<Symbole>
<Tete> ::= <Symbole>
<Label> ::= <Symbole>
```

Un **<Symbole>** est un identificateur au sens des langages de programmation. La `<Tete>` est le nom d'un type atomique, un `<Label>` est le nom d'un attribut et les `<Etiq>` permettent de noter l'adresse d'un sous-Ψ-terme et de définir des liens de *coréférences* entre les sous-Ψ-termes qui autorisent le partage de structures.

Exemple de Ψ-terme :

```
bébé(nom => @N : chaîne ;
     prénom => chaîne ;
     parents =>
       @P : couple (mari => adulte (nom => @N ;
                                    prénom => chaîne) ;
                    femme => adulte (nom => @N ;
                                     prénom => chaîne)) ;
     gardé_par => @P)
```

Ce Ψ-terme décrit un bébé dont le nom et le prénom sont des chaînes de caractères, dont les parents forment un couple où le mari et la femme portent le même nom que le bébé (contrainte imposée par le lien de coréférence noté @N). De plus ce bébé est gardé par ses parents.

Ce type de structure ressemble aux structures d'enregistrement des langages de programmation classique comme Pascal ou C (les types atomiques sont les noms de type et les labels les noms de champs), mais les liens de coréférence transforment la structure arborescente décrite par les types classiques en une structure de graphe orienté. On trouvera figure 3.1 le graphe associé au Ψ-terme de l'exemple ci-dessus.

L'interprétation naturelle qu'on peut associer à ce type de structure est de considérer les types atomiques comme des définitions en intension d'ensembles d'objets. Les labels peuvent alors être considérés comme des fonctions entre ces ensembles ; ainsi sur l'exemple on aura une fonction `nom` de l'ensemble des bébés dans l'ensemble des chaînes. On peut noter sous la forme de concaténation des labels la composition de ces fonctions ; ainsi on notera `parents.mari` la fonction de l'ensemble des bébés dans l'ensemble des adultes, qui associe à un bébé son père. Avec cette interprétation, les liens de coréférence dénotent des équivalences de fonctions qu'on peut noter par de simples égalités entre fonctions ; ainsi sur l'exemple on aura :

```
nom = parents.mari.nom = parents.femme.nom
gardé_par = parents
```

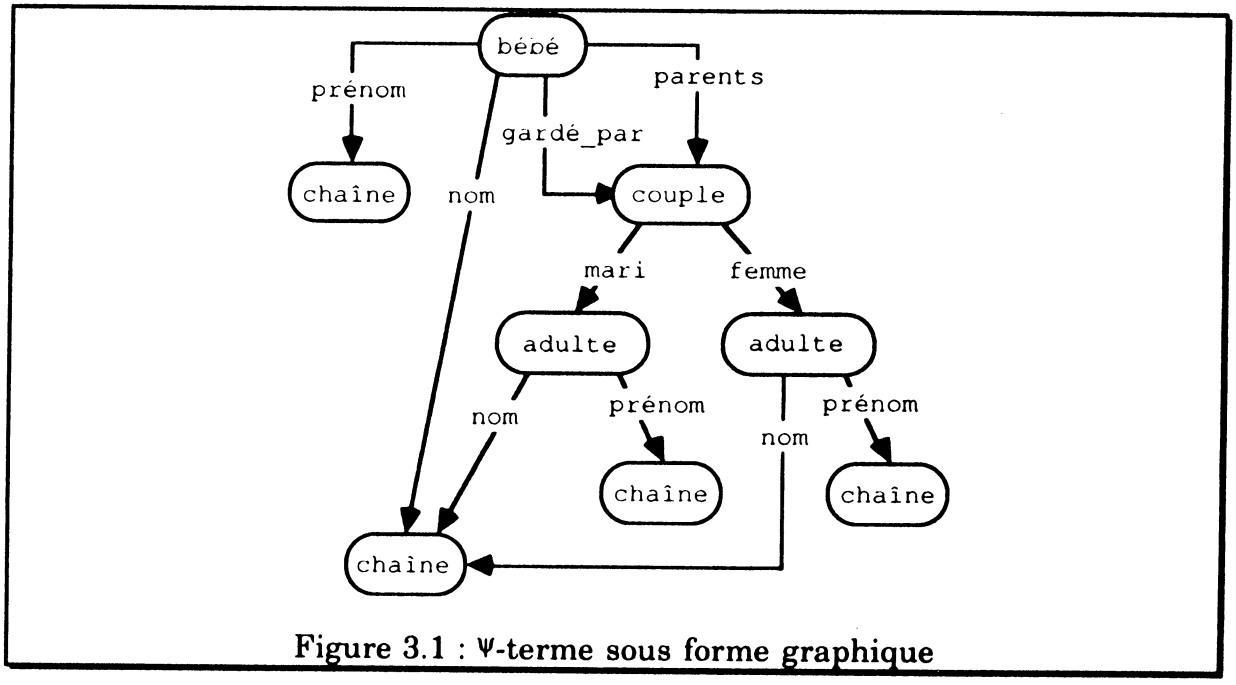


Figure 3.1 : Ψ-terme sous forme graphique

Un Ψ-terme décrit donc un sous-ensemble de l'ensemble des objets associés au type de la racine (bébé sur l'exemple). La restriction est définie en intension grâce à l'énumération de propriétés : existence de certaines fonctions et contraintes d'identité entre ces fonctions. Un des principaux attraits des Ψ-termes en ce qui concerne la représentation des connaissances est la possibilité de définir une relation d'ordre partiel sur les Ψ-termes qui traduise directement l'intersection entre les ensembles d'objets qu'ils décrivent. Cette relation d'ordre est une extension aux Ψ-termes d'une relation d'ordre partiel définie sur les types atomiques. Nous donnons au paragraphe suivant une définition plus formelle des Ψ-termes qui permet de donner ensuite la définition de l'ordre.

3.1.1. Définitions

Signature

L'ensemble des types atomiques est appelée *signature* et est un ensemble partiellement ordonné de symboles qui contient au moins deux symboles spéciaux notés_TOUT_ et _RIEN_[1]. On note Σ cet ensemble et « la relation d'ordre partiel. La hiérarchie de catégories décrite au §2.2.2 constitue un bon exemple d'un tel ensemble. Si on interprète un symbole S comme la définition en intension d'un ensemble d'objets $I(S)$, alors l'interprétation de la relation d'ordre est l'inclusion entre ensembles : si $S_1 \ll S_2$ alors $I(S_1) \subset I(S_2)$.

[1]La convention habituelle est plutôt d'utiliser $\top$ et $\bot$ mais _TOUT_ et _RIEN_ sont les symboles utilisés en pratique par notre système et sont plus facile à taper et à lire.

Labels et chemins

Soit $\mathcal{L}$ un ensemble fini de symboles appelés *labels*. Un *chemin* est une chaîne finie de labels, c'est-à-dire un élément fini de $\mathcal{L}^*$. On note la concaténation des symboles avec un point. Le terme de chemin vient de la représentation des Ψ-termes sous forme de graphe orienté : un chemin est la concaténation des symboles portés par les arcs du graphe. Un chemin peut être vu comme l'adresse d'un sous-Ψ-terme, l'adresse du Ψ-terme complet étant désignée par la chaîne vide notée ε.

Domaine

Un *domaine* $\mathcal{D}$ est un ensemble régulier de chemins, fermé par l'opération préfixe c'est-à-dire que si $\alpha.\beta \in \mathcal{D}$ avec $\alpha, \beta \in \mathcal{L}^*$, alors $\alpha \in \mathcal{D}$. Cette fermeture implique qu'un domaine contient toujours la chaîne vide notée ε. Le domaine d'un Ψ-terme est l'ensemble des chemins qu'on peut construire en parcourant le graphe depuis la racine.

Relation de coréférence

Une relation de coréférence @ sur un domaine $\mathcal{D}$ est une relation d'équivalence sur $\mathcal{D}$ invariante à droite, c'est-à-dire que si $\alpha, \beta \in \mathcal{D}$ et α @ β alors $\forall x \in \mathcal{L}$, si $\alpha.x$ et $\beta.x \in \mathcal{D}$ on a $\alpha.x$ @ $\beta.x$. Cette relation définit une partition de $\mathcal{D}$ en classes d'équivalences notée $\mathcal{D}_{/@}$.

Exemples :

Avec le Ψ-terme de l'exemple ci-dessus, on a :

$\mathcal{L}$ = {nom, prénom, parents, mari, femme, gardé_par}

$\mathcal{D}$ = {ε, nom, prénom, parents, gardé_par,
parents.mari, parents.femme,
parents.mari.nom, parents.mari.prénom,
parents.femme.nom, parents.femme.prénom,
gardé_par.mari, gardé_par.femme,
gardé_par.mari.nom, gardé_par.mari.prénom,
gardé_par.femme.nom, gardé_par.femme.prénom}

$\mathcal{D}_{/@}$ = { {ε}, {prénom}, {parents, gardé_par}
{nom, parents.mari.nom, parents.femme.nom,
gardé_par.mari.nom, gardé_par.femme.nom},
{parents.mari.prénom, gardé_par.mari.prénom},
{parents.femme.prénom, gardé_par.femme.prénom},
{parents.mari, gardé_par.mari}
{parents.femme, gardé_par.femme}}

On peut noter que la syntaxe externe des Ψ-termes ne fait pas apparaître tous les chemins : on ne note qu'une fois une structure partagée.

Fonction de typage

On peut associer à chaque chemin de $\mathcal{D}$ un type dans Σ qui est le type atomique de la racine du Ψ-terme désigné par ce chemin. La fonction correspondante est étendue à $\mathcal{L}^*$ sous la forme d'une fonction Ψ, dite *fonction de typage*, qui à chaque élément de $\mathcal{L}^*$ qui n'est pas dans $\mathcal{D}$ associe le type _TOUT_. Ainsi sur l'exemple : Ψ(ε) = bébé, Ψ(nom) = chaîne, Ψ(nom.prénom) = _TOUT_.

Avec ces définitions, un Ψ-terme est entièrement caractérisé par un triplet ($\mathcal{D}$, @, Ψ) et l'ordre partiel sur la signature peut être étendu aux Ψ-termes par la définition suivante.

Ordre sur les Ψ-termes

On note cet ordre avec le même symbole « que l'ordre sur la signature car tous les éléments de Σ sont aussi des Ψ-termes. On appelle $\mathcal{T}$ l'ensemble des Ψ-termes définis sur Σ et $\mathcal{L}$.

Un Ψ-terme T_1 est un *sous-type* d'un Ψ-terme T_2, qu'on note $T_1 \ll T_2$, si et seulement si :

- soit T_1 = _RIEN_
- soit $T_1 = (\mathcal{D}_1, @_1, \Psi_1)$, $T_2 = (\mathcal{D}_2, @_2, \Psi_2)$ et :

$$\mathcal{D}_2 \subset \mathcal{D}_1 \quad (1)$$

$$@_2 \subset @_1 \quad (2)$$

$$\forall x \in \mathcal{L}^*, \Psi_1(x) \ll \Psi_2(x) \quad (3)$$

Au niveau syntaxique, T_2 est une restriction de T_1 (T_2 est un sous-graphe de T_1), alors qu'en terme d'interprétation l'ensemble défini par T_1 est un sous ensemble de l'ensemble défini par T_2.

Exemple :

Avec Σ = {bébé, couple, personne, adulte, chaîne, jean, Dupond} où :
bébé « personne, adulte « personne, jean « chaîne, Dupond « chaîne,
et T_1 =
```
bébé(nom => @N : Dupond ;
     prénom => jean ;
     parents =>
       couple (mari => adulte (nom => @N);
               femme => adulte (nom => @N))
```
T_2 =
```
personne(nom => @N : chaîne ;
         parents =>
           couple (mari => personne ;
                   femme => personne (nom => @N))
```
on a $T_1 \ll T_2$. On voit que tous les chemins de T_2 apparaissent dans T_1 (ligne (1) de la définition), que tous les liens de coréférence de T_2 sont dans T_1 (ligne (2)) et qu'à chemin égal, les types atomiques de T_1 sont inférieurs à ceux de T_2. D'autre part, il est clair que l'ensemble décrit par T_1 est inclus dans l'ensemble décrit par T_2.

3.1.2. Unification

Le principal intérêt de l'extension de l'ordre sur la signature aux Ψ-termes est de pouvoir étendre également l'unification. L'unification sur Σ, $\sqcup_\Sigma$ définie au §2.2.2.1 associe à deux types atomiques l'ensemble des éléments maximaux de l'intersection des ensembles de minorants de ces deux types. [AIT-KACI 84] a montré que si l'ordre sur Σ est tel que pour toute paire (x, y) de Σ il existe une borne inférieure b alors pour toute paire (s, t) de $\mathcal{T}$ la borne inférieure existe également. Autrement dit, si (Σ, «) est un demi-treillis inférieur, alors ($\mathcal{T}$, «) est aussi un demi-treillis inférieur.

Nous allons dans un premier temps définir l'unification sur $\mathcal{T}$, en supposant que $(\Sigma, \ll)$ est un demi-treillis inférieur (nous renvoyons au §3.1.4 la discussion concernant un ordre partiel quelconque). Avec cette propriété, l'unification sur Σ, qui à chaque paire d'éléments associe un ensemble d'éléments peut être redéfinie comme suit :

$$\sqcup_\Sigma : \Sigma \times \Sigma \rightarrow \Sigma$$
$$(x, y) \rightarrow b$$

où b est la borne inférieure de (x, y) c'est-à-dire l'unique élément de l'ensemble $\mathrm{Max}(\mathit{Min}(x) \cap \mathit{Min}(y))$.

On peut alors définir la borne inférieure de deux Ψ-termes T_1 et T_2 comme le plus grand Ψ-terme B inférieur à la fois à T_1 et T_2. Autrement dit, on a les propriétés : $B \ll T_1$, $B \ll T_2$ et s'il existe $T \in \mathcal{T}$ tel que $T \ll T_1$, $T \ll T_2$ alors $T = B$.

L'unification sur $\mathcal{T}$ est alors définie comme la fonction :

$$\sqcup : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$$
$$(T_1, T_2) \rightarrow B$$

où B est la borne inférieure de T_1 et T_2.

Informellement[1], la borne inférieure $B = (\mathcal{D}, @, \Psi)$ de deux Ψ-termes $T_1 = (\mathcal{D}_1, @_1, \Psi_1)$ et $T_2 = (\mathcal{D}_2, @_2, \Psi_2)$ est obtenue de la manière suivante : on calcule d'abord $\mathcal{D}^u = \mathcal{D}_1 \cup \mathcal{D}_2$ car il est clair que $\mathcal{D}$ contiendra au moins $\mathcal{D}^u$. On procède ensuite au calcul des extensions réflexives de $@_1$ et $@_2$ à $\mathcal{D}^u$, c'est-à-dire que pour chaque élément x de $\mathcal{D}^u$, on s'assure que $x@x$. On calcule alors la fermeture transitive $@^u$ de l'union de ces deux extensions, c'est-à-dire que pour tout x, y, z de $\mathcal{D}^u$, si $x@^u y$ et $y@^u z$ alors $x@^u z$. Il reste à s'assurer que $@^u$ est bien invariante à droite : si $\alpha@^u\beta$ alors s'il existe γ tel que $\alpha.\gamma \in \mathcal{D}^u$, on doit avoir $\alpha.\gamma@^u\beta.\gamma$. Si ce n'est pas le cas, on complète $@^u$ et éventuellement $\mathcal{D}^u$ si $\beta.\gamma \notin \mathcal{D}^u$; on obtient alors @ et $\mathcal{D}$. Ψ est obtenue en calculant pour tout x de $\mathcal{D}$: $\Psi(x) = \sqcup_\Sigma(\Psi_1(x), \Psi_2(x))$. Cette définition doit être modulée par le fait que s'il existe $y \in \mathcal{D}$ tel que $\Psi(y) =$ _RIEN_ alors B = _RIEN_.

On s'assure ainsi que $B \ll T_1$ et $B \ll T_2$ et le fait qu'il soit le plus grand se déduit du mode de calcul des types atomiques (avec $\sqcup_\Sigma$) et du fait qu'on a dans $\mathcal{D}$ tous les chemins de $\mathcal{D}_1$ et $\mathcal{D}_2$ et aucun autre et donc qu'un Ψ-terme supérieur à B (ayant a priori moins de chemins) ne pourrait être à la fois inférieur à T_1 et T_2. Il est clair que cette opération d'unification est commutative et que si $T_1 \ll T_2$ alors $B = T_1$.

Exemple d'unification :

Avec la signature de l'exemple donné plus haut et :

```
T1 = bébé(prénom => jean ;
          parents =>
            couple (mari => adulte (nom => @N : chaîne);
                    femme => adulte (nom => @N))
```

[1]La définition formelle est donnée par [AIT-KACI 86a], pages 313, 314

```
T2 = personne(nom => @N : Dupond ;
              parents =>
                couple (mari => personne ;
                        femme => personne (nom => @N))
on aura :
B = bébé(nom => @N : Dupond ; prénom => jean ;
         parents =>
           couple (mari => adulte (nom => @N);
                   femme => adulte (nom => @N))
```

L'unification peut également être vue comme un mécanisme d'*héritage* dans le sens où la borne inférieure de deux Ψ-termes *hérite* des propriétés (attributs) propres à chaque Ψ-terme et synthétise les propriétés communes aux deux.

3.1.3. Algorithme d'unification

Une qualité importante des Ψ-termes réside dans l'efficacité de leur mise en œuvre. Nous présentons ici un algorithme efficace d'unification basé sur l'algorithme qui décide de l'équivalence de deux automates d'états finis [AHO & HOP & ULL 74]. La version que nous avons mise en œuvre dérive directement de la version présentée dans [AIT KACI 86b], mais nous n'avons pas traité le problème UNION/TROUVE, arguant du fait que contrairement aux automates d'états finis (qui peuvent contenir plusieurs centaines d'états), les Ψ-termes (utilisés pour représenter la connaissance) sont de taille limitée.

Nous supposons que les Ψ-termes sont codés dans un vecteur `T` indexé par un intervalle d'entiers noté `Indice` ne contenant pas 0. Chaque élément T[i], i ∈ `Indice`, du vecteur contient trois champs :

`type` : un type atomique, élément de Σ

`sous-termes` : un ensemble de couples `(l, x)` où `l` est un label, élément de $\mathcal{L}$ et x ∈ Indice.

`coréférence` : un élément de `Indice` ∪ {0}.

La coréférence permettra de coder les liens de coréférence entre deux éléments des deux Ψ-termes à unifier. Elle crée une structure d'arbre inversé où les liens vont des feuilles vers la racine. Un tel arbre code une classe de coréférences dont le représentant est la racine. Un élément du vecteur sera le représentant de sa classe si le lien de coréférence est 0.
Un Ψ-terme est donc simplement désigné par l'indice de sa racine dans T.

Exemple :

```
Le Ψ-terme personne(nom => @N : Dupond ;
                   parents =>
                     couple (mari => personne ;
                             femme => personne (nom => @N))
```

sera codé dans T :

Indice	Type	Sous-termes	Coréférence
1	personne	{(nom, 2) (parents, 3)}	0
2	Dupond	{}	0
3	couple	{(mari, 4) (femme, 5)}	0
4	personne	{}	0
5	personne	{(nom, 2)}	0

On suppose définies les primitives suivantes :
Classe(i) : rend le représentant de la classe de i (Indice)
Labels(i) : rend l'ensemble des labels extraits des sous-termes de i
Valeur(i, l) : si l ∈ Labels(i), rend l'indice associé à l dans l'ensemble des sous-termes de i.
On utilise les notations Pascal pour l'accès aux éléments de T et aux champs.

Prédicat Unification(T_1, T_2, B)
Entrées : T_1, T_2, indices dans T des racines des deux Ψ-termes à unifier
Sorties : une valeur booléenne indiquant si l'unification a réussi et si oui alors B est l'indice dans T de la racine de la borne inférieure de T_1 et T_2

```
ResteATraiter := {(T1, T2)}
possible := vrai
Tant que ResteATraiter ≠ ∅ faire
    Supprimer le premier élément (x, y) de ResteATraiter
    x := Classe(x)
    y := Classe(y)
    Si x ≠ y alors
        t := ⊔Σ(T[x].type, T[y].type)
        Si t = _RIEN_ alors possible := faux
        sinon
            T[x].coréférence := y
            T[y].type := t
            pour chaque l ∈ labels(x) ∪ labels(y) faire
                si l ∈ labels(x) ∩ labels(y) alors
                    ResteATraiter :=
                        ResteATraiter ∪ {(valeur(x, l), valeur(y, l))}
                sinon
                    si l ∉ labels(y) alors
                        T[y].sous-termes := T[y].sous-termes
                                ∪ {(l, Classe(Valeur(x, l)))}
Si possible alors
    Reconstruit(T2, B)
    Unification := vrai
sinon
    Unification := faux
```

Algorithme Reconstruit(R, B)
Entrées : R, indice dans T de la racine du Ψ-terme à reconstruire
Sorties : B, indice dans T de la racine du Ψ-terme reconstruit par fusion des classes de coréférences de T en un seul élément.

```
c := Classe(R)
Si DéjàTraité(c) alors
    B := Représentant(c)
Sinon
    B := premier indice disponible dans T
    T[B].type := T[c].type
    T[B].coreference := 0
```

```
T[B].sous-termes := {}
pour chaque élément (l, x) de T[c].sous-termes faire
    Reconstruit(x, nouv)
    T[B].sous-termes := T[B].sous-termes ∪ {(l, nouv)}
```

Le prédicat DéjàTraité permet de tester si le représentant d'une classe a déjà fait l'objet d'une reconstruction et si c'est le cas la primitive Représentant donne le résultat obtenu. On évite ainsi de dupliquer les éléments de la même classe de coréférences et surtout on garantit l'arrêt de l'algorithme en cas de coréférence circulaire.

Par rapport à la version de Aït-Kaci, nous avons séparé la reconstruction de l'unification proprement dite afin de pouvoir utiliser l'algorithme Reconstruit comme une procédure de copie de Ψ-termes. En effet, si tous les champs coréférence de R sont à 0, l'algorithme Reconstruit crée une copie de R.

Cet algorithme est à peu près linéaire en $n_1 + n_2$ où n_i est le nombre d'éléments de T_i. L'à peu près est fonction de la longueur des chaînes de coréférences qui est limitée en pratique.

3.1.4. Codage et disjonctions

L'algorithme ci-dessus est défini dans le cas où la signature Σ possède une structure de demi-treillis inférieur. Avec une telle structure, on peut mettre en œuvre un codage de la signature qui permet une très grande efficacité dans le calcul de $\sqcup_\Sigma$. Il suffit de projeter la structure de Σ dans une restriction du treillis construit avec la relation d'inclusion sur l'ensemble $\mathcal{P}(\Sigma)$ des parties de Σ de la manière suivante :

- _RIEN_ → ∅
- x ∈ Σ - {_RIEN_} → *Min*(x) - {_RIEN_}

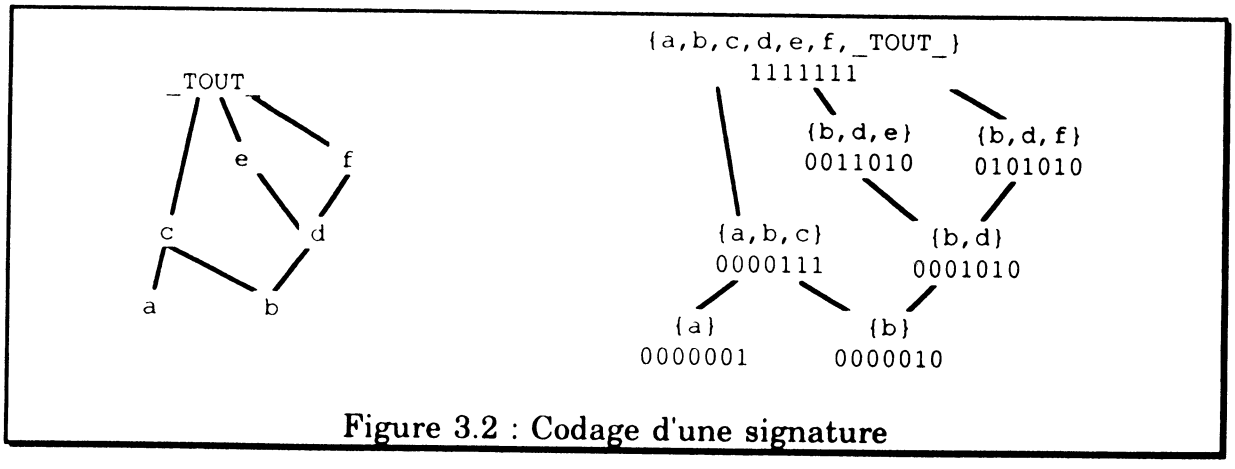


Figure 3.2 : Codage d'une signature

Si on utilise pour coder les ensembles des chaînes de bits, le code de la borne inférieure de deux éléments de Σ est obtenue par un simple et-logique entre les codes des deux éléments. Comme _RIEN_ ne contribue pas, la longueur d'une chaîne de bits est exactement égale au nombre d'éléments de la signature. On

trouvera figure 3.2 un exemple de signature et la projection associée ; on ne fait pas apparaître _RIEN_ qui sera systématiquement codé par une chaîne de 0 ; on ne fait pas non plus apparaître les liens qui peuvent se déduire des autres par transitivité (comme entre a ou b et _TOUT_).

Ce codage permet d'obtenir un coût constant de l'unification sur Σ, pour peu que la signature puisse être codée sur un mot de la machine sur laquelle est implanté le système. Une telle propriété est illusoire pour des applications de traitement de la langue où l'on peut imaginer que la signature pourra comporter plusieurs centaines, voire quelques milliers d'éléments. [AIT-KACI 89b] propose des méthodes de codage beaucoup plus efficaces en terme d'encombrement grâce à l'exploitation de la structure de la signature. Nous ne les détaillerons pas ici, préférant présenter les problèmes posés par une signature qui n'est pas un demi-treillis inférieur et par les Ψ-termes disjonctifs en général.

Un problème se pose quand pour deux types atomiques de Σ il n'existe pas de borne inférieure. Comme nous avons ajouté le type _RIEN_ qui est inférieur à tous les éléments de Σ, ce problème n'apparaît que si l'intersection des ensembles de minorants des deux types possède plusieurs éléments maximaux incomparables. Avec la signature de la figure 3.3a, on aura $\amalg_\Sigma(c, d) = \{a, b\}$.On peut résoudre ce problème si au lieu de considérer Σ comme ensemble de types atomiques, on considère la projection de Σ dans le plus petit demi-treillis qui le contienne et qui préserve ses propriétés. L'idée est simplement de construire la borne inférieure quand celle-ci n'existe pas. On peut ainsi sur la figure 3.3a ajouter un type a_b qui deviendrait la borne inférieure de c et d.

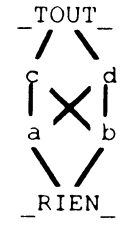


Figure 3.3a : Signature qui n'est pas un demi-treillis inférieur

Cet ajout peut être formalisé et réalisé automatiquement en projetant Σ dans le sous-ensemble de $\mathcal{P}(\Sigma)$ dont les éléments sont incomparables deux à deux. Ces ensembles sont appelés *co-chaînes* de Σ ou *couronnes*. Ils sont ordonnés par la relation d'ordre (notée encore «) définie comme suit :

$A \ll B$ si et seulement si $\forall t \in A, \exists u \in B / t \ll u$.

La figure 3.3b montre la projection de la signature de la figure 3.3a.

La projection est simplement obtenue en associant à chaque élément t de Σ le singleton {t}. Il est clair que si $t \ll u$ alors $\{t\} \ll \{u\}$ et l'unification sur $\mathcal{P}(\Sigma)$ telle que nous l'avons définie au §2.2.2.1 donne la borne inférieure de deux couronnes.

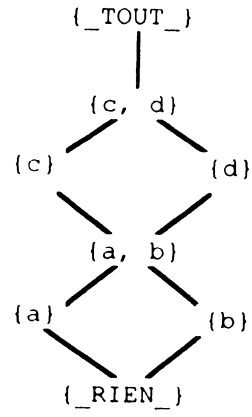


Figure 3.3b : Projection dans un demi-treillis

On peut appliquer au demi-treillis obtenu les techniques de codage booléen définies ci-dessus et faire toutes les manipulations de Ψ-termes avec ces codes. On peut ainsi calculer l'unification de deux Ψ-termes de manière efficace et on n'a besoin de connaître le résultat d'une unification de types atomiques (un type ou un ensemble) qu'au moment de l'affichage du résultat. Ce résultat peut être interprété de deux façons :

(1) comme la borne inférieure des deux Ψ-termes si on considère la projection de Σ sur le demi-treillis comme la signature ;

(2) comme l'ensemble des maximaux de l'intersection des minorants des deux Ψ-termes si on considère Σ comme signature.

Exemple :

Avec la signature de la figure 3.3a et les deux Ψ-termes :

T_1 = c(un => b)

T_2 = d(un => c ; deux => a)

Le résultat de l'unification avec l'interprétation (1) pourrait s'écrire :

{a, b}(un => b ; deux => a)

Tandis qu'avec l'interprétation (2) on aura plutôt :

{a(un => b ; deux => a), b(un => b ; deux => a)}

La différence peut paraître subtile mais en fait la mise en œuvre de l'interprétation (2) autorise l'écriture d'ensembles de Ψ-termes dont l'interprétation naturelle sera la *disjonction* de types. L'ensemble des objets décrit par une disjonction sera l'union des ensembles associés aux éléments de la disjonction. Si on autorise l'écriture d'un ensemble de Ψ-termes partout où on peut écrire un Ψ-terme, alors on peut écrire par exemple :

```
bébé(nom => @N : chaîne ;
     prénom => chaîne ;
     parents => @P : couple (mari => adulte ;
                             femme => adulte) ;
     gardé_par => {@P, adulte(profession => baby_sitter)})
```

Cette possibilité est extrêmement intéressante pour la représentation des connaissances et notamment pour le traitement de la langue naturelle où les ambiguïtés sont nombreuses.

Exemple :

Les informations morphologiques attachées à la forme "aime" pourront s'écrire :

```
UL(cat => verb;
   conjug => {_TOUT_(mode => {indicatif, subjonctif};
                     temps => présent;
                     nombre => singulier;
                     personne => {première, troisième}) ;
              _TOUT_(mode => impératif;
                     nombre => singulier;
                     personne => deuxième)})
```

De la même manière qu'on a étendu l'unification sur Σ à $\mathcal{P}(\Sigma)$, on peut étendre l'unification sur les Ψ-termes aux ensembles de Ψ-termes à condition de se limiter à des ensembles finis [AIT-KACI 86a]. Bien qu'on puisse trouver des algorithmes d'unification intégrant les structures disjonctives [EISELE 88] [KASPER 87], nous ne les avons pas mis en œuvre dans notre transducteur, nous contentant d'autoriser les disjonctions de types atomiques (comme on l'a vu au §2.2.2 pour les catégories).

3.2. Utilisation des Ψ-termes

Nous décrivons dans ce paragraphe comment les Ψ-termes sont intégrés au transducteur décrit au chapitre 2, d'un point de vue linguistique (quelles informations contiennent-ils ?) et d'un point de vue informatique (quel usage faire de ces informations et comment ?).

Nous associons à chaque nœud des arbres de dépendances un Ψ-terme qui porte les variables morphologiques, les fonctions syntaxiques et éventuellement les traits et relations sémantiques. L'idée de base est de construire parallèlement à la structure de dépendance une structure (un Ψ-terme) qui puisse être exploitée dans diverses applications : détection/correction d'erreurs, interfaces homme/machine en français, enseignement assisté du français,...

3.2.1. Représentation des connaissances linguistiques

Nous supposons que les unités lexicales (UL) en entrée du transducteur sont des Ψ-termes (nous verrons comment adapter l'analyseur morphologique au chapitre 5). Chaque UL est attachée à la racine d'un arbre à un nœud avant d'être intégrée à la structure de travail du transducteur (voir §2.2).

Comme la catégorie d'une UL joue un rôle particulier dans la construction des arbres, nous avons prédéfini dans le transducteur une signature minimale qui contient :

- _TOUT_ et _RIEN_ ;
- CLS qui est la catégorie la plus générale, c'est-à-dire que toute catégorie x doit être telle que x « CLS ;
- UL qui est le type d'une unité lexicale.

Une UL devra de plus comporter au minimum un attribut CAT dont la valeur est la catégorie. La plus petite UL (en termes syntaxiques) s'écrit donc :
UL(CAT => CLS)

En termes d'interprétation, elle représente l'ensemble de toutes les unités lexicales possibles puisque elle est supérieure à tout Ψ-terme représentant une UL.

La définition d'autres attributs associés aux UL est laissée au linguiste chargé de la définition du lexique et de l'écriture des règles de grammaire. Nous donnons quand même ci-dessous quelques définitions d'attributs qui nous paraissent indispensables, c'est-à-dire qu'ils apparaîtront sous cette forme ou une autre dans tout système d'analyse d'une langue. Ces définitions seront de plus utilisées dans les exemples.

3.2.1.1. Variables morphologiques

Toutes les variables morphologiques classiques (genre, nombre, temps,...) peuvent être codées par des attributs de manière simple.

On donne ci dessous quelques exemples de Ψ-termes et en commentaire (après deux tirets) une forme pouvant correspondre à ce Ψ-terme.

```
UL(CAT => subc;                                          -- "bébé"
   gnr => masculin; nbr => singulier)
UL(CAT => adjq; nbr => singulier;                        -- "jeune"
   gnr => {masculin, féminin}) -- le genre est quelconque
UL(CAT => pron;                                          --"le"
   gnr => masculin; nbr => singulier;
   usage => cod)                  -- pronom complément d'objet
UL(CAT => verb;
   mode => indicatif; tps => présent;                    -- "irons"
   pers => première; nbr => pluriel)
```

On peut envisager le regroupement de certains attributs sous un type commun en vue d'une application particulière, ainsi pour contrôler l'accord en genre et en nombre on pourra écrire :

```
UL(CAT => adjq ;                                         -- "jeune"
   accord => _TOUT_(nbr => singulier;
                    gnr => {masculin, féminin}))
```

ou utiliser les deux notations grâce au partage de structure :

```
UL(CAT => adjq;                                          -- "jeune"
   accord => _TOUT_(nbr => @N : singulier;
                    gnr => @G : {masculin, féminin})
   nbr => @N; gnr => @G)
```

Certaines variables morphologiques ne peuvent pas être calculées par l'analyseur morphologique et doivent donc faire l'objet d'un traitement au niveau syntaxique ; ainsi pour les temps composés par exemple, on devra combiner les informations de l'auxiliaire et du participe passé :

```
UL(CAT => xav;                                           -- "a"
   mode => indicatif; tps => présent;
   pers => troisième; nbr => singulier)
UL(CAT => ppas ; nbr => singulier)                       -- "mangé"
```

pour produire :

```
UL(CAT => verb;                                          -- "a mangé"
   mode => indicatif; tps => passé-composé ;
   pers => troisième; nbr => singulier)
```

Suivant la grammaire, ce Ψ-terme remplacera celui du participe ou de l'auxiliaire.

3.2.1.2. Fonctions et variables syntaxiques

Les fonctions syntaxiques définissent les rôles des mots ou groupes de mots à l'intérieur d'un groupe. Pour une proposition on aura, par exemple, les fonctions sujet, objet et prédicat. Dans un groupe nominal, on a des fonctions précisant la définition, la quantification,... Leur codage nécessite d'établir des liens entre les mots ou entre les groupes. Ces liens seront calculés sur la base de l'arbre de dépendances et pourront être intégrés au Ψ-terme du gouverneur du groupe (classiquement le verbe pour une proposition et le nom pour un groupe nominal). Il semble naturel de coder ces liens sous la forme d'attributs dont la valeur est le Ψ-terme lié au groupe image de la fonction. Ainsi pour la proposition "le bébé mange la soupe", on aura par exemple le Ψ-terme (on n'a pas fait apparaître les variables morphologiques) :

```
UL(CAT => verb;                                    -- "mange"
   syn => proposition(
     prédicat => MANGER;
     sujet => UL(
       CAT => subc;                                -- "bébé"
       syn => groupe_nominal(
         Déterminant => UL(CAT => Det;             -- "le"
                           Définition => Défini)))
     objet =>  UL(
       CAT => subc;                                -- "soupe"
       syn => groupe_nominal(
         Déterminant => UL(CAT => Det;             -- "la"
                           Définition => Défini))))
```

Certaines variables syntaxiques attachées aux propositions (affirmative/interrogative, active/passive, ...) ou aux groupes nominaux (on peut considérer la définition comme une simple variable plutôt que comme une fonction), peuvent être attachées de la même manière aux gouverneurs. On peut par exemple ajouter un trait `voix` de valeur `active` à la `proposition` dans le Ψ-terme ci-dessus.

Remarques :

- Il va de soi que les Ψ-termes attachés aux racines des arbres de dépendances ne se rapportent plus seulement au nœud racine mais à tout l'arbre. La dualité de représentation permet de s'affranchir de la limite des grammaires de dépendances pures dans lesquelles il est justement difficile de coder des informations sur les groupes.
- Il ne faut pas voir les types des groupes (proposition, groupe nominal,...) comme des répliques des constituants des théories Chomskyennes et dérivées mais plutôt comme une classification par niveau des constituants les plus significatifs d'une phrase, un peu à la manière des grammaires utilisées par [WINOGRAD 73] où trois niveaux apparaissent : mot, groupe et proposition.

L'utilisation de ces fonctions pour les verbes permet le codage d'une forme de sous-catégorisation : on intègre aux informations de base du verbe (c'est-à-dire dans le lexique) une description des fonctions syntaxiques et on peut donc a

priori contraindre les compléments fondamentaux (sujet, objet direct, objet indirect). La description de "mange" dans le lexique pourra s'écrire :

```
UL(CAT => verb;                                          -- "mange"
   syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal)
     objet => UL(syn => groupe_nominal)))
```

alors que celle de "donne" s'écrira :

```
UL(CAT => verb;                                          -- "donne"
   syn => proposition(
     prédicat => DONNER;
     sujet => UL(syn => groupe_nominal)
     objet => UL(syn => groupe_nominal)
     objet-indirect => UL(
        CAT => à ;
        syn => groupe-prépositionnel)))
```

où à est par exemple déclaré dans la signature comme une sorte de préposition (à « prep) pouvant être utilisée comme catégorie de "à" "au" "aux".

3.2.1.3. Traits et relations sémantiques

On peut encore enrichir les Ψ-termes en ajoutant des attributs de nature sémantiques :

- soit comme des traits simples précisant simplement le concept associé au mot (ou au groupe) ;
- soit des traits à valeur complexe précisant les relations sémantiques entre les groupes.

On pourra par exemple avoir :

```
UL(CAT => subc;                                          -- "bébé"
   sém => PERSONNE)
UL(CAT => subc;                                          -- "soupe"
   sém => COMESTIBLE)
UL(CAT => verb;                                          -- "mange"
   sém => MANGER(
     agent => ANIME;
     patient => COMESTIBLE)))
```

Ces informations sémantiques pourront être utilisées pour contraindre les informations syntaxiques, ainsi on pourra indiquer que le sujet du verbe est l'agent de l'action alors que l'objet est le patient. On écrira pour "mange" :

```
UL(CAT => verb;                                          -- "mange"
   syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal;
                 sém => @S : ANIME)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(
     agent => @S;
     patient => @O)))
```

Ce type de contrainte peut être utilisé pour la résolution de certaines ambiguïtés structurales et pour la recherche des référents des pronoms.

3.2.2. Exploitation par la grammaire

L'utilisation des connaissances portées par les Ψ-termes nécessite l'augmentation du langage d'écriture des règles de la grammaire : pour accéder aux valeurs des attributs, pour tester ces valeurs et pour les modifier.

Nous avons ajouté deux zones aux règles :

- une zone décrivant des conditions dont la vérification conditionne l'application de la règle. Ces conditions portent sur les attributs des Ψ-termes associées aux arbres mis en jeu par la règle, elles sont donc vérifiées après instanciation.
- une zone décrivant des actions à exécuter sur la forêt construite. Ces actions modifient les attributs des Ψ-termes associés à la forêt.

Ces deux zones sont décrites à l'aide d'un langage d'expressions dont nous décrivons ici les éléments fondamentaux[1], la syntaxe complète du langage d'écriture des règles est donnée au chapitre 4.

3.2.2.1. Constantes

Le langage ne manipule que des Ψ-termes et donc tout Ψ-terme dont la valeur est compatible avec la signature est une constante valide.
Exemples :

```
masculin
UL(CAT => subc)
```

3.2.2.2. Variables : accès à la valeur d'un attribut

Les Ψ-termes fournissent un moyen évident d'accès aux attributs : ce sont les chemins qui donnent l'adresse d'un sous Ψ-terme sous la forme de la concaténation des arcs à traverser depuis la racine jusqu'au sous-Ψ-terme. Ainsi dans la dernière description de "mange" donnée ci-dessus :

syn.prédicat désigne MANGER
sém.agent et syn.sujet.sém désignent ANIME
syn.objet désigne UL(syn=>groupe_nominal;sém=>@O:COMESTIBLE)

Il reste à définir l'accès à un Ψ-terme d'un nœud particulier des arbres mis en jeu par la règle. Ceci est simplement réalisé en ajoutant devant un chemin la référence du nœud, c'est-à-dire le numéro qui permet de désigner le nœud dans les schémas de forêt de la règle. Pour les variables de forêt, le seul accès possible consiste à désigner la forêt entière.

Tout chemin constitue un nom de variable dont la valeur pourra être lue ou modifiée.
Exemples :

```
1.gnr, 1.syn.sujet, 1
```

Un numéro seul désigne tout le Ψ-terme associé au nœud.

L'interprétation de la valeur d'une variable dépendra du contexte :

- une variable pourra être vue comme une valeur booléenne qui sera vraie si le chemin existe et fausse sinon ;

[1]Ces éléments sont ceux qui ont été mis en oeuvre dans la version actuelle de notre système

- une variable pourra être vue comme un Ψ-terme et dans ce cas, si le chemin n'existe pas, la valeur sera _TOUT_.

3.2.2.3. Expressions logiques : conditions

On pourra utiliser les trois opérateurs logiques classiques ET, OU, NON avec des arités n-aires pour ET et OU et unaire pour NON.

On définit en plus l'opérateur binaire UNIF dont la valeur est vrai si les deux Ψ-termes passés en paramètre s'unifient et faux sinon.

Exemples[1] :

`ET(UNIF(1.gnr, 2.gnr), UNIF(1.nbr, 2.nbr))` sera vrai si le genre et le nombre des nœuds 1 et 2 s'unifient.

`OU(1.sujet, 1.objet, NON(1.sém.agent))` sera vrai si un des 2 premiers chemins cités existent ou si le dernier n'existe pas.

3.2.2.4. Expressions à valeur de Ψ-terme : actions

L'opérateur UNIF peut également être utilisé pour calculer la borne inférieure de deux Ψ-termes. S'ils s'unifient la valeur est la borne inférieure mais s'il ne s'unifient pas la valeur est, comme pour les variables non définies, le Ψ-terme _TOUT_.

L'opérateur fondamental pour la modification des Ψ-termes est l'opérateur d'affectation AFFECT. C'est un opérateur binaire dont le premier argument est une variable et le deuxième un Ψ-terme. L'effet (classique) est de donner à la variable la valeur du Ψ-terme. La notation et le comportement sont ceux d'une procédure (il ne rend aucune valeur).

Exemples :

```
AFFECT(1.gnr, mas)
AFFECT(1.syn.sujet, UNIF(1.syn.sujet, 2))
```

Sa sémantique est particulière si la variable n'est pas définie, c'est-à-dire si le chemin associé n'existe pas. Dans ce cas le chemin est crée et sa valeur est celle du Ψ-terme.

Exemples :

Si la variable 1 a pour valeur :

```
UL(syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal;
                 sém => @S : ANIME)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => @S;patient => @O)))

AFFECT(1.syn.sujet, UNIF(1.syn.sujet, UL(sém => PERSONNE))
```

donne à 1 la valeur :

```
UL(syn => proposition(
     prédicat => MANGER;
```

[1]Tous les constructeurs d'expressions (opérateurs et actions) sont écrits en notation fonctionnelle.

```
      sujet => UL(syn => groupe_nominal;
                  sém => @S : PERSONNE)
      objet => UL(syn => groupe_nominal;
                  sém => @O : COMESTIBLE))
   sém => MANGER(agent => @S;patient => @O)))
```

sous réserve que `UNIF(ANIME, PERSONNE) = PERSONNE`. **On notera que l'affectation préserve les relations de coréférences (dans la mesure où les attributs sont conservés). Ainsi ci-dessus on retrouve la relation** `(syn.sujet.sém, sém.agent)` **malgré l'affectation d'une valeur à** `syn.sujet` **car le trait** `sém` **apparaît aussi dans la nouvelle valeur.**

`AFFECT(1.CAT, verb)` **donne à 1 la valeur :**

```
UL(syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal;
                 sém => @S : PERSONNE)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => PERSONNE;patient => @O));
   CAT => verb)
```

Bien que la sémantique de AFFECT permette l'ajout de traits, nous avons défini deux opérateurs plus "naturels" : PLUS chargé de cette opération, et MOINS chargé de la suppression d'un trait. Ce sont deux opérateurs unaires dont le paramètre est une variable et dont la sémantique peut être définie comme suit :

- **PLUS(v) équivaut à AFFECT(v, _TOUT_) si v n'est pas définie et ne fait rien sinon.**
- **MOINS(v) efface le dernier attribut de v.**

Exemple :

Si la variable 1 a pour valeur :

```
UL(syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal;
                 sém => @S : PERSONNE)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => @S;patient => @O)))
```

MOINS(1.syn.sujet) donne à 1 la valeur :

```
UL(syn => proposition(
     prédicat => MANGER;
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => PERSONNE;patient => @O)))
```

On notera que la suppression efface la valeur d'un trait en préservant les éléments partagés (`PERSONNE` dans l'exemple ci-desssus, partagé avec le trait `agent` du trait `sém`).

Nous sommes conscient du caractère limité de ce langage et de la nécessité de lui adjoindre des fonctionnalités du type :

- **traitement de chaînes de caractères afin de pouvoir manipuler la forme externe des unités lexicales ;**

- traitement d'entiers pour pouvoir, par exemple, évaluer la longueur d'une forêt ou affecter à un trait une valeur entière (le trait nombre associé à "trois" serait 3) ;
- possibilité d'utiliser des registres de travail pour stocker des résultats temporaires ou pour définir des constantes nommées de portée globale.

Il est probablement nécessaire aussi d'augmenter les possibilités de manipulation sur les Ψ-termes (test d'infériorité, d'égalité, échange de valeurs,...) mais comme ce langage n'a pas une ambition générale mais une ambition de langage spécialisé pour l'écriture de grammaires, la définition de ces ajouts suppose la collaboration d'un linguiste qui précise quelles sont les fonctionnalités indispensables.

4

Réalisation

Nous décrivons dans ce chapitre le prototype (baptisé CADET pour Constructeur d'Arbres de DEpendances Typés) que nous avons réalisé. Il s'agit d'un poste de développement de grammaires de dépendances dont le noyau est une mise en œuvre du transducteur présenté au chapitre 2 intégrant les Ψ-termes présentés au chapitre 3. Il permet l'écriture de grammaires et de corpus de textes, il est écrit entièrement en Pascal et fonctionne sur tout micro-ordinateur compatible PC.

4.1. Architecture

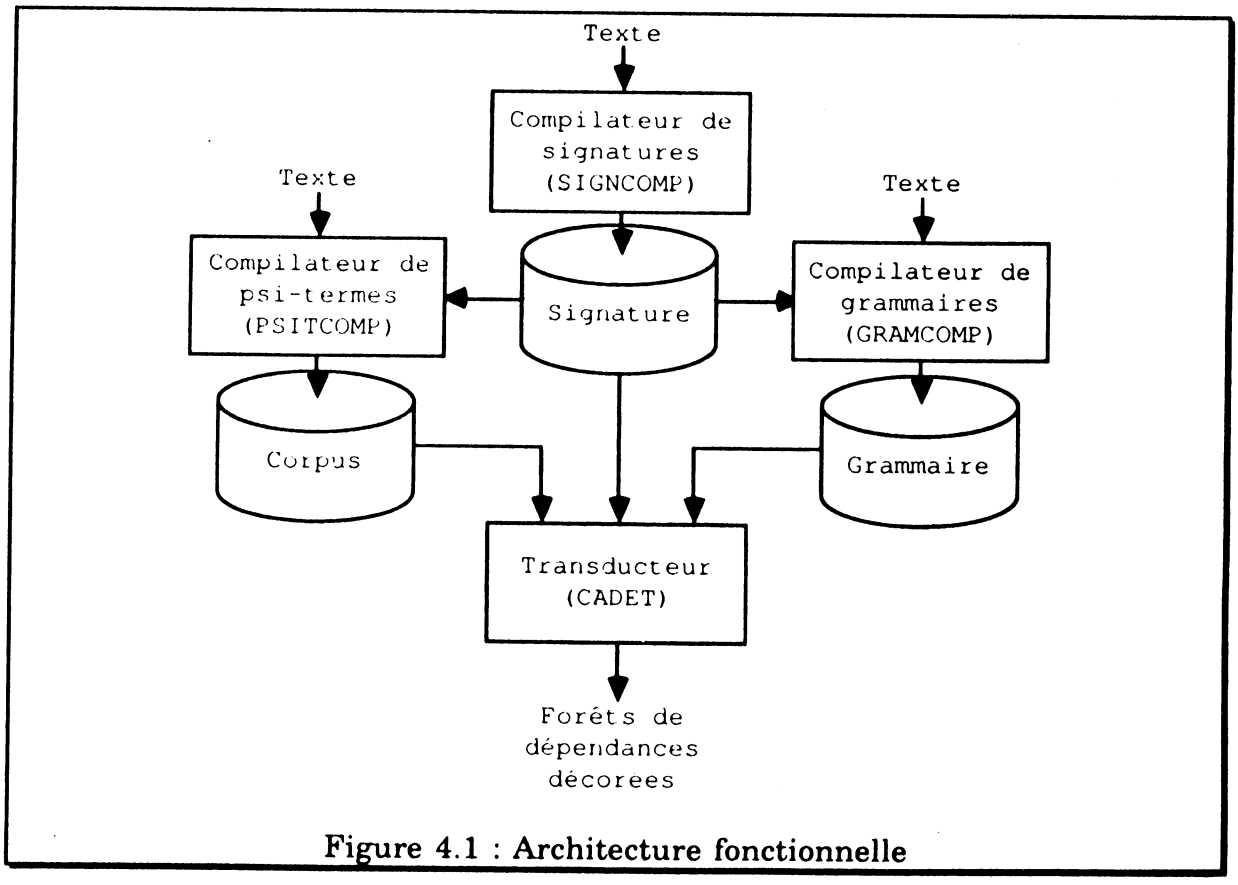


Figure 4.1 : Architecture fonctionnelle

Nous avons adopté comme principe de base que toutes les données fondamentales : signatures, Ψ-termes, grammaires, seraient créées et stockées sous forme de texte. Cela présente de nombreux avantages :

- la création, la modification et l'impression peuvent être réalisées avec les outils standard du système hôte (éditeur, imprimeur). Côté utilisateur, on évite l'apprentissage d'un éditeur ad hoc, côté programmeur, on évite la mise en œuvre de cet éditeur. Et ce n'est pas seulement un gain de temps mais aussi un gain de portabilité, car si tous les systèmes fournissent des outils de manipulation de textes, l'écriture d'un éditeur moderne (plein écran) exploite les entrées/sorties de bas niveau (propres à un système) et son portage d'une machine à l'autre s'en trouve limité.
- pour peu que la syntaxe le permette, l'utilisateur peut commenter à loisir les signatures ou grammaires qu'il écrit.

Cette technique présente bien sûr l'inconvénient d'obliger à une re-compilation des données après chaque modification, ce qui peut être fastidieux sur de grosses applications. Dans notre cas, on peut envisager de développer les grammaires par morceaux suffisamment petits pour que les temps de compilation soient supportables et l'inconvénient ci-dessus n'intervient qu'en fin de développement.

L'architecture fonctionnelle du système, présentée figure 4.1, est une conséquence immédiate de ce choix.

Comme l'analyseur morphologique dont nous disposons ne permet pas actuellement d'obtenir des Ψ-termes, nous lui avons substitué un compilateur de Ψ-termes qui permet de créer des jeux de phrases tests stockées dans un corpus (voir §4.3).

Pour faciliter l'utilisation des divers programmes, nous avons réalisé une petite interface basée sur un système de menus. Elle utilise un environnement de travail qui peut être modifié et grâce auquel on peut par simple appui sur une touche : éditer ou compiler des données ou lancer une analyse

Pour le développement proprement dit nous avons fait porter nos efforts sur la modularité, l'encapsulation et la généricité. La figure 4.2 donne l'architecture logicielle du système : les programmes correspondant aux carrés de la figure 4.1 sont notés en majuscules tandis que les minuscules sont réservées aux modules. Les liens dénotent les dépendances fonctionnelles : ainsi le programme `SIGNCOMP` utilise les modules `analex` et `signature` qui eux-mêmes utilisent les modules `listes` et `symboles`. Chacun de ces modules contient la définition d'une structure de données précise et toutes les primitives qui permettent de la manipuler (typiquement : insertion, modification, suppression, chargement, sauvegarde).

Nous donnons ci-dessous une description succincte de ces modules et nous détaillons un peu plus les programmes dans les paragraphes suivants.

Listes : Gestion de listes dynamiques quelconques (insertion, suppression, copie, longueur).

Symboles : Tables de symboles gérées par hash-code.

Analex : Analyse lexicale (lecture de symboles, de listes et d'ensembles de symboles, gestion d'erreurs,...). C'est également ce module qui se charge des commentaires : tout le texte compris entre deux tirets (--) et une fin de ligne.

Signature : Structure interne d'une signature et procédure d'unification de deux types atomiques. La version actuellement mise en œuvre utilise un codage par liste de successeurs immédiats.

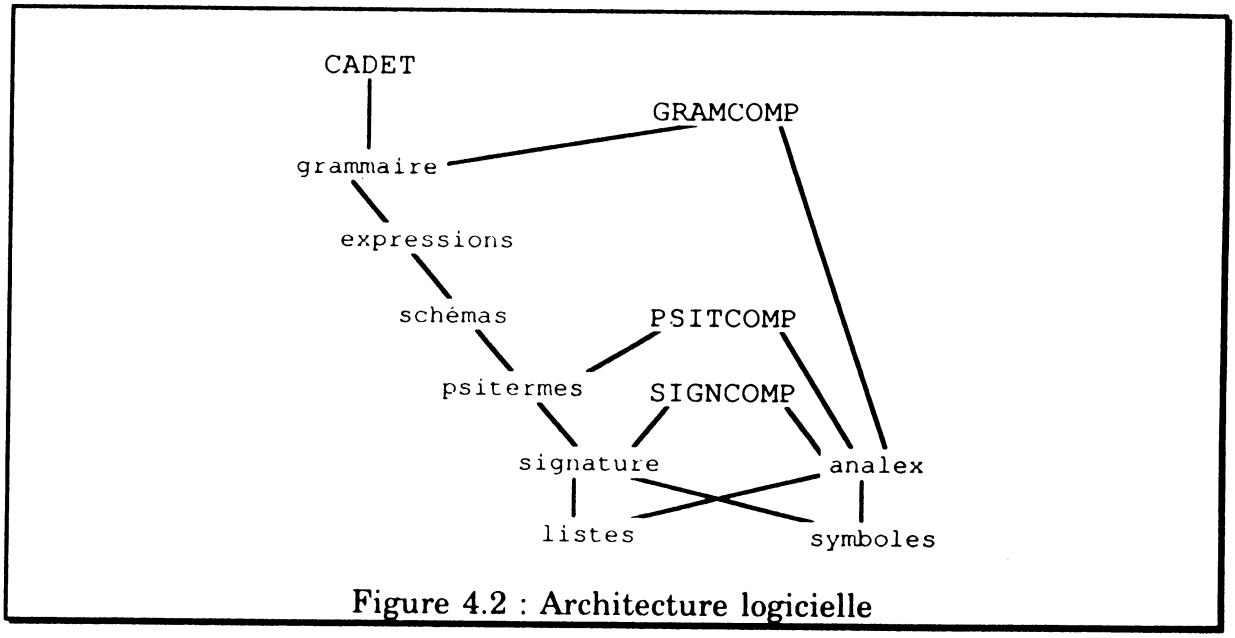


Figure 4.2 : Architecture logicielle

Psitermes : Ψ-termes et procédures d'unification et de reconstruction (cf §3.1.3). Les Ψ-termes sont codés dans des registres spécialisés (tables), la version actuelle en compte 3 : un pour les unités lexicales et les arbres, un pour la grammaire et un registre de travail pour l'unification.

Schémas : Structure interne des forêts de dépendances et des schémas de forêt de dépendances ainsi que l'instanciation et la construction de l'arbre résultat (cf §2.2.2 et §2.2.3).

Expressions : Structure interne des expressions (cf §3.2.2) et des routines d'évaluation. Cette structure est une table alternant-successeur.

Grammaire : Structure interne des règles et les moyens d'y accéder.

4.2. Signature

La définition d'une signature suppose l'établissement d'une hiérarchie de symboles. Nous avons autorisé la définition d'une relation d'infériorité entre des symboles ou des ensembles de symboles. Nous donnons ci-dessous la syntaxe reconnue par le compilateur.

Syntaxe :

```
<Signature> ::= <ListeArcs>
<ListeArcs> ::= <Arc> [ <ListeArcs]
<Arc> ::= <EnsSymbole> [ < <Arc> ]
<EnsSymbole> ::= <Symbole> | { <ListeSymboles> }
<ListeSymboles> ::= <Symbole> [ , <ListeSymboles>
```

Un symbole est un identificateur au sens des langages de programmation classique. Nous ne distinguons pas les majuscules des minuscules mais nous avons autorisé les lettres minuscules accentuées. Le format est très libre et les commentaires peuvent apparaître partout où peut apparaître un espace.

Exemples :

```
personne < humain      -- commentaire
{humain,               -- commentaire dans un ensemble
 animal } < animé
ppas < {Adj, Verbe} < CLS
voiture                -- un symbole seul ( < _TOUT_)
```

Le compilateur construit un graphe orienté du plus général au plus particulier. Ce graphe est ensuite rendu connexe par ajout d'un lien de _TOUT_ vers tous les éléments maximaux. On vérifie ensuite que le graphe ne comporte pas de cycles et on le rend minimal en éliminant tous les arcs qui peuvent se déduire par transitivité. On conserve pour chaque symbole défini la liste de ses successeurs immédiats.

Une évolution indispensable consiste à remplacer cette liste par un code plus compact en utilisant une technique de codage telle que celles définies dans [AIT-KACI 89b].

4.3. Corpus

La représentation d'une phrase après analyse morphologique est une liste de Ψ-termes où chacun porte les informations associées à un segment de la phrase. Le compilateur de Ψ-termes utilise la syntaxe donnée au chapitre 3 avec quelques particularités :

- il construit un seul Ψ-terme qui regroupe les éléments de la liste sous une disjonction ;
- les homographes sont écrits dans la forme externe comme des disjonctions et sont codés tels quels.

Le Ψ-terme résultant de la compilation est donc une disjonction pouvant contenir des disjonctions. C'est la procédure NouveauMot (définie au §2.2.6) qui sera chargée d'extraire le Ψ-terme correspondant à un segment de la phrase et éventuellement de l'éclater si celui ci correspond à une homographie.

Exemple :

```
-- "la belle ferme", catégories uniquement
{ UL(cat => Det) ; UL(cat => pron) }      -- "la"
{ UL(cat => adjq) ; UL(cat => subc) }     -- "belle"
{ UL(cat => adv) ; UL(cat => adjq) ;      -- "ferme"
  UL(cat => subc) ; UL(cat => verb) }
```

4.4. Grammaire

Nous donnons ci dessous la syntaxe complète de la grammaire, reprenant la partie géométrique donnée au chapitre 2 en lui adjoignant une liste de conditions dans la partie gauche et une liste d'actions dans la partie droite.

Syntaxe :

```
<Règle> ::= <Symbole> [ <Gauche> => <Droite> ]
<Droite> ::= <Forêt> [ ; <ListeActions> ]
<Gauche> ::= ( <LSchema> ) / [ <ListeConditions> ] /
<LSchema> ::= <Schema> [ , <LSchema> ]
<Schema> ::= <Arbre> [ <Limite> ] | <VarForêt>
<Arbre> ::= [ <Forêt> ] <Noeud> [ <Forêt> ]
<Forêt> ::= ( ) | ( <LssArbre> )
<LssArbre> ::= <ssArbre> [ , <LssArbre> ]
<ssArbre> ::= <Arbre> | <VarForêt>
<VarForêt> ::= $<Symbole> [ : <EnsCat> ] | 0
<Noeud> ::= <Entier> [ : <EnsCat> ]
<EnsCat> ::= { <LCateg> }
<LCateg> ::= <Symbole> [ ; <LCateg> ]
<Limite> ::= ? <EnsCat> | / <EnsCat>
<ListeConditions> ::= <Condition> [ , <ListeConditions> ]
<Condition> ::=
      <Var> |
      NON( <Condition> ) |
      ET( <Condition> , <ListeConditions> ) |
      OU( <Condition> , <ListeConditions> ) |
      UNIF( <Expression> , <Expression> )
<ListeActions> ::= <Action> [ ; <ListeActions> ]
<Action> ::=
      PLUS( <Var> ) |
      MOINS( <Var> ) |
      AFFECT( <Var> , <Expression> ) |
<Expression> ::=
      <Const> |
      <Var> |
      UNIF( <Expression> , <Expression> )
<Const> ::= <Psiterme>
<Var> ::= <Entier> [ .<ListeLabels> ]
<ListeLabels> ::= <Label> [ .<ListeLabels> ]
<Label> ::= un symbole interprété comme un nom d'attribut.
```

En cas d'erreur dans la compilation d'une règle, on essaie de se rattraper sur le prochain crochet fermant. Cela permet dans la plupart des cas de procéder à la vérification de toutes les règles même si une erreur apparaît sur la première.

Une règle s'applique à une forêt si cette forêt est une instance du schéma de la partie gauche de la règle et si dans ce cas toutes les conditions de la liste sont vérifiées. En fait cette liste aurait pu être remplacée par une seule condition, sous la forme d'un ET regroupant ses éléments.

Si une règle s'applique, alors toutes les actions de la liste de la partie droite sont exécutées, il n'y a pas d'impossibilités puisqu'on a donné aux opérateurs une sémantique pour tous les cas (voir AFFECT au §3.2.2.4).

4.5. Transducteur

C'est une mise en œuvre directe du transducteur présenté au chapitre 2 ; il construit une liste de forêts (pile d'arbres) qui sont affichées à l'écran. Nous avons tenté de rendre le format de sortie le plus lisible possible, compte tenu des objectifs fixés (obtenir un poste de développement). Nous avons donc défini et réalisé une procédure d'affichage d'arbres (alternants-successeurs avec une distinction des successeurs droits et gauches) sous forme semi-graphique.

Nous avons défini deux modes de fonctionnement pour le transducteur :

- un mode TRACE dans lequel il affiche la progression de l'analyse : état de la pile après introduction d'un nouveau mot, arbre avant et après l'application d'une règle et nom de cette règle,...
- un mode NORMAL qui n'affiche que les résultats valides, c'est-à-dire les piles d'arbres ne contenant qu'un seul arbre après que tous les mots ont été introduits et les règles appliquées.

4.6. Exemple d'analyse

On trouvera dans l'annexe C la trace du transducteur pour la phrase : "les trois ou quatre personnes que je connais à Paris travaillent à l'école de commerce". La signature et la grammaire utilisées sont données dans l'annexe B. A cette phrase on associe la liste de Ψ-termes :

```
{ UL(cat => Def) ;          -- les
  UL(cat => PPer ; fonc => cod) }
UL(cat => ACard)            -- trois
UL(cat => CoCoS)            -- ou
UL(cat => ACard)            -- quatre
UL(cat => subc ;            -- personnes
   sem => HUMAIN)
UL(cat => CoSub)            -- que
UL(cat => PPer ;            -- je
   fonc => suj ;
   sem => ANIME)
UL(cat => VAction ;         -- connais
   syn => PP(sujet => GN(sem => @S:ANIME) ;
             objet => GN(sem => @O:{ANIME ; OBJET})) ;
   sem => SAVOIR(détenteur => @S ; objet => @O))
UL(cat => Prep)             -- à
UL(cat => subp)             -- Paris
UL(cat => VAction ;         -- travaillent
   syn => PP(sujet => GN(sem => @S:ANIME)) ;
   sem => ACTION(agent => @S))
UL(cat => Prep)             -- à
{ UL(cat => Def) ;          -- l'
  UL(cat => PPer ; fonc => cod) }
UL(cat => subc ;            -- école
   sem => LIEU)
UL(cat => Prep)             -- de
UL(cat => subc)             -- commerce
```

Conclusion de la première partie

Nous avons obtenu un premier prototype d'un transducteur d'arbres qui permet la construction d'analyseurs syntaxico-sémantiques du français. Notre ambition n'était pas d'obtenir un système général capable d'accepter n'importe quelle phrase de la langue mais plutôt un outil adaptable permettant de construire des analyseurs de manière simple et rapide pour des domaines limités. L'utilisation d'un langage très opératoire et d'une hiérarchie de catégories nous a permis d'atteindre ces objectifs.

L'utilisation typique de CADET est la construction d'un analyseur à partir d'un corpus : on détermine un ensemble de catégories élémentaires très précises en fonction du vocabulaire et on utilise ces catégories pour la construction du lexique. On regroupe ensuite ces catégories en ensembles plus généraux en fonction des structures qu'on veut obtenir. On écrit enfin les règles d'analyse de manière incrémentale : d'abord des règles générales qu'on affine ensuite après les avoir testées sur le corpus. On précise dans les règles les manipulations sémantiques qu'on veut effectuer en fonction de l'application envisagée.

Mais l'adaptabilité de CADET à une large gamme d'applications suppose des améliorations, qui, sans remettre en cause les fondements, obligent à un travail important de programmation.

Les limites les plus importantes concernent le langage de représentation de la connaissance. Sans prétendre obtenir un langage de programmation général, il est quand même indispensable de pouvoir manipuler autre chose que des symboles et des Ψ-termes. Il faudra donc intégrer au langage :

- le traitement de chaînes de caractères afin de pouvoir au moins manipuler la forme externe des unités lexicales ;
- le traitement de nombres (au moins les entiers) pour pouvoir, par exemple, évaluer la longueur d'une forêt ou affecter à un trait une valeur entière ;
- la possibilité de définir des objets qu'on peut ensuite réutiliser : des registres de travail pour stocker des résultats temporaires ou des valeurs de portée globale. On peut aussi envisager de donner la possiblité à l'utilisateur de décrire ces propres primitives de traitment sur la base des primitives prédéfinies ; cela impose la définition d'un minimum de structures de contrôle.

Il est également nécessaire d'augmenter les possibilités de manipulation sur les Ψ-termes eux-mêmes (test d'infériorité, d'égalité, échange de valeurs,...) et sur les types de la signature (calcul de bornes supérieures, de

complémentaires,...). Enfin il paraît indispensable de pouvoir manipuler des Ψ-termes disjonctifs pour représenter une partie au moins des ambiguïtés.

Une méthode très simple pour intégrer ces extensions serait de construire le transducteur sur un langage de programmation extensible comme Lisp, on profite ainsi des objets du langage sans effort supplémentaire de programmation (traitement de chaînes, de nombres, structures de contrôle, définition de primitives).

Concernant le transducteur lui-même, il est nécessaire d'améliorer le contrôle : c'est-à-dire de donner à l'écrivain de grammaire des moyens plus importants pour délimiter la portée d'une règle. Nous avions choisi dans un premier temps de faire simple, mais l'écriture de grammaires importantes pose des problèmes de cohérence : il faut s'assurer, par exemple, que deux règles différentes ne s'appliquent pas à la même forêt en fournissant le même résultat. Sans atteindre la complexité de mécanismes de contrôle tels que celui de Robra (§1.2.2), on peut imaginer de regrouper, par exemple, les règles en paquets tels qu'une seule des règles d'un paquet pourrait s'appliquer à une forêt donnée. On pourrait aussi envisager un mécanisme de validations et de saturations similaire à celui mis en oeuvre pour l'analyse morphologique de PILAF (voir [COURTIN 77] ou le chapitre 5, §5.1.2). Enfin, on pourrait aussi augmenter les règles elles-mêmes en donnant la possibilité d'écrire des parties droites complexes ; on aurait des règles du type :

PartieGauche
$\quad$ $Condition_1$ => $Arbre_1$, $Action_1$
$\quad$ $Condition_2$ => $Arbre_2$, $Action_2$
$\quad$ .....
$\quad$ $Condition_n$ => $Arbre_n$, $Action_n$

La partie gauche, sous forme d'un schéma de forêt, permettrait la sélection de la règle. Les conditions, portant sur les traits des nœuds de l'instance, permettraient de sélectionner un arbre résultat et la liste d'actions correspondante. Pour choisir une de ces méthodes, ou un mélange de toutes, il est indispensable d'utiliser le prototype actuel pour écrire des grammaires importantes afin de se rendre compte des besoins réels.

Pour passer du stade de prototype au stade d'un vrai système d'analyse du français, il faut donc dans un premier temps faire un travail de linguiste permettant, sur des exemples concrets de traitement de la langue, de préciser quelles sont les lacunes principales du système. Dans un deuxième temps, il faut déterminer les solutions permettant de combler ces lacunes et les mettre en oeuvre.

Partie II

Vers un système complet de détection/correction

5

Le niveau lexical

Les erreurs apparaissant dans un texte écrit peuvent être classées par niveau en considérant à quel niveau de l'analyse d'une phrase l'erreur sera détectée :

- au niveau lexical : relèvent de ce niveau toutes les erreurs qui conduisent à la non-reconnaissance d'un mot ("voture", "bin", "shmurfle").
- le niveau syntaxique concerne les erreurs d'accord, de construction de la phrase, d'ordre des mots ("la chat", "Je à Paris vais").
- le niveau sémantique concerne l'emploi d'un mot ou d'un groupe dont le sens n'est pas en accord avec le reste de la phrase ("le mètre fait travailler les élèves").

On pourrait ajouter à ces trois niveaux le niveau pragmatique (ou logique) qui concerne les erreurs liées à l'emploi du langage dans un certain contexte. "Prends ce sac et amène le moi" peut être considéré comme incorrect si on sait que le sac pèse plus de 200 kilos.

La détection et la correction des erreurs ont fait l'objet de nombreux travaux qui ont donné lieu à d'autres classifications.

[CATACH 80] distingue six catégories d'erreurs :

- *phonétique* : qui donne une phonétique approximative du mot, comme "maitenant" pour "maintenant" ou "arbrustre" pour "arbuste".
- *phonogrammiques* : mauvaise transcription écrite des sons, comme "cheuveu" ou "ceuillir".
- *morphogrammiques* : erreurs syntaxiques ("sac de bille", "des ombres passes") ou de mauvaise transcription des sons mais au niveau des groupes ("un névier") ou des désinences ("nous vogons").
- erreurs concernant les *homophones* ("l'arme" pour "larme" ou "encore sage" pour "en corsage").
- erreurs concernant les *idéogrammes* : mauvais emploi des majuscules ou des ponctuations.
- erreurs concernant les *lettres non fonctionnelles* ("sculter", "téatre", "ocuper").

Cette typologie des erreurs a été réalisée dans le cadre de l'enseignement de l'orthographe, elle met donc l'accent sur la transcription de l'oral vers l'écrit.

[VERONIS 88b, 88c] propose quand à lui une classification plus simple en erreurs de *compétence* et erreurs de *performance*. La compétence concerne la

connaissance de la langue alors que la performance concerne son utilisation dans une situation réelle. Véronis se place dans la situation d'un dialogue homme-ordinateur et fait donc intervenir les erreurs des deux interlocuteurs, il distingue donc les erreurs :

- de compétence de l'utilisateur : un mot qu'il ne sait pas écrire correctement ("hortodocse", "infractus").
- de performance de l'utilisateur : erreurs de frappe ("semgent" pour "segment").
- de compétence du système : l'utilisateur emploie un mot correct mais qui n'est pas dans le lexique.
- de performance du système : défaillance du matériel telle que mauvais fonctionnement du clavier ; ce type d'erreurs peut être négligé compte tenu de la fiabilité actuelle du matériel.

Nous n'avons donné ici que des exemples relevant du niveau lexical mais Véronis applique cette classification aux erreurs syntaxiques et sémantiques.

Dans le cadre du système VORTEX [LAHENS 86, PERENNOU 86], une classification est proposée pour les erreurs de niveau lexical :

- erreurs *typographiques* : ce sont toutes les erreurs introduites à la saisie des textes. Elles peuvent être liées à l'utilisation d'un clavier mais aussi à d'autres modes de saisie (lecture optique), ou à la transmission télématique. Elles donnent lieu dans VORTEX à des méthodes de correction particulières.
- erreurs *orthographiques* : ce sont les fautes d'usage et tout type de faute qui ne relève pas d'une erreur typographique, elles sont corrigées plutôt par transcription phonétique.

Notre but étant la construction d'un système informatique de détection et de correction des erreurs, nous avons adopté une classification en fonction de la technique capable de proposer des corrections. Nous distinguons trois catégories :

- les erreurs typographiques : omission, insertion, substitution d'une lettre ou transposition de deux lettres (voir §5.2.1).
- les erreurs phonétiques : le mot n'est pas correct mais sa transcription phonétique l'est (voir §5.2.2).
- les erreurs de désinence : la disponibilité d'un générateur morphologique permet de corriger les erreurs liées à la mauvaise formation d'un pluriel ("chevals" pour "chevaux"), ou à la mauvaise conjugaison d'un verbe ("allerons" pour "irons") (voir §5.2.3).

Nous présentons ces différentes techniques au §5.2 après avoir décrit le module de détection qui n'est autre qu'un analyseur morphologique.

5.1. Analyse morphologique

L'analyseur morphologique dont nous disposons est celui du système PILAF (Procédures Interactives Linguistiques Appliquées au Français). PILAF [COURTIN 76, 77, 86] est conçu comme une boîte à outils logiciels comprenant :

- une base lexicale constituée d'un dictionnaire de bases et d'un dictionnaire de suffixes. Cette base est gérée par un programme

interactif permettant à tout moment l'insertion, la suppression ou la modification d'une entrée.
- une base de règles et de modèles morphologiques accompagnée également d'un programme de gestion.
- un analyseur et un générateur morphologique dont le noyau est un transducteur général d'états finis qui s'appuie sur les deux bases ci-dessus.
- un analyseur syntaxique : constructeur de structures de dépendances (décrit au §1.4.2).
- un ensemble d'utilitaires d'édition, de génération et de statistique appliqués au deux bases.

La principale caractéristique de ce système est qu'il est entièrement paramétré et que tous ses paramètres sont modifiables grâce à des éditeurs interactifs. Le transducteur a d'ailleurs été utilisé pour des applications très diverses : transcription graphique/phonétique, traduction d'un langage de programmation dans un autre, analyse et génération morphologique du français et plus récemment du portugais [COURTIN 89b].

Nous présentons ici de manière succincte le fonctionnement de l'analyseur morphologique et la structure de la base lexicale.

Le but de tout système d'analyse morphologique est de segmenter une suite de caractères en une suite de chaînes en calculant le maximum d'informations linguistiques sur ces chaînes. Il s'agit donc d'effectuer une transduction d'une chaîne de caractères vers une chaîne d'unités lexicales : des mots auxquels sont associées des propriétés linguistiques.

A chaque unité lexicale correspond un segment unique de la chaîne d'entrée et la version actuelle du transducteur suppose de plus que les segments obtenus sont insegmentables. Pour une phrase donnée, on obtient donc un nombre unique de segments. Cette restriction est imposée par l'analyseur syntaxique de PILAF qui doit impérativement connaître le nombre exact de segments de la phrase. On ne peut envisager de calculer toutes les segmentations et de les fournir à l'analyseur syntaxique sans porter atteinte à l'efficacité globale de l'analyse. On ne calcule donc qu'une segmentation en essayant de la rendre la plus cohérente possible avec le modèle syntaxique. On verra au §5.3.3 que cette contrainte n'a plus de raison d'être avec le constructeur de structures de dépendances présenté au chapitre 2.

Exemples de décompositions :

"Il mange de la soupe." se décompose en
"Il" + "mange" + "de" + "la" + "soupe" + "."
"Il y a à ce propos une controverse." se décompose en
"Il y a" + "à ce propos" + "une" + "controverse" + "."

On notera que le blanc n'est pas un séparateur, ce qui permet de considérer les locutions "il y a" et "à ce propos" comme des unités lexicales entières.

En français, chaque segment peut être composé de : préfixe(s), base ou racine, suffixe(s) et désinence.

Exemple :

"retrouvera " se décompose en :
préfixe : "re"

base : "trouv"
suffixe : "er"
désinence : "a "
Les parties suffixe, préfixe et désinence peuvent être vides.

La version actuelle (dictionnaires, règles et modèles) ne traite que la morphologie flexionnelle : nous n'avons pas distingué les préfixes et nous nous sommes limités aux suffixes et désinences liés aux dérivations de nombre, de genre et de conjugaison. Nous n'avons pas par exemple introduit les dérivations nominales ("réaliser" → "réalisation") ou adjectivales ("réaliser" → "réalisable") du verbe. Le langage du transducteur permet l'expression de telles dérivations mais elles font l'objet de tellement d'exceptions et de cas particuliers que nous avons préféré les ignorer afin de limiter le nombre de règles et de modèles morphologiques.

Un segment S de la chaîne d'entrée est reconnu comme une unité lexicale si on peut trouver une *décomposition valide* de S. La décomposition est obtenue par identification d'une partie contiguë de S avec un élément du dictionnaire. La validité est garantie par la grammaire qui s'assure que la concaténation des divers éléments de la décomposition est acceptable.

La transduction est une interprétation du segment S obtenu : les informations linguistiques associées à l'unité lexicale résultante sont calculées par les règles de la grammaire sur la base des informations portées par les différents éléments de la décomposition (base, suffixe, désinence).
Exemple :

"couvent " → base /couvent/ : substantif commun masculin
désinence / / : singulier

"couvent " → base /couv/ : verbe 1er groupe
désinence /ent / :
3ème pers. pluriel présent indicatif
ou 3ème pers. pluriel présent subjonctif

On notera que la décomposition d'un segment n'est pas unique mais peut donner lieu à plusieurs interprétations d'une même unité lexicale. On appelle de tels mots des mots *homographes*.

5.1.1. Le dictionnaire (ou base lexicale)

La structure du dictionnaire reflète une caractéristique de notre analyseur : ne pas considérer le blanc comme un séparateur. Pour déterminer la segmentation d'une phrase, on doit donc procéder par essais successifs et la structure du dictionnaire doit permettre de limiter leur nombre.

5.1.1.1. Organisation

Le dictionnaire comprend deux parties : un dictionnaire de bases (ou racines) et un dictionnaire de suffixes et désinences. Cette distinction, qui n'existait pas dans les premières versions du système, a été introduite pour des raisons d'efficacité : elle a permis de diviser par un facteur 2 le temps nécessaire à l'analyse d'une phrase.

Les entrées du dictionnaire de bases sont réparties dans des fichiers dont la taille est bornée. L'accès à un fichier se fait à travers une table d'index appelée *table maître*. Pour chaque fichier, la table maître contient le plus petit et le plus grand élément, ainsi que le nom du fichier. Tous les fichiers (ainsi que la table maître) sont triés en ordre alphabétique inverse. Cette organisation est une version simplifiée de la technique des B-arbres : quand un fichier arrive à saturation (après un certain nombre d'insertions), la moitié de ses éléments est transférée dans un nouveau fichier et une nouvelle entrée est ajoutée à la table maître. Cette dernière est chargée en mémoire, et l'accès dans un fichier est réalisé par dichotomie ; le nombre d'accès disque pour retrouver un élément quelconque est donc inférieur au logarithme à base 2 du nombre maximum d'éléments d'un fichier. Cette organisation permet donc un accès rapide tout en conservant la possibilité d'un accès séquentiel (qui est indispensable comme on le verra plus loin).

Le dictionnaire des suffixes et désinences est de taille réduite et est donc stocké en mémoire sous forme d'une table (accès par dichotomie).

Les entrées des dictionnaires sont des enregistrements contenant 6 champs :

- *lettres* : chaîne de caractères (base, suffixe ou désinence)
- *occlettres* : numéro d'occurrence de la chaîne dans le dictionnaire
- *modèle* : numéro de modèle
- *indécoupable* : booléen signalant que l'élément est non-réductible
- *chcirc* : chaîne de caractères (élément suivant dans le chaînage circulaire)
- *occlrc* : numéro d'occurrence de l'élément suivant

Remarques :

Les champs occurrences distinguent deux éléments identiques sur *lettres*. La paire (*lettres, occlettres*) constitue donc une clé d'accès.

Le chaînage circulaire est utilisé par la génération morphologique, pour engendrer par exemple toutes les formes d'un verbe.

Exemple : Les différentes bases du verbe "aller" sont chaînées circulairement :

/all/ → /aill/ → /ir/ → /va / → /vas / → /vais / → /vont / → /all/

La valeur vrai de l'indicateur *indécoupable* permet d'arrêter la décomposition du segment courant de la chaîne d'entrée.

5.1.1.2. Utilisation

Comme on l'a déjà dit, on ne dispose pas de séparateur d'unités lexicales, l'entrée de l'algorithme est donc un simple pointeur sur le premier caractère de la chaîne restant à analyser. On a la possibilité, bien entendu, de lire tout ou partie de la chaîne restante (droit de regard en avant).

L'algorithme procède en deux étapes :

1 - Déterminer le fichier du dictionnaire de bases dans lequel on a une chance de trouver la base du mot qui commence au caractère courant. Pour cela, on consulte simplement la table maître en

utilisant comme clé K la chaîne constituée des n premiers caractères de la chaîne restante (n est arbitrairement fixé à 10).

2- Trouver la plus longue base de ce fichier qui soit un préfixe de la chaîne restante. On effectue une recherche dichotomique de la première base inférieure ou égale à K. L'ordre alphabétique inverse assure que cette base est celle qui est le plus long préfixe de la chaîne restante (dans le cas contraire la chaîne contient une erreur).

On a alors une première base possible pour la chaîne restante. La base et son modèle sont alors transmis au transducteur qui va ensuite piloter les accès en fonction des résultats obtenus. On a le schéma d'algorithme suivant :

Si la base est à elle seule une unité lexicale complète (cette information est contenue dans le modèle) ; c'est le cas de tous les mots invariables : adverbes ("cependant ", "ainsi "), prépositions ("en ", "par"), etc... alors
    Stocker ce résultat.
Sinon
    Lancer l'analyse sur la suite de la base dans la chaîne restante mais en limitant les accès au seul fichier des suffixes et désinences.
    Si cette analyse fournit un résultat, valider la concaténation de ce résultat et de la base courante.
Si l'indicateur *indécoupable* est faux alors
    chercher dans le fichier initial (par simple accès séquentiel), la prochaine base préfixe de la précédente et recommencer la transduction avec elle.
L'algorithme s'arrête quand tous les découpages possibles d'un même segment ont été obtenus.

Exemple :

Dictionnaire : (on ne mentionne que les champs *lettres* et *indécoupable*)

/s /vrai	/allée/faux
/ées /vrai	/all/vrai

Chaîne restante : "allées "

La première base obtenue : /allée/ n'est pas un segment possible, on relance une recherche avec le pointeur d'entrée positionné immédiatement après "allée", sur le "s". On obtient alors la désinence /s / et le segment est alors bien déterminé :

"allées " → base /allée/ substantif commun féminin
désinence /s / pluriel

/s / est indécoupable, on arrête donc le traitement et on revient à /allée/ qui n'est pas indécoupable, on cherche donc une base plus courte : ici on obtient /all/. Le transducteur se charge de la reconnaissance de /ées /, on obtient donc :

"allées " → base /all/ verbe 1er groupe
désinence /ées / participe passé féminin pluriel

Remarques :

- Pour reconnaître des locutions telles que "à ce propos " comme des unités lexicales entières, il suffit d'avoir l'entrée correspondante dans le dictionnaire.

- Le résultat de l'analyse est une seule segmentation de la chaîne d'entrée mais pour chaque segment on peut avoir diverses interprétations.

5.1.2. Le transducteur

5.1.2.1. Les règles

Le modèle théorique est celui des *grammaires à validations et saturations* (GVS) [COURTIN 77]. Ce modèle est équivalent formellement aux grammaires d'états finis mais permet une formulation plus concise.
Une GVS est définie par un quadruplet (V, S, P, E) où :

V vocabulaire terminal

E ensemble des noms de règles, $E \subset [1..n]$

S symbole initial, élément de $\mathcal{P}(E) \times \mathcal{P}(E)$

P ensemble des productions (règles). P est une application de E dans $V \times \mathcal{P}(E) \times \mathcal{P}(E)$, un élément de P est noté :

$e \rightarrow v[e_1 \ldots e_p][f_1 \ldots f_q]$, $e, e_i, f_j \in E$, $v \in V$

Les e_i sont les *validations* et définissent les règles applicables après application de e. Les f_j sont les *saturations* et définissent les règles interdites après application de e.

Exemple de GVS :

GVS = ({a,b,c,d}, [1, 2][], P, {1,2,3,4})

avec P = { 1 → a[1,2,3][], 2 → b[2,3][1], 3 → c[1,3,4][2], 4 → d[][] }

Dérivation :

On appelle *état d'une dérivation* un élément de $V^* \times \mathcal{P}(E) \times \mathcal{P}(E)$. Un état i est noté : $A_i[VAL_i][SAT_i]$ ou $A \in V^*$, $VAL_i, SAT_i \in \mathcal{P}(E)$. VAL_i est la liste des validations de l'état i et SAT_i la liste des saturations. L'état initial 0 est S (on a A_0 vide). Pour passer d'un état i à un état i+1, on choisit dans la liste des validations de i une règle e_j qu'on applique à i de la manière suivante :

$A_{i+1} = A_i$ + le symbole de V en partie droite de e_j

$SAT_{i+1} = SAT_i \cup$ les saturations de la partie droite de e_j

VAL_{i+1} = les validations de la partie droite de e_j moins SAT_{i+1}.

La dérivation s'arrête dès que la liste des validations de l'état courant est vide.

Exemple de dérivation : (on reprend la GVS de l'exemple ci-dessus)

[1,2][]	on choisit une validation :
1 → a[1,2,3][]	
1 → aa[1,2,3][]	
2 → aab[2,3][1]	union des saturations
3 → aabc[3,4][1,2]	1 est une saturation : elle est éliminée
4 → aabcd[][1,2]	de la liste des validations

Après application de 2, on ne peut plus appliquer 1, ce qui se traduit par la suppression de 1 de la liste des validations associée à la règle 3.

Dans le système d'analyse morphologique les règles de la grammaire sont nommées (RIB, PR5,...) et le vocabulaire est constitué d'informations linguistiques. Les règles ont la forme :

```
nom : informations linguistiques;
      VAL:=(liste de noms de règles) ou valeur prédéfinie;
      SAT:=(liste de noms de règles) ou valeur prédéfinie;
      indicateurs
```

VAL et SAT définissent respectivement les validations et les saturations. On a deux indicateurs possibles : FIM qui indique une règle terminale et ND qui indique une non-désinence (voir §5.1.2.4). Le mot clé VALOO désigne la liste prédéfinie des règles initiales.

5.1.2.2. Les modèles

Pour assurer l'interface entre le dictionnaire et la grammaire on utilise des modèles. Ce sont des représentants d'une classe de mots ayant le même comportement morphologique (les verbes du 1er groupe par exemple). Ils sont notés :

```
/nom/ : REG:=(liste de noms de règles) ou valeur prédéfinie;
        informations linguistiques;
        VAL:=(liste de noms de règles) ou valeur prédéfinie;
        SAT:=(liste de noms de règles) ou valeur prédéfinie;
```

On a les mêmes interprétations que ci-dessus avec en plus REG qui donne une liste des règles applicables à ce modèle.

5.1.2.3. Les informations linguistiques

Elles sont constituées d'expressions utilisant les informations portées par les modèles et par les éléments déjà reconnus. Ces informations sont rendues accessibles par deux sortes de registres :

- *types* : ils ne peuvent prendre qu'une seule valeur, ils sont au nombre de deux : CL (Classe Lexicale) et CD (Code de Dérivation). Le code de dérivation ne prend qu'une seule valeur mais elle peut être la concaténation de deux autres. CD est utilisé pour calculer les formes dérivées de certains mots.
- *variable* : VAR, elle peut prendre plusieurs valeurs, par exemple (FEMININ SINGULIER) pour DIVISION.

CL,CD,VAR désignent l'état courant alors que CLG, CDG et VARG désignent les valeurs des registres associés aux éléments déjà reconnus (c'est-à-dire à l'état précédent, G signifie Gauche). CLD, CDD, VARD désignent les valeurs des registres associés au modèle courant. Les expressions autorisées dépendent du registre mais visent en général à calculer la valeur de l'état courant en fonction de l'état précédent et du modèle.

Exemples : pour un registre X

X := XG	valeur de l'état précédent dans la dérivation
X := XD	valeur contenue dans le modèle associé à l'unité lexicale
CD := CDG||CDD	concaténation des valeurs de l'état précédent et du modèle
X := valeur	si X est un type
VAR := liste de valeurs	si c'est VAR
VAR := VARG-SIN+PLU	on remplace SIN par PLU.

Les variables contiennent des ensembles de valeurs dénotant des propriétés morphologiques (genre : mas, fem, neu ; personne : uno, dos, tre ; nombre : sin, plu ; etc... [COURTIN 76]).

5.1.2.4. Fonctionnement du transducteur

Le transducteur utilise les règles de la même façon qu'une GVS mais en tenant compte du modèle et du code morphologique. Le code morphologique (CM) est une liste de règles associée à un système de désinence qui pourra être activé dans certaines conditions (notamment si l'indicateur ND n'est pas présent dans la règle courante). Pour passer d'un état i à un état i+1 en appliquant la règle r, et pour le modèle m, on utilise les formules :

$SAT_{i+1} = \cup (SAT_i, SAT_r, SAT_m)$

$VAL_{i+1} = \cup (VAL_r, VAL_m) - SAT_{i+1}$

Si le numéro du modèle sélectionné est alors m+1, la liste des règles applicables est donnée par :

$\cap (REG_{m+1}, VAL_{i+1})$

Si cette intersection est vide et que r ne contient pas l'indicateur ND, alors la liste des règles applicables est :

$\cap (REG_{m+1}, CM_r)$ (code morphologique)

Pour l'état initial on a : SAT_0 =[] et VAL_0 =VALOO.

Nous allons illustrer les formules et le fonctionnement du transducteur sur un exemple (inspiré de [COURTIN 77]).

Exemple :

```
VALOO:=RIB ...
VAL1:=E1 E8
```

On a les règles :

```
RIB : CL:=CLD ; CM:=CMD ;
      VAR:=VARD.
E1 :  CL:=CLG ;
      VAR:=VARG+SIN ; FIM.
PR4 : CL:=CLG ;
      VAR:=VARD+PRE+IND ; FIM.
SU1 : CL:=CLG ;
      VAR:=VARD+PRE+SUB ; FIM.
```

Les modèles :

```
/femme/ :  REG:=RIB ; CL:=SUBC ;
           VAR:=FEM ; VAL:=VAL1.
/ / :      REG:=NF1 PR7 E1 ;
           VAR:=TRE+SING.
/aim/ :    REG:=RIB ; CL:=VERB ;
           CM:=PR4 SU1.
/ent / :   REG:=PR4 IS3 SU1 ;
           VAR:=TRE+PLU.
```

Dans la plupart des règles et des modèles ci-dessus, les listes de validation et saturation étant vides, les mots clés VAL et SAT n'apparaissent pas.

Le dictionnaire : (on ne donne que la chaîne et le modèle associé)

/maison/femme/	/chant/aim/
///	/ent /ent /

Analyse de "maison " et "chantent " :

a) "maison " : le dictionnaire fournit la base `/maison/` dont le modèle est `/femme/` ; par intersection entre `REG` de `/femme/` et `VALOO` on sélectionne la règle `RIB` qui donne :

`CL:=CLD` → `CL:=SUBC` (substantif commun)

`VAR:=VARD` → `VAR:=FEM` (féminin)

`CM:=CMD` → `CM:=()` puisque pas de `CM` dans le modèle.

Les validations sont égales à `VAL1` et sont donc `(E1,E8)`. On identifie ensuite la désinence / / dont le modèle est / /, avec `REG = NF1 PR7 E1`. Par intersection avec les validations `(E1,E8)` on sélectionne `E1` qui donne :

CL:=CLG transmission de `CL` donc `CL:=SUBC`

`VAR:=VARG+SIN` transmission de `VAR:=FEM` à laquelle on ajoute `SIN` (singulier).

De plus, la présence de l'indicateur `FIM` sur la règle `E1` permet l'acceptation de l'unité lexicale "maison " avec les informations linguistiques substantif commun féminin singulier.

b) "chantent " : on procède de la même façon, le modèle de `/chant/` : `/aim/`, ayant `REG:=RIB`, on applique `RIB` et on obtient :

`CL:=VERB, CM:=PR4 SU1`

Les validations étant vides et l'indicateur `ND` étant absent, on utilise les règles du code morphologique. La désinence `/ent /`, dont le modèle a pour `REG (PR4, IS3, SU1)`, permet par intersection avec `CM` d'obtenir deux règles `PR4` et `SU1`, elles assurent toutes les deux la transmission de `CL` et :

`PR4` → `VAR:=VARD+PRE+IND` → `VAR:=TRE+PLU+PRE+IND`
(troisième personne du pluriel du présent de l'indicatif)

`SU1` → `VAR:=VARD+PRE+SUB` → `VAR:=TRE+PLU+PRE+SUB`
(troisième personne du pluriel du présent du subjonctif)

5.1.3. Traitement des erreurs

Le transducteur lui-même est incapable de traiter un mot inconnu. Nous avons donc ajouté à l'algorithme un mécanisme de reprise en cas d'échec. Le transducteur se trouve dans un état d'échec : si aucune base préfixe de la chaîne à analyser n'est trouvée dans le lexique ou si aucune des bases trouvées ne conduit à une segmentation correcte. Dans ce cas, l'algorithme d'analyse considère le segment compris entre le caractère courant et le premier blanc qui suit ce caractère. Ce segment est considéré comme ne donnant pas de résultat et la transduction est relancée sur la suite de ce segment dans la chaîne d'entrée. Le segment incorrect pourra ensuite être corrigé manuellement ou soumis aux techniques de correction du §5.2.

Ce mécanisme très simple a l'inconvénient de considérer le blanc comme un caractère particulier et cela rend quasi impossible la correction des erreurs d'insertion d'un blanc au milieu d'un mot ou d'effacement d'un blanc. En effet,

la segmentation donnera soit deux mots incorrects correspondant à un seul, soit un mot incorrect correspondant à deux mots corrects.
Exemple :

"le méca nisme de reprise " donne comme résultat :

"le "	déterminant, masculin singulier
	ou pronom personnel masculin singulier, complément d'objet
"méca "	pas de résultat
"nisme "	pas de résultat
"de "	préposition
"reprise "	nom commun, masculin singulier
	ou participe passé, féminin singulier

Cet analyseur morphologique pêche un peu par la pauvreté des informations qu'il calcule. Cette pauvreté est bien sûr liée au faible volume d'information contenu dans les entrées du lexique. Mais nous disposons d'un lexique de 35000 entrées dans le dictionnaire de bases. Ces bases permettent de reconnaître et d'engendrer environ 250000 formes du français.

5.2. Techniques de correction de niveau lexical

Un segment de la chaîne d'entrée est un mot erroné si l'analyseur morphologique ne peut pas le reconnaître comme un assemblage valide d'éléments du dictionnaire. Cette définition restreint bien entendu l'ensemble des mots corrects à ceux qui peuvent être reconnus à partir du lexique. Quelle que soit la taille de ce lexique, la couverture d'une langue ne peut être que partielle car un lecteur (ou un auditeur) humain a toujours l'aptitude à reconnaître comme corrects des mots nouveaux dès l'instant qu'il peut leur associer une interprétation (par exemple "solution" → "solutionner", "nationalisation" → "dénationalisation"). C'est un invariant des systèmes basés sur des lexiques que de considérer la notion d'erreur relativement à un certain *référent* : le dictionnaire.

Les techniques que nous avons développées visent à corriger automatiquement un mot erroné : il s'agit de trouver dans le dictionnaire le (ou les) mot qui puisse être considéré comme la correction du segment erroné.

Plus généralement, il va s'agir d'extraire du dictionnaire les chaînes de caractères *les plus proches* du segment incorrect, selon un certain critère. La définition de ce critère revient à la définition d'une fonction S, dite *fonction de similarité*, utilisée comme suit : si E est la chaîne erronée, on extrait du dictionnaire toutes les chaînes C telles que S(E) = S(C). De la qualité de cette fonction va dépendre la qualité des corrections ; il faut notamment trouver un compromis entre une fonction trop précise (par exemple S(E) = E) donnant d'excellentes corrections mais trop sensible aux erreurs, et une fonction trop souple (par exemple S(E) = longueur de E) peu sensible aux erreurs mais donnant trop de corrections. [MARET 87] contient une discussion approfondie des problèmes liés au calcul de distance entre chaînes.

5.2.1. Clés de similarité

Une clé de similarité est un exemple typique de fonction de similarité. L'image d'une chaîne est une autre chaîne contenant les mêmes lettres mais éventuellement dans un ordre différent, en évitant les répétitions, en éliminant les diacritiques, ...

Les clés de similarité trouvent leur justification dans les résultats des travaux de [DAMERAU 64]. En analysant la nature des erreurs lexicales des textes, Damerau a constaté que 80% des mots erronés ne comportent qu'une faute et que cette faute est de 4 natures différentes :

- effacement d'une lettre (D)
- insertion d'une lettre (I)
- substitution d'une lettre par une autre (S)
- transposition de deux lettres adjacentes (T)

Comme exemple de clé de similarité, on peut citer l'alphagramme [DEBILI 86], qui reprend toutes les lettres du mot dans l'ordre alphabétique en éliminant les accents ("papier" → "aeippr"). Débili justifie cette clé par la rareté des anagrammes en français.

La clé de similarité que nous avons choisi de mettre en œuvre [LU 86, COHARD 88] est la *clé squelette* [POLLOCK 84]. Cette clé est construite en gardant la première lettre, puis en concaténant les consonnes dans leur ordre d'apparition mais sans répétition, puis les voyelles dans leur ordre d'apparition sans répétition. De plus, pour le français, nous avons remplacé les lettres comportant un signe diacritique (accent, cédille) par leur équivalent sans diacritique (voir exemples figure 5.1).

Chaîne	Clé	Type d'erreur
occasion	ocsnaio	
ocasion	ocsnaio	Effacement d'un c
occasion	ocznaio	Substitution de s par z
inonder	indroe	
innonder	indroe	Insertion d'un n
aéroport	arpteo	
aéropor	arpeo	Effacement du t
aréoport	arpteo	Transposition du r et du e

Figure 5.1 : Exemples de clés squelettes

Pollock et Zamora [POLLOCK 84] justifient cette clé avec les observations suivantes :

- la première lettre est la plus rarement mal orthographiée ;
- les consonnes portent plus d'information que les voyelles ;
- l'ordre des consonnes est très souvent conservé ;
- cette clé est insensible au doublement des consonnes et peu sensible aux transpositions.

En considérant les mots ayant une seule erreur le programme de Pollock et Zamora a pu corriger entre 85 et 95% des erreurs d'édition (insertion, effacement, transposition, substitution), et entre 65 et 80% des erreurs totales.

L'utilisation de cette clé repose sur l'existence d'une table T donnant, pour chaque forme possible, la clé correspondante. Cette table est triée par ordre alphabétique des clés. En présence d'un mot erroné, on construit sa clé K et on consulte la table pour trouver l'indice i de la clé la plus proche de K. Pour garantir un meilleur taux de correction, on extrait de la table toutes les formes dont l'indice est compris entre i-n+1 et i+n où n est une constante fixée. Les 2n formes obtenues sont alors filtrées : on ne considère parmi ces formes que celles qui sont égales à la forme erronée à une erreur d'édition près (insertion, effacement, transposition, substitution). Les formes restantes constituent les propositions de correction de la forme erronée.

Exemple de correction : (emprunté à [LU 86])

Mot à corriger : "line"

Extrait de la table (centré sur "lion" dont la clé "lnio" est la plus proche de celle du mot faux "lnie") :

Clé	Forme	Clé	Forme	Clé	Forme	Clé	Forme
lgnie	ligne	lmie	lime	lnoi	loin	lrdou	lourd
lgre	léger	lmpae	lampe	lnte	lent	lreu	leur
lgroe	loger	lngie	linge	lnue	lune	lrgae	large
lieu	lieu	lngo	long	loi	loi	lrie	lier
lmae	lame	**lnio**	**lion**	lpoue	loupe	lrie	lire

Corrections :	Distance du centre :
"line" est l'omission de "linge"	2
"line" est la substitution de "lune"	3
"line" est la substitution de "lime"	4
"line" est l'omission de "ligne"	9
"line" est la substitution de "lire"	10

D'un point de vue pratique, cette méthode de correction (quelle que soit la clé choisie) impose la construction d'un dictionnaire clé/formes. Dans le cadre de la réalisation d'un prototype de détection/correction (DECOR [COHARD 88]), nous avons développé un outil capable de construire de manière entièrement automatique un tel dictionnaire. Cet outil est une simple utilisation du générateur morphologique de PILAF.

Que ce soit dans le prototype DECOR ou dans les travaux de [LU 86], le dictionnaire clé/forme utilisé était de taille très réduite (< 4000 formes). Le gros dictionnaire de bases dont nous disposons actuellement nous a permis de créer un dictionnaire clé/forme d'une taille appropriée à une application réelle : il contient environ 250000 formes. On constate avec un dictionnaire de cette taille que la méthode de correction présentée ci-dessus n'est plus appropriée. En effet, pour obtenir les mêmes corrections de "line" que dans l'exemple ci-dessus, on doit donner à n une valeur supérieure ou égale à 1130 (qui est la longueur de l'intervalle séparant "ligne" du centre "lien"). Une telle valeur grève les performances au point de rendre la méthode inutilisable de manière interactive (les temps de correction devenant trop élevés). Ce problème est lié au fait qu'un grand nombre de forme de longueur très inférieure ou très supérieure à celle de "line" sont venues s'insérer entre les formes présentées ci-dessus. Toutes ces formes ne peuvent pas être des solutions puisse qu'on ne

considère qu'une seule erreur par mot et donc la longueur de la correction est égale à celle du mot faux à 1 près.

Pour améliorer les résultats de cette méthode avec un gros dictionnaire, on pourrait fixer un n très grand mais ne retenir dans l'intervalle que les formes dont la longueur diffère de celle du mot faux d'au maximum 1. Mais le problème de la taille de l'intervalle ne serait pas résolu : presque 10000 formes séparent "sbnquoie" qui est la clé la plus proche de "sbnao" ("sabon") et "svnao" la clé de "savon". On voit que le simple filtrage sur les longueurs prendrait un temps considérable.

Une meilleure méthode consiste à découper le dictionnaire clé/forme en fonction de la **longueur de la forme** en conservant dans chaque sous-table un tri par ordre alphabétique des clés. Pour corriger une forme incorrecte E, de longueur L, on calculerait sa clé et on appliquerait la méthode initiale aux trois tables dont les formes ont les longueurs L-1, L et L+1. Dans la table L-1, on ne traiterait que les erreurs d'effacement, dans la table L les erreurs de substitution et de transposition et dans la table L+1 les erreurs d'insertion.

D'un point de vue théorique, le gros défaut de cette clé est sa sensibilité aux erreurs situées en début de mot : une erreur sur la première lettre a très peu de chance d'être corrigée correctement. Il en va de même pour les erreurs portant sur les consonnes de début de mot. Par contre, grâce à sa faible sensibilité aux erreurs de transposition et de dédoublement ainsi qu'aux erreurs portant sur les signes diacritiques, elle nous paraît très bien adaptée à la correction d'erreurs typographiques (mauvaise utilisation du clavier) qui apparaissent plus fréquemment en milieu ou fin de mot.

5.2.2. Phonétique

Dans ce type de méthode, la fonction de similarité entre chaînes calcule la représentation phonétique de la chaîne. Cette méthode a été utilisée dans de nombreux systèmes [LAHENS 86, MARET 87, EMIRKANIAN 88a, VERONIS 87] et d'autres. L'importance de cette méthode apparaît si on considère que l'ordinateur est devenu accessible à un très grand nombre d'utilisateurs dans des domaines très divers ; la compétence moyenne de ces utilisateurs concernant la langue naturelle s'en trouve naturellement diminuée mais les erreurs de compétence préservent très souvent la phonétique des mots (si on utilise un mot, c'est qu'on l'a entendu, mais on ne sait pas forcément l'écrire).

L'algorithme de correction par cette méthode est le suivant : étant donné une forme incorrecte E, calculer la représentation phonétique de E puis extraire comme hypothèses de correction toutes les formes du lexique qui ont la même représentation phonétique. Nous allons présenter comment ces deux étapes ont été mises en œuvre dans notre équipe.

La première étape s'appuie sur le transducteur d'état finis de PILAF. [STRUBE DE LIMA 90] définit un dictionnaire de p-graphèmes qui ne sont pas exactement les graphèmes élémentaires du français mais un ensemble de pseudo-graphèmes adaptés à la transcription. Les modèles et les règles sont ici de nature phonétique ; l'analyse d'un mot consiste à trouver un découpage (ou plusieurs) de ce mot en p-graphèmes et les règles valident ce découpage et

effectuent la transduction c'est-à-dire la transcription graphique → phonétique. Cette transcription ne suit pas exactement le codage phonétique du français : des simplifications ont été introduites d'une part pour faciliter la construction de la grammaire d'états finis et d'autre part pour rendre la transcription moins sensible aux erreurs. Il faut noter qu'à une chaîne graphique donnée peuvent correspondre plusieurs chaînes phonétiques (l'inverse est également vrai). La figure 5.2 donne des exemples de transcription (empruntés à [STRUBE DE LIMA 90]).

Chaîne graphique	**Chaînes phonétiques (ou orales)**
vinku	v.in.k.u.
okurranse	o.k.u.r.an.s.
peti	p.é.t.i. p.e.t.i.
echo	e.ch.o e.k.o é.ch.o é.k.o

Figure 5.2 : Exemples de transcriptions

Pour la deuxième étape, on ne parcourt pas tout le dictionnaire en calculant pour chaque forme sa transcription phonétique puis en comparant cette dernière à celles obtenues pour le mot faux. On utilise plutôt la fonction inverse de la fonction de similarité précédente : on calcule pour chaque chaîne phonétique associée au mot faux toutes les transcriptions graphiques possibles. Cette étape a donné lieu à deux mises en œuvre différentes : l'idée de la première est de calculer directement des chaînes graphiques correctes alors que la deuxième admet de calculer des chaînes incorrectes qui seront ensuite filtrées.

La première méthode repose sur l'utilisation d'un analyseur phonétique (l'équivalent d'un analyseur morphologique mais pour des chaînes phonétiques). Elle s'appuie sur le transducteur d'états finis de PILAF et suppose donc la définition d'un dictionnaire de bases phonétiques ainsi que de règles et de modèles assurant une bonne transduction phonétique → graphique. D'abord envisagée sur la base des travaux de [COURTIN 77], cette méthode n'a pas été retenue car elle impose la construction d'un dictionnaire de bases phonétiques équivalent du dictionnaire de bases morphologiques et une telle construction ne peut se faire que manuellement.

Règles associées au p-phonème "m."

p-graphème	**début**	**milieu**	**fin**
m	oui	oui	oui
me	non	non	oui
m m	non	oui	non
mme	non	non	oui

Figure 5.3 : Exemple de règles de correspondance

La deuxième méthode n'utilise pas le transducteur de PILAF mais un ensemble de règles très simples de correspondance entre les p-phonèmes et les p-graphèmes (voir figure 5.3). A chaque p-phonème, on fait correspondre la liste des p-graphèmes qui peuvent le représenter ; on associe de plus à chaque p-graphème possible un triplet de booléens indiquant si ce p-graphème peut

être une représentation du p-phonème en début, milieu ou fin de mot (ceci afin de limiter le nombre de chaînes graphiques produites).

Les informations portées par les booléens ont été calculées en analysant le dictionnaire de 250000 formes dont nous disposons.

L'inconvénient évident de cette méthode est le nombre considérable de chaînes graphiques obtenues à partir d'une chaîne phonétique. Pour limiter la combinatoire, [STRUBE DE LIMA 90] propose deux améliorations :

- utilisation d'une liste de cas particulier permettant de restreindre les règles. Ainsi dans la traduction du phonème "i." on accepte le graphème "î" en début de mot car il apparaît dans les formes "île", "îles", "îlot" et "îlots". On préfère dans ce cas éliminer la possibilité de "î" en début de mot et stocker dans un fichier de cas particuliers les chaînes phonétiques concernées et leurs représentations graphiques (par exemple "i.l.o." associée à "îlot" et "îlots").
- analyse du dictionnaire de formes pour produire une table d'adjacence des bi-grammes de graphèmes. On parcourt le dictionnaire en calculant pour chaque paire de graphèmes si leur concaténation apparaît dans une forme. Ce calcul n'est bien sûr réalisé qu'une fois. Cette table est ensuite utilisée pour limiter les combinaisons au moment de la reconstitution des chaînes graphiques (voir ci-dessous).

La génération des chaînes graphiques correctes à partir des chaînes phonétiques associées à la forme erronée comprend 3 étapes :

1- Pour chaque p-phonème de chaque chaîne phonétique on calcule la liste des p-graphèmes possibles en utilisant les règles de correspondance. On aura bien entendu pris soin de vérifier que la chaîne phonétique n'est pas un cas particulier (dans ce cas, le traitement de cette chaîne est terminé).

2- On assemble pour chaque chaîne phonétique toutes les combinaisons possibles de p-graphème en vérifiant pour chaque paire que l'assemblage est possible (par consultation de la matrice d'adjacence).

3- La liste des chaînes graphiques obtenue est filtrée par consultation du dictionnaire de formes (on aurait pu la filtrer par analyse morphologique).

Les corrections obtenues par cette méthode sont de bonne qualité même si les temps de réponse (sur un micro-ordinateur) ne sont pas aussi rapides qu'on le voudrait (entre 3 et 8 secondes pour corriger un mot). Le principal problème reste le nombre de corrections proposées, principalement quand la forme incorrecte correspond à une forme d'un verbe du premier groupe : pour "alez" par exemple, on obtient 30 chaînes graphiques ("allai", "allaient", ..., "halai", "halaient", "hâlais", ...).

5.2.3. Génération morphologique

Cette troisième méthode de correction [COHARD 88] tranche un peu avec les deux précédentes en ce sens qu'elle n'exploite pas la notion de similarité des chaînes mais qu'elle se place à un niveau plus élevé : celui de l'interprétation

des éléments constitutifs d'un segment. Elle a été mise en œuvre dans le but de corriger les erreurs liées à une mauvaise utilisation des désinences (erreurs morphogrammiques dans la classification de Catach). Ces erreurs de compétence peuvent être particulièrement fréquentes si l'utilisateur maîtrise mal la langue : un enfant ou un étranger en situation d'apprentissage du français.

Exemples :

Très classiques : "chevals", "journals", "faisez"
Plus rares : "allerons", "chacaux"

L'idée est de calculer avec l'analyseur morphologique un maximum d'informations (genre, nombre, temps,...) sur la forme erronée puis d'utiliser ces informations pour engendrer le mot correct avec le générateur morphologique.

Pratiquement, on procède comme suit :

1- extraire la plus longue racine correcte du mot faux : par exemple "all" pour "allerons" ou "chev" pour "chevals".
2- faire l'analyse morphologique du reste du mot pour obtenir un certain nombre d'informations : par exemple première personne du pluriel du futur de l'indicatif pour "erons" et masculin pluriel pour "als".
3- utiliser le générateur morphologique sur la racine en ne retenant que les formes engendrées possédant les mêmes valeurs de variables que celles calculées pour le mot faux. On obtient ainsi "irons" pour "allerons" et "chevaux" pour "chevals".

Bien que d'une portée limitée (il faut trouver une racine correcte et qui soit la bonne et il faut que le reste de la chaîne soit analysable) cette méthode permet de corriger des erreurs sur lesquelles les deux méthodes précédentes auraient échoué.

5.2.4. Conclusion

Ces trois méthodes ont été intégrées à un prototype d'éditeur de textes baptisé DECOR [COHARD 88]. L'idée est que chaque méthode étant bien adaptée à un certain type d'erreurs, l'utilisation conjointe des trois doit permettre d'obtenir de bonnes propositions de correction. Dans DECOR, les méthodes sont chaînées par paires : clé squelette + phonétique ou clé squelette + morphologie.

L'architecture du système de correction repose donc sur une simple composition séquentielle. On peut cependant envisager des architectures plus sophistiquées : [COURTIN 89c] propose une architecture dans laquelle le choix de la méthode de correction à utiliser en priorité est laissé à un *programme pilote* qui effectue ce choix en fonction de l'utilisateur. Cette architecture s'appuie sur le constat que la typologie des erreurs commises est toujours fonction de l'utilisateur : un étranger ou un enfant maîtrisant mal le français commettront sans doute plus d'erreurs de compétence qu'un ingénieur rédigeant un rapport. Ce dernier risque par contre de commettre des erreurs de performance liées à l'utilisation du clavier. Il paraît judicieux de privilégier dans le premier cas les méthodes de correction par phonétique et par génération morphologique, et dans le second cas la méthode de la clé squelette.

Parmi les conclusions que nous avons tirées de la réalisation de ce prototype, deux sont apparues importantes dans l'optique d'un système réel de détection et de correction lexicales :

- si l'utilisation conjointe de trois méthodes permet d'obtenir de meilleurs résultats, elle implique par contre l'exploitation d'un dictionnaire de base et d'un dictionnaire clé/forme. Pour obtenir une bonne couverture du français, on doit accroître la taille de ces dictionnaires et l'encombrement devient alors non négligeable (environ 7 Mo pour les deux dans la version actuelle). Comme le support de diffusion typique de ce type de système est le micro-ordinateur, il importe de réduire au maximum cet encombrement. Des travaux ont été menés dans cette voie [BAINER 89, LOPEZ 90] et ont permis une première réduction de 50% environ.
- quelle que soit la méthode utilisée, le nombre de corrections obtenues est rarement 1 et même dans ce cas, la correction n'est pas nécessairement la bonne. Il paraît donc exclu d'envisager une correction entièrement automatique au niveau lexical (on verra que c'est encore plus vrai au niveau syntaxique). La vérification et la correction d'un texte complet ne peuvent être qu'assistées par ordinateur. Les problèmes qui se posent alors relèvent de l'ergonomie : il s'agit de proposer les meilleures solutions possibles et en nombre peu important. Dans le cas contraire, l'utilisateur aura plus vite fait de consulter un dictionnaire et de corriger manuellement ; il vaudrait mieux alors exploiter nos outils pour fournir un dictionnaire électronique accessible par des méthodes efficaces (par phonétique ou recherche incrémentale par exemple). Nous reviendrons sur ces problèmes de stratégie au chapitre 7.

5.3. Vers une intégration des Ψ-termes au niveau lexical

Les informations linguistiques calculées par l'analyseur morphologique actuel sont insuffisantes pour une application de détection/correction d'erreurs dès qu'on veut dépasser le niveau lexical (voir à ce sujet le chapitre 6 et [STRUBE DE LIMA 90]. Nous allons voir comment les Ψ-termes présentés au chapitre 3 peuvent être intégrés à cet analyseur sans modification notable du modèle de base. Il s'agit de faire calculer par l'analyseur morphologique un maximum d'information sur chaque unité lexicale prise indépendamment. Nous avons classé ces informations en 3 niveaux : morphologique, syntaxique et sémantique (voir également le §3.2.1), mais il est clair que les frontières entre ces niveaux sont plutôt floues.

5.3.1. Représentation des connaissances de nature morphologique

Les connaissances de nature morphologique sont celles qui concernent la nature et la forme de l'unité lexicale seule, indépendamment des rapports qu'elle peut entretenir avec les autres mots. Ces informations sont codées dans l'analyseur morphologique actuel sous forme d'un ensemble de symboles. La présence d'un symbole indique que l'unité lexicale possède la propriété associée

à ce symbole (l'interprétation de cette propriété est laissée aux modules ultérieurs : syntaxique et/ou sémantique).

Exemples :

"aimer" : {infi}
"chanterons" : {verb, uno, plu, fut, ind}
"livre" : {subc, mas, sin}
ou {subc, fem, sin}
ou {verb, tre, sin, pre, ind}
ou {verb, tre, sin, pre, sub}
ou {verb, dos, imp}

Ce type de connaissance peut être codé dans les Ψ-termes sous forme de paires attribut-valeur, la seule contrainte par rapport à l'ancien codage est qu'il faut nommer les propriétés.

Exemples :

"aimer" : UL(cat => infi)
"chanterons" : UL(cat => verb; pers => uno;
nbr => plu; tps => fut; mode => ind)
"livre" : UL(cat => subc; gnr => mas; nbr => sin)
ou UL(cat => subc; gnr => fem; nbr => sin)
ou UL(cat => verb; pers => tre;
nbr => sin; tps => pre; mode => ind)
ou UL(cat => verb; pers => tre;
nbr => sin; tps => pre; mode => sub)
ou UL(cat => verb; pers => dos; mode => imp)

L'adaptation de l'analyseur morphologique actuel suppose :

- de coder les connaissances sous forme de Ψ-termes dans le lexique, les règles et les modèles ;
- de munir le langage du transducteur des moyens permettant de combiner ces Ψ-termes, c'est-à-dire pour l'essentiel des mêmes actions que l'analyseur syntaxique : affectation, unification, ajout ou suppression d'un trait.

On reprend ci-dessous les règles et les modèles donnés en exemple au §5.1.2.4 en transformant ce qui concerne le codage des informations linguistiques par un codage utilisant les Ψ-termes. On utilise le mot clé `VAR` pour désigner le Ψ-terme courant et les références `D` et `G` pour `VARD` et `VARG`. Les registres `CL`, `CLD`, `CLG` sont remplacés par le trait `cat` de `VAR`, `D` et `G`. Tout ce qui concerne le fonctionnement du transducteur proprement dit n'est pas modifié (listes de règles : validations, saturations, code morphologique).

Exemple :

```
VALOO:=RIB ...
VAL1:=E1 E8
```

On a les règles :

```
RIB : CM:=CMD ;
      VAR := D.
E1 :  VAR:= UNIF(G, UL(nbr => SIN)) ; FIM.
PR4 : VAR:= UNIF(D, UL(tps => PRE; mode => IND)) ;
      VAR.cat:= G.cat ; FIM.
SU1 : VAR:= UNIF(D, UL(tps => PRE; mode => SUB)) ;
```

```
        VAR.cat:= G.cat ; FIM.
```

Les modèles :

```
/femme/ :   REG:=RIB ;
            VAR := UL(cat => SUBC; gnr => FEM);
            VAL:=VAL1.
/ / :       REG:=NF1 PR7 E1 ;
            VAR := UL(pers => TRE; nbr => SIN).
/aim/ :     REG:=RIB ; VAR := UL(cat => VERB) ;
            CM:=PR4 SU1.
/ent / :    REG:=PR4 IS3 SU1 ;
            VAR := UL(pers => TRE; nbr => PLU).
```

5.3.2. Représentation des connaissances de nature syntaxique

Sont de nature syntaxique toutes les connaissances qui concernent les liens liant une unité lexicale au reste de la phrase.

Certaines de ces connaissances sont codées sous forme de variables dans l'analyseur actuel (par exemple la fonction des pronoms personnels codée par les variables `suj` (sujet), `cod` (complément d'objet direct),...). Pour celles-ci on peut procéder de la même manière que pour les connaissances morphologiques.

De par leur rôle particulièrement important dans la construction des phrases, les noms et surtout les verbes, sont porteurs de nombreuses informations dont le premier rôle est de faciliter la construction des structures de dépendances en permettant la levée de nombreuses ambiguïtés. On aura par exemple pour les verbes les traits :

- `pron` indiquant si le verbe peut être pronominal ou non
- `trans` verbe transitif ou intransitif
- `aux` auxiliaire de conjugaison du verbe
- `syn` trait complexe précisant le régime de complémentation du verbe : quels compléments admet-il ? avec quelle fonction syntaxique ? introduit par quelle préposition ?

Ces traits ne relèvent pas de la morphologie et ne peuvent donc pas être calculés par le biais des règles et des modèles morphologiques. En effet, deux verbes ayant la même conjugaison ("sembler" et "donner"), n'ont pas forcément le même régime de complémentation. Ils doivent donc être intégrés aux entrées lexicales (bases).

Exemple :

Pour /donn/, base de "donner", on aura dans le lexique le Ψ-terme :

```
UL(CAT => verb;
   trans => _TOUT_;
   aux => xav;                    -- avoir
   syn => proposition(
     prédicat => DONNER;
     sujet => UL(syn => groupe_nominal)
     objet => UL(syn => groupe_nominal)
     objet-indirect => UL(
        CAT => à ;
        syn => groupe-prépositionnel)))
```

Le lexique actuel ne contient pour une base que le numéro du modèle morphologique, l'introduction de ces connaissances syntaxiques impose donc une modification de la structure du lexique (voir 5.3.4).

5.3.3. Représentation des connaissances de nature sémantique

Sont de nature sémantique les connaissances qui définissent le sens du mot. Ce sens est toujours relatif à un certain contexte : celui du monde réel dans lequel il est employé.
Ces connaissances peuvent prendre la forme de traits à valeur simple désignant le concept associé au mot, ou entrer dans des constructions plus complexes établissant des relations entre les concepts (notamment pour les verbes et pour certains noms à valeur prédicative). Là encore, ces informations ne relèvent pas de la morphologie et doivent donc être portées par la base dans le lexique.
Exemple :

Pour la base /mang/ de "manger" :

```
UL(sém =>
     MANGER(agent => ANIME;
            patient => COMESTIBLE)))
```

5.3.4. Mise en œuvre d'une nouvelle analyse morphologique

Comme on l'a vu ci-dessus, l'adaptation de la morhopologie actuelle aux Ψ-termes ne pose pas de problèmes particuliers si on se limite aux connaissances morphologiques.

Pour ce qui concerne les connaissances syntaxiques et sémantiques, il est nécessaire d'augmenter le volume des entrées du dictionnaire de base. La méthode la plus simple consiste bien sûr à ajouter un Ψ-terme à chaque entrée en stockant dans ce Ψ-terme toutes les informations nécessaires. On peut ainsi très aisément établir des liens entre les fonctions syntaxiques et les relations sémantiques (voir §3.2.1.3). Mais on imagine sans peine qu'avec une telle solution, la taille du lexique va croître considérablement.

Mais les travaux sur la syntaxe du verbe en français [GROSS 75a] et notamment les travaux récents du CRISS[1] [LALLICH-BOIDIN 86, BLANK 87a, 87b], n'ont fait apparaître que peu de régimes de complémentation différents (moins d'une vingtaine dans [BLANK 87a]). De la même manière, on peut imaginer qu'à un même concept ou prédicat sémantique seront associées plusieurs unités lexicales. On propose donc d'ajouter à chaque entrée du dictionnaire de bases deux pointeurs : un vers un Ψ-terme d'une base de Ψ-termes syntaxique, l'autre vers une base de Ψ-termes sémantiques (voir figure 5.4).

Le problème qui reste à résoudre est celui de la combinaison de ces Ψ-termes. Le meilleur moyen est de procéder à l'unification de **3** Ψ-termes : celui qui aura été calculé par l'analyseur morphologique et les deux Ψ-termes portés par l'entrée de la base dans le lexique. Mais la séparation de la syntaxe et de la sémantique, si elle permet des factorisations évidentes, interdit de coder les

[1]Centre de Recherche en Informatique et Sciences Sociales, Université de Grenoble II.

liens entre les fonctions syntaxiques et les relations sémantiques. Compte tenu du nombre réduit de compléments d'un verbe (si on exclut les compléments circonstanciels), on peut ajouter à l'entrée lexicale une relation d'équivalence sous la forme de couples de chemins. Chaque couple établit une relation de coréférence entre deux sous-Ψ-termes du résultat. Cette relation apparaît sur la figure 5.4 comme le dernier champ de l'entrée lexicale.

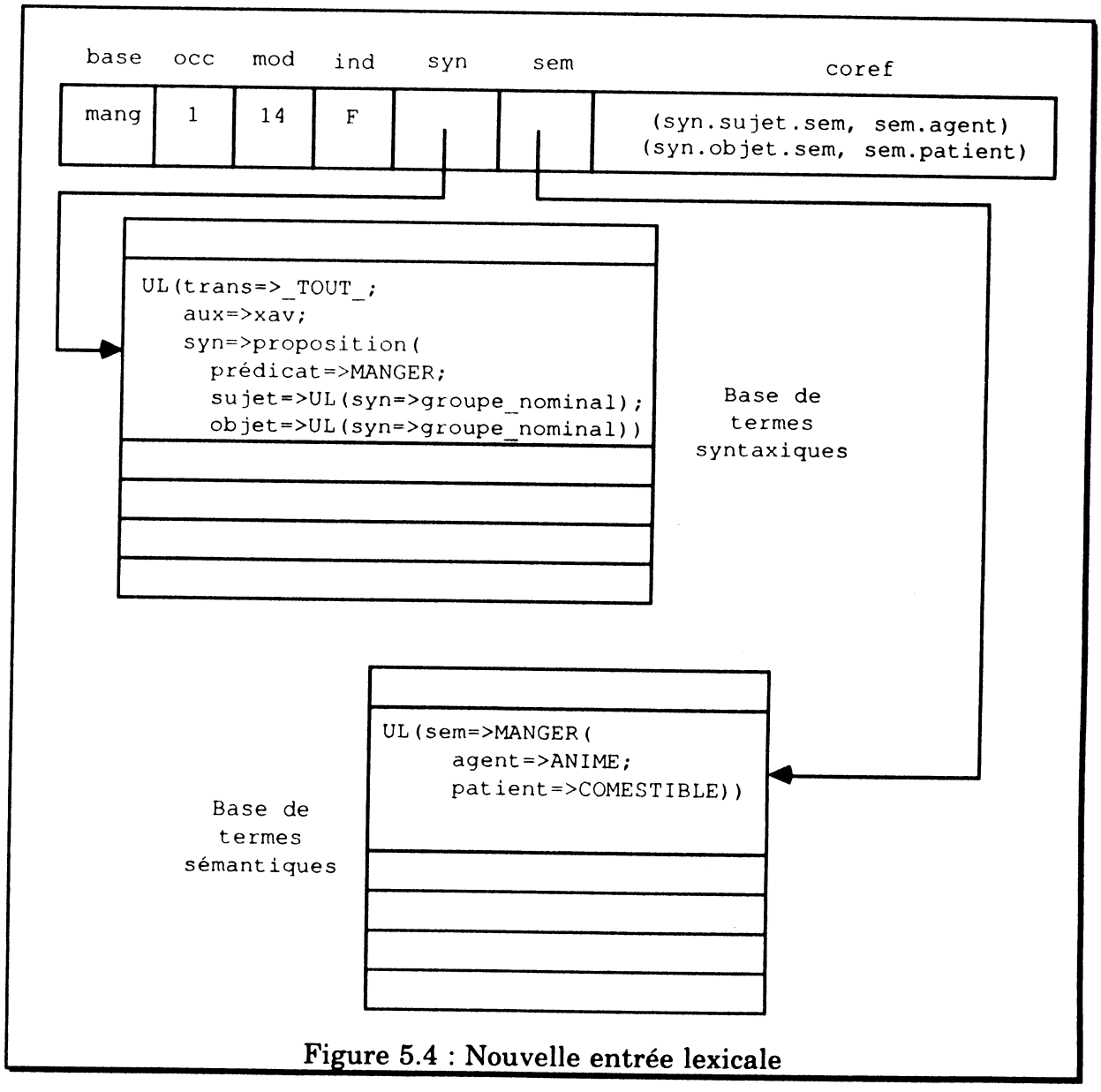


Figure 5.4 : Nouvelle entrée lexicale

Si un mot possède, pour une même interprétation morphologique, plusieurs interprétations syntaxiques et/ou sémantiques, on devra soit l'indexer autant de fois que nécessaire, soit donner plusieurs valeurs alternatives aux trois derniers champs de l'entrée lexicale. La première solution présente l'inconvénient de produire une redondance des calculs dans l'analyse morphologique alors que la deuxième pose le problème de la détermination du nombre maximum d'interprétations possibles et de l'encombrement inutile pour les mots qui n'ont qu'une interprétation.

Exemple :
Le verbe "fumer" possède 3 interprétations syntaxico-sémantiques différentes illustrées par les phrases :
"Le volcan fume."
"Pierre fume la pipe."
"Le paysan fume son champ."

Quoiqu'il en soit, il est extrêmement difficile de construire une base sémantique complète et cohérente pour un dictionnaire de cette taille. De plus, la multiplicité des ambiguïtés produites risquerait de grever considérablement les performances du système. Nous préférons donc, dans un premier temps, imaginer que les deux bases de Ψ-termes sont propres à une application particulière, où les multiples sens sont rares et où les régimes de complémentation des verbes sont relativement bien déterminés.

La structure de lexique présentée ci-dessus a un avantage énorme qui concerne sa construction : on peut très facilement ajouter trois champs à chaque entrée du dictionnaire de bases actuel. On peut ensuite affecter aux nouveaux champs des valeurs sans effet sur le résultat de la morphologie seule (par exemple `UL(syn => _TOUT_)` pour le champ `syn`). L'intérêt évident est qu'on obtient ainsi très facilement un dictionnaire syntaxico-sémantique de 35000 bases. Il est clair que ce dictionnaire est beaucoup trop général pour être exploité, mais on peut ensuite progressivement l'enrichir (c'est-à-dire le spécialiser), en construisant les deux bases de Ψ-termes.

Pour terminer, on peut remarquer que la contrainte imposée par l'analyseur syntaxique de PILAF, à savoir qu'un segment est insegmentable, n'est plus obligatoire avec l'analyseur syntaxique décrit au chapitre 2. En effet, si un même segment de la chaîne d'entrée donne lieu à plusieurs découpages, on peut les intégrer simultanément dans la structure de données de l'analyseur syntaxique. L'unité minimale ajoutée à cette structure n'est plus obligatoirement un arbre portant une unité lexicale mais peut être une forêt.
Exemple :
Avec le segment "de la" dans les deux phrases :
"Il mange de la soupe."
"Il vient de la montagne."
on a deux découpages possibles : dans le premier cas on considère "de la" comme une seule unité lexicale (article partitif) et dans le deuxième cas comme deux unités "de" (préposition) et "la" (article). Mais l'analyseur morphologique doit calculer les deux découpages à chaque fois, laissant le soin à l'analyseur syntaxique de déterminer lequel est le bon dans chaque cas. Après avoir introduit le pronom et le verbe, on aura l'arbre :

```
     verb
    /
pper
```

L'introduction des résultats de l'analyse du segment "de la" va produire les forêts :

```
     verb - prep - art
    /
pper
     verb - art_part
    /
pper
```

6

Le niveau syntaxique

Deux types de problèmes se posent au niveau syntaxique dans un système de détection et de correction d'erreurs :

- le premier problème est de gérer les erreurs de niveau lexical qui n'ont pas pu être corrigées. Il va s'agir pour l'analyseur syntaxique d'être capable de manipuler des mots inconnus qui peuvent être soit des mots trop mal formés qui n'ont pas permis aux techniques de correction lexicale de proposer des hypothèses, soit des mots corrects qui ne sont pas dans le lexique, soit enfin des mots incorrects mais auxquels un lecteur humain sait donner une interprétation (néologismes).
- le deuxième problème est de détecter et si possible de corriger les erreurs de niveau syntaxique : ce sont toutes les erreurs détectées au niveau syntaxique, c'est-à-dire qui entraînent l'impossibilité d'associer une structure correcte à une phrase donnée.

Nous détaillons, dans un premier paragraphe, le type d'erreurs que l'on peut trouver au niveau syntaxique et les différentes stratégies que l'on peut adopter pour les corriger. Après avoir précisé nos choix, nous expliquons dans le deuxième paragraphe comment l'analyseur présenté dans la première partie de cette thèse peut manipuler des mots inconnus. Nous terminons en présentant le prototype de vérificateur syntaxique développé dans notre équipe [STRUBE DE LIMA 90] et les améliorations apportées par le modèle présenté dans la première partie.

6.1. Classification et traitement des erreurs de niveau syntaxique

6.1.1. Classification

Nous distinguons deux classes d'erreurs de niveau syntaxique en fonction des techniques de correction qui peuvent leur être appliquées :

1 - les erreurs portant sur les variables morpho-syntaxiques attachées aux mots. Ce type d'erreur présente l'avantage de ne pas bloquer l'analyse si on utilise un analyseur qui ne tient pas compte des variables morphologiques associées aux mots mais seulement de la catégorie du mot. C'est le cas de l'analyseur de dépendances de PILAF qui a permis la réalisation du prototype de vérificateur présenté au §6.3.

2- les erreurs portant sur un ou plusieurs mots, ou les erreurs de construction syntaxiques. Dans les deux cas, ces erreurs entraînent l'impossibilité pour l'analyseur de construire une structure.

Les erreurs de type 1 sont typiquement les erreurs d'accord en genre et en nombre à l'intérieur du groupe nominal, les erreurs d'accord en nombre et en personne entre le verbe et son sujet :

"le représentante des agriculteurs..."
"ils ferons ce qu'ils voudront."

Mais suivant le type d'analyse effectuée, de telles erreurs peuvent se traduire par des inversions de mots : si les deux pronoms "le" et "lui" ne sont distingués que par une variable indiquant leur fonction syntaxique, par exemple "cod" pour "le" et "dat" pour lui, l'erreur dans la phrase "je le lui donne" est une erreur portant sur les variables.

Les erreurs de type 2 peuvent être classées en deux catégories :

- les erreurs d'édition : par analogie avec les erreurs de niveau lexical, définie par des opérations d'édition sur les chaînes de caractères, on peut définir des opérations d'édition sur les chaînes de mots :
  effacement d'un mot (D)
  insertion d'un mot (I)
  substitution d'un mot par un autre (S)
  transposition de deux mots adjacents (T)
  Exemples :
  (D) : "je vais Paris" au lieu de "je vais à Paris"
  (I) : "cette technique pallie à la défaillance..." au lieu de "cette technique pallie la défaillance..."
  (S) : "je en vais pas à Paris" au lieu de "je ne vais pas à Paris"
  (T) : "je le lui donne" au lieu de "je lui le donne"
- les erreurs de construction en général.
  Exemples :
  "j'ai à Paris été" au lieu de "je suis allé à Paris"
  "j"ai monté sur l'échelle" au lieu de "je suis monté sur l'échelle"

La méthode de traitement de telles phrases dépend en grande partie des objectifs recherchés dans ce traitement. Nous distinguons deux grandes classes d'applications : celles qui visent à obtenir une interprétation sémantique de la phrase même si celle-ci est incorrecte et celles qui visent à corriger l'erreur.

La première classe concerne les systèmes où la langue naturelle est utilisée comme outil de communication entre l'homme et la machine (interrogation de banques de données, commande de robot, ...). Dans ces applications la forme compte moins que le fond et l'accent est mis sur les contraintes sémantiques associées aux mots. Les travaux de Carbonnell et Hayes [CARBONNELL 81, 83, 84] [HAYES 80, 86] utilisent une grammaire de cas où le correcteur tente de compléter la structure de cas induite par le verbe. Dans le système NOMAD [GRANGER 83], la méthode de correction utilise des prévisions syntaxiques et sémantiques mais surtout la connaissance de l'environnement d'utilisation de la langue (connaissances pragmatiques).

La deuxième classe concerne les systèmes où la langue est la matière première à manipuler (traitements de textes, enseignement assisté par ordinateur). Le but est ici non seulement de détecter les erreurs mais aussi de corriger l'erreur ou au moins de proposer des corrections. Comme nous nous rangeons plutôt dans cette classe, nous allons détailler un peu ses caractéristiques en nous appuyant sur des exemples de systèmes.

6.1.2. Stratégies de correction

Le système ESOPE [ST-DIZIER 84] traite les fautes d'accord en genre et en nombre dans le groupe nominal et l'accord du verbe avec son sujet. En cas d'échec de l'analyse, ESOPE reprend l'analyse en inhibant les contraintes d'accord. En présence d'une erreur il applique des règles de correction automatique fixées a priori du style : "en cas d'incompatibilité de genre entre le nom et l'article, le nom impose son genre à l'article".
Les systèmes EPISTLE [HEIDORN 82] et CRITIQUE [RICHARDSON 85] traitent également des fautes d'accord mais aussi des problèmes de construction tels que le placement des pronoms, les formes verbales avec auxiliaire, ... ainsi que des problèmes de style (phrases trop longues ou trop complexes, double négation, répétitions, ...). EPISTLE est limité au domaine particulier des lettres d'affaires, mais il fournit, grâce à un générateur, une version du texte avec critiques et observations.
Citons enfin le système de Richard et Lapalme [LAPALME 86], spécialisé dans les fautes d'accord et plus particulièrement dans le délicat problème de la correction des accords des participes passés en français. Leur étude a montré l'importance d'informations plus détaillées que le simple genre et nombre de chaque mot : ils considèrent notamment comme indispensables les traits sémantiques des sujets et compléments acceptables pour le verbe ainsi que l'espèce du verbe.

Une première remarque concernant cette classe est que si on veut pouvoir traiter automatiquement les erreurs de type 2 (édition ou construction), on doit disposer de la connaissance pour le faire. Ceci implique que la grammaire utilisée ne doit pas pouvoir engendrer plus de phrases que les phrases correctes de la langue pour laquelle elle a été conçue. Il est en effet connu qu'une grammaire conçue pour l'analyse de textes corrects admet plus de phrases que les phrases de la langue. Ce n'est pas gênant si on suppose que les phrases d'entrée sont correctes et que le rôle de la grammaire est simplement de produire une structure pour cette phrase. Prenons un exemple simple.
Exemple :

Soit la grammaire :

GN → Det + N | Det + GA + N

GA → Adj + GA | Adj

Cette grammaire produit des structures pour des groupes nominaux tels que "le bébé", "le petit bébé", "le gentil petit bébé". Elle est donc suffisante pour l'analyse des déterminants et adjectifs situés à gauche du nom. Mais elle accepte aussi des groupes comme "le gentil petit blond bébé" ou "le gentil petit blond rose bébé".

L'utilisation d'une stratégie en deux étapes (une étape d'analyse classique puis **en cas d'échec** un étape d'analyse avec les contraintes inhibées augmentée de

méthodes de correction) suppose donc la disponibilité d'une grammaire de la langue qui reconnaisse les phrases de la langue mais uniquement celles-ci. L'écriture d'une telle grammaire pour toute une langue étant extrêmement difficile, cette stratégie ne semble adaptée qu'à des domaines restreints.

Une autre stratégie possible consiste à utiliser un formalisme grammatical qui permette d'expliciter le traitement des exceptions et des erreurs. On procède à une analyse classique dans laquelle les exceptions sont ignorées et en cas d'échec, on revient en arrière en validant le traitement des exceptions et des erreurs jusqu'à obtenir un déblocage. C'est la méthode proposée par [FOUQUERE 88] et [GOESER 90]. Cette stratégie impose d'une part la possibilité d'une analyse bidirectionnelle, d'autre part l'écriture dans la grammaire de toutes les exceptions. Il faut donc prévoir les exceptions et les erreurs de manière exhaustive. De plus, cette méthode peut avoir des effets de bord masquant certaines interprétations. Ainsi, si on considère, par exemple, la phrase "la maison de l'oncle que nous avons vu", on la trouve parfaitement correcte si on utilise le genre et le nombre de "vu" pour rattacher la relative à "oncle". On masque ainsi une possible erreur d'accord dont la correction aurait permis le rattachement de la relative à "maison".

Quoiqu'il en soit, il nous paraît extrêmement difficile à court terme d'espérer corriger les erreurs de type 2. D'autre part, on peut faire les deux remarques suivantes :

- d'après certaines études [EMIRKANIAN 88a, 88b], les erreurs de ce type sont relativement peu fréquentes en français (du moins en français écrit par des Français) ;
- la correction entièrement automatique de ce type d'erreur supposerait une compréhension approfondie de la phrase et donc une connaissance de son contexte (textuel mais aussi réel).
  Exemple : (emprunté à [FOUQUERE 88])
  "l'avion, il ne se sent pas bien" peut être corrigé d'au moins deux manières :
  "dans l'avion, il ne se sent pas bien" (oubli de mot)
  "l'avion, il ne se vend pas bien" (substitution de "vend" par "sent")

Il semble donc raisonnable de n'envisager que la détection de ce type d'erreur, remettant à l'utilisateur le travail de la correction. Si on veut envisager des applications telles que l'enseignement assisté du français, il faut quand même prévoir la possibilité de fournir à l'utilisateur un minimum d'explications concernant la nature de l'erreur ou au moins la définition des mots employés. Dans l'exemple ci dessus, on pourrait donner un message du style : "un avion (machine volante) ne peut pas se sentir, seul un être vivant peut se sentir".

Nous proposons d'utiliser une stratégie en deux étapes distinctes :

1 - procéder à la construction de structures syntaxiques en n'utilisant que la catégorie lexicale du mot et les informations sémantiques qui lui sont associées. On pourra également utiliser certaines informations syntaxiques (régime de complémentation des verbes par exemple), mais pas du tout les informations morphologiques (genre, nombre, personne) qu'on se propose de vérifier dans la deuxième étape. L'analyseur ne remettra donc pas en cause la validité d'un mot mais simplement la validité de ses propriétés morphologiques. Il

devra être robuste, c'est-à-dire qu'il devra le plus souvent possible produire une structure même si cela risque de multiplier les interprétations possibles d'une même phrase.

2- faire passer les structures produites dans un vérificateur syntaxique chargé de les valider ou de les invalider en contrôlant la compatibilité des variables morphologiques de chaque mot avec celles de ses voisins dans la structure. Nos objectifs sont limités dans un premier temps au contrôle des accords en genre, nombre, personne à l'intérieur d'une phrase.

Cette stratégie risque de pêcher par une certaine inefficacité en terme de performance car l'inhibition des contrôles morphologiques en cours d'analyse va multiplier les ambiguïtés et contraindre à la construction de plusieurs structures pour une même phrase. Par contre, la séparation en deux étapes permet la mise au point d'heuristiques de correction plus fines sur les structures produites. On pourra, par exemple, vérifier toutes les structures et ne lancer de corrections que si on n'en trouve aucune de correcte, ou bien attribuer à chaque structure un certain facteur de confiance en fonction du nombre et du type des erreurs trouvées et ne soumettre à la correction que les deux ou trois structures ayant le facteur de confiance le plus grand.

Concernant la correction d'une structure, il est préférable de posséder la structure complète car cela permet également la mise en œuvre d'heuristiques plus fines. Ainsi, si on détecte une incompatibilité d'accord entre l'article et le nom, on peut choisir d'imposer les variables du nom ou bien choisir de consulter les mots voisins (adjectifs, voire verbe si le nom est sujet) et déterminer quelles sont les valeurs de variables les plus probables.

Exemple :

Il n'est pas logique d'imposer le nombre du nom à l'article dans une phrase comme : "les belles voiture anciennes sont très bien cotées.". Un tel choix impose en effet de modifier non seulement le nombre de l'article et des adjectifs, mais également le nombre du verbe et de l'attribut du sujet.

6.2. Hiérarchie de catégories et robustesse

Bien que nous ayons pour l'instant limité nos objectifs à la détection et à la correction des erreurs d'accord, il est important de disposer d'un analyseur syntaxique robuste. Par robuste, nous entendons que l'analyseur doit pouvoir fournir un résultat pour toutes les phrases, même si ce résultat est incomplet. La condition minimale pour corriger est, en effet, d'avoir quelque chose à corriger. La robustesse de l'analyseur que nous avons présenté au chapitre 2 se situe à deux niveaux :

- c'est un transducteur qui transforme une forêt d'arbres à un nœud (les mots de la phrase initiale) en une liste de forêts ou chaque forêt est une structure syntaxique possible pour la phrase. Le résultat idéal est bien sûr une liste d'une seule forêt contenant un seul arbre : l'arbre de dépendance de la phrase. Le résultat minimal est la forêt initiale (si aucune règle ne s'applique).
- il utilise une hiérarchie de catégories qui permet le traitement de mots complètement inconnus [GENTHIAL 90].

Le premier niveau a été largement détaillé au chapitre 2, nous allons ici préciser plus longuement comment les mots inconnus peuvent être manipulés.

Nous classons les mots inconnus en trois catégories :

1- les mots corrects mais qui n'appartiennent pas au lexique (notamment les noms propres) ;
2- les mots incorrects mais qui peuvent se voir attribuer une interprétation par un lecteur humain (néologismes comme "solutionner", "dénationaliser",...) ;
3- les mots corrects mal orthographiés et qui n'ont pu être corrigés par les techniques de corrections présentées au chapitre 6.

Les deux premières catégories peuvent faire l'objet d'un traitement spécial : on peut imaginer de mettre en place un module supplémentaire basé sur des techniques d'analyse morphologique sans dictionnaire comme celle de [VERGNES 90] ou de [PALMER 90]. Un tel module tente, en utilisant un ensemble réduit de règles morphologiques, de prédire la catégorie et les variables morphologiques d'un mot en fonction de sa forme.

Exemple de règles :

Les mots finissant par "isme" sont des substantifs masculin singulier.
Les mots finissant en "aux" sont des substantifs pluriel.

Ce type de technique ne permet pas d'obtenir des résultats dans tous les cas (que dire par exemple du mot inconnu "sate"), mais quand elle en obtient cela limite la combinatoire au niveau syntaxique.

Il reste donc toujours des mots inconnus pour lesquels il est impossible de faire des hypothèses. Pour que de tels mots ne bloquent pas l'analyse en cours, on peut mettre à contribution la hiérarchie des catégories et leur affecter la catégorie la plus générale : `CLS` (la figure 6.1 donne un exemple simple de hiérarchie).

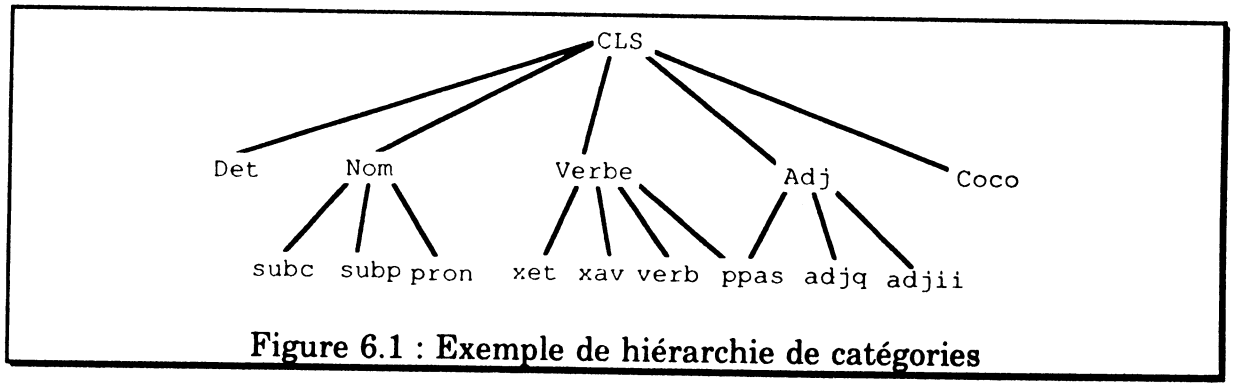


Figure 6.1 : Exemple de hiérarchie de catégories

L'information attachée au mot est donc le Ψ-terme `UL(cat => CLS)`. Comme `CLS` est la catégorie la plus générale, elle s'unifie avec n'importe quelle catégorie de la hiérarchie et donc n'importe quelle règle peut lui être appliquée. L'application d'une règle à un mot inconnu sera seulement subordonnée au contexte immédiat de ce mot.

Exemple :

On se donne la grammaire très simple (basée sur la hiérarchie de la figure 6.1) :

```
Det_Nom [
    (1:{Det}, $A:{Adj}, ()2:{Nom}) => ( ( 1, $A ) 2 ) ]
```

```
Adj_Nom [
    (1:{Adj}, (0, $A:{Adj}) 2:{Nom}) => ( ( 1, $A ) 2 ) ]
Nom_Adj [
    (1:{Nom}?{Nom}, 2:{Adj}) => ( 1 ( 2 ) ) ]
Nom_Verbe [
    (1:{Nom}, () 2:{Verbe}) => ( ( 1 ) 2 ) ]
Verbe_Nom [
    (1:{Verbe} (), 2:{Nom}) => ( 1 ( 2 ) ) ]
```

A la phrase "le chne mange la soupe", on associe la séquence :
`Det - CLS - verb - Det - subc`
Le transducteur ne pourra pas appliquer de règles avant l'introduction de l'arbre portant `CLS`, l'application de la règle `Det_Nom` produit alors l'arbre `(Det)Nom` et on a après introduction du verbe la forêt :
`(Det)Nom - verb`
Une erreur sur le déterminant ("li chien mange la soupe") aurait permis l'application des trois règles `Det_Nom`, `Adj_Nom` et `Verbe_Nom` dès l'introduction du subc, produisant après introduction du verbe :
`(Det)subc - verb`
`(Adj)subc - verb`
`Verbe(subc) - verb`
Seules les deux premières hypothèses donneront un résultat avec cette grammaire car dans la dernière il ne sera pas possible de lier les deux verbes.

On peut remarquer que l'algorithme du transducteur privilégie le contexte local c'est-à-dire les mots les plus proches. En effet dès qu'une règle est applicable, elle est appliquée avec pour effet de transformer la catégorie `CLS` en une catégorie plus précise. C'est cette transformation qui permet de créer les hypothèses mais elle a l'inconvénient de masquer des hypothèses qui auraient pu être faites avec des règles de portée plus grande.

Exemple :

On reprend l'exemple ci-dessus avec la phrase "le bo chien mange...", correspondant à la séquence :
`Det - CLS - subc - verb`
Le transducteur appliquera la règle `Det_Nom` dès l'introduction de CLS produisant alors l'arbre `(Det)Nom` et on a après introduction du nom la forêt :
`(Det)Nom - subc`
Cette forêt ne produira pas de résultat alors que l'examen d'un mot de plus de la séquence, c'est-à-dire l'application de la grammaire à la forêt `Det - CLS - subc` aurait produit, grâce aux règles Det_Nom (3 fois), Verbe_Nom et Adj_Nom, les forêts :

`(Det)Nom - subc`	Règle Det_Nom, CLS = Nom
`Det - (Det)subc`	Règle Det_Nom, CLS = Det
`(Det, Adj)subc`	Règle Det_Nom, CLS = Adj
`Det - Verbe(subc)`	Règle Verbe_Nom, CLS = Verbe
`Det - (Adj)subc`	Règle Adj_Nom, CLS = Adj

Bien qu'on ait multiplié les hypothèses incorrectes, on a la bonne hypothèse dans la liste.

L'exploitation de la hiérarchie de catégories évite donc un blocage de l'analyse puisqu'on peut fournir une interprétation d'un mot inconnu. Cette hiérarchie

confère donc au système une certaine tolérance qui présente l'avantage d'être **implicite** : il n'est pas nécessaire dans l'écriture des règles de prévoir cette tolérance. Mais comme on vient de le voir dans les exemples, le fonctionnement du transducteur par rapport aux mots inconnus n'est pas satisfaisant. Pour l'améliorer sans remettre en cause la transparence de la tolérance (son caractère implicite), on peut envisager d'effectuer un traitement spécial pour les mots complètement inconnus (facilement reconnaissables à la catégorie CLS). On pourrait par exemple conserver le mot inconnu actif pendant un certain nombre de cycles de l'analyseur. Ainsi sur l'exemple ci-dessus, après avoir appliqué la règle Det_Nom à la séquence Det - CLS, on introduit le nom en conservant quand même le mot inconnu actif, on a alors les forêts :

```
(Det)Nom - subc
Det - CLS - subc
```

Le risque est alors bien évidemment de produire plusieurs fois la même hypothèse. Pour l'éviter, on peut tenir compte des hypothèses faites et chaque fois qu'une règle s'applique, réduire l'ensemble des catégories potentielles du mot inconnu. Ainsi sur l'exemple, on n'utilisera pas CLS mais une catégorie qui ne recouvre pas Nom mais qui recouvre toutes les autres. En terme d'ensembles, on prendra le complémentaire de Nom dans CLS. Avec la hiérarchie de la figure 6.1, on aura pour l'exemple les deux forêts :

```
(Det)Nom - subc
Det - {Det, Verbe, Adj, Coco}  - subc
```

On notera qu'avec cette même hiérarchie le complémentaire de Verbe n'est pas l'ensemble {Det, Nom, Adj, Coco} mais l'ensemble {Det, Nom, adjq, adji, Coco}. Il faut en effet éliminer de l'ensemble Adj la catégorie ppas qui appartient à l'ensemble Verbe.

Nous nous sommes limités jusqu'ici à la manipulation de la catégorie des mots inconnus. Mais le principe de l'unification et la sémantique de l'opérateur d'affectation (définie au chapitre 2) permet aussi d'exploiter les règles pour calculer des hypothèses sur les autres informations portées par les arbres. Considérons une règle plus complexe comme la règle attachant son sujet à un verbe :

```
sujet [ (1 : {Nom}, 2:{Verbe})
/Unif(1, 2.syn.sujet)/
=>
( ( 1 ) 2 ) ;
AFFECT(2.syn.sujet, Unif(1, 2.syn.sujet)) ;
AFFECT(1, 2.syn.sujet)
]
```

Considérons l'application de cette règle à la forêt (Det)Nom - verb, où le nœud Nom est un mot inconnu dont la catégorie a été calculée par la règle Det_Nom. On suppose que le premier arbre (Det)Nom porte le Ψ-terme :

```
UL(cat => Nom)
```

et que le deuxième porte :

```
UL(cat => verb;
   syn => proposition(
     prédicat => MANGER;
     sujet => UL(syn => groupe_nominal;
                 sém => @S : ANIME)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => @S;patient => @O))
```

Les deux affectations de la règle sujet vont transformer ces deux Ψ-termes respectivement en :

```
UL(cat => Nom ;
   syn => groupe_nominal;
   sém => ANIME)
```

et :

```
UL(cat => verb;
   syn => proposition(
     prédicat => MANGER;
     sujet => UL(cat => Nom ;
                 syn => groupe_nominal;
                 sém => @S : ANIME)
     objet => UL(syn => groupe_nominal;
                 sém => @O : COMESTIBLE))
   sém => MANGER(agent => @S;patient => @O)))
```

L'application de la règle a donc permis de faire une hypothèse sur les traits `syn` et `sém` du mot inconnu. Il est clair que ce mécanisme n'est vraiment efficace que si le mot inconnu est une "feuille" du Ψ-terme, c'est-à-dire s'il porte peu d'informations sur les liens qu'il a avec les autres mots. En effet, ce sont ces informations qui produisent les hypothèses. Si on applique la règle `sujet` à une séquence où c'est le verbe qui est inconnu, par exemple `Nom - CLS`, avec pour `Nom : UL(cat => Nom; sém => CANINE)`, on obtiendra pour le mot inconnu le Ψ-terme :

```
UL(cat => verb;
   syn => _TOUT_(
     sujet => UL(cat => Nom ;
                 sém => CANINE)))
```

Même si on arrive à produire une hypothèse du même genre pour le trait `syn.objet`, on ne pourra retrouver ni le trait `sém` du verbe ni les liens entre les fonctions syntaxiques et les relations sémantiques.

En conclusion, on peut remarquer que si la manipulation de mots complètement inconnus est possible, l'ambiguïté infinie d'un tel mot conduit à des traitements très coûteux si on veut être sûr d'obtenir la bonne hypothèse. Il nous semble donc préférable dans un système de détection/correction de privilégier l'utilisation des techniques de correction lexicale afin d'obtenir le maximum d'informations sur le mot.

6.3. Vérification syntaxique

Une fois obtenue la (ou les) structure(s) syntaxique(s) d'une phrase, on doit vérifier la compatibilité des informations morphologiques portées par les nœuds. Pour ce faire, il faut d'une part déterminer quels mots doivent être compatibles et d'autre part vérifier pour ces mots la compatibilité. Nous allons présenter dans ce paragraphe le prototype de vérificateur syntaxique développé dans notre équipe par Véra Lucia Strube de Lima [STRUBE DE LIMA 90]. Nous insisterons sur les problèmes rencontrés et sur les solutions apportées. Nous essayerons notamment de montrer comment l'analyseur et le modèle de représentation de la connaissance présentés dans la première partie s'intègrent dans ces solutions.

6.3.1. Le prototype CORSYNT

Le vérificateur/correcteur syntaxique (baptisé CORSYNT pour CORrecteur SYNTaxique) est composé de trois modules :

- un analyseur morpho-syntaxique ;
- un vérificateur syntaxique ;
- un générateur morphologique.

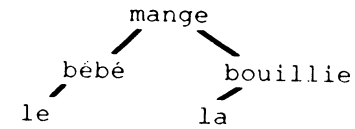
L'analyseur morpho-syntaxique regroupe l'analyseur morphologique et le constructeur de structures de dépendances de PILAF. Il est chargé de produire, pour une chaîne de caractères donnée, un (ou plusieurs) arbres de dépendances. Chaque nœud de l'arbre porte une unité lexicale à laquelle sont associées des variables morphologiques.

Le générateur morphologique est celui de PILAF (analyseur inversé) ; il produit, pour une base et un ensemble de variables morphologiques données, toutes les formes dérivables à partir de la base qui possèdent l'ensemble de variables.

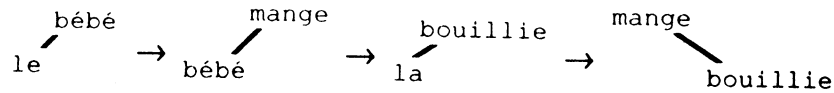
Nous décrivons le fonctionnement du vérificateur puis la manière dont les trois modules ont été intégrés.

6.3.1.1. Principe de la vérification syntaxique

L'entrée du vérificateur est un arbre de dépendances décoré par des variables morphologiques. Le vérificateur parcours l'arbre en ordre post-fixé gauche en examinant toutes les paires de nœuds (dépendant gauche, gouverneur) et (gouverneur, dépendant droit). Ainsi, avec l'arbre :



les paires de nœuds sont examinés dans l'ordre :



Ces paires sont notées (gauche * droit). Le vérificateur tente d'appliquer à chaque paire rencontrée un ensemble de *règles de vérification*.

Règles de vérification

Ces règles comportent trois parties :

- un entête qui précise la paire à laquelle la règle peut s'appliquer ;
- une partie SI qui contient des conditions d'application portant sur les variables morphologiques ;
- une partie ALORS contenant des actions qui calculent les variables morphologiques du résultat.

Le vérificateur est écrit en PROLOG et les règles sont simplement les clauses d'un prédicat PROLOG.

Exemple :

Règle de vérification de l'accord entre le déterminant et le nom :

```
regle(ms(MotGauche, deto, Gauche),
```

```
        ms(MotDroit, gnomo, Droit),
        R, Rep) :-
/* SI */
  coraccord(ms(MotGauche, deto, Gauche),
            ms(MotDroit, gnomo, Droit),
            ["GNR", "NBR"], R1, Rep)
/* ALORS */
, somme([R1, ["art", "tre"]], R).
```

Les symboles commençant par des majuscules sont des variables. Cette règle s'applique à la paire (deto, gnomo) et vérifie l'accord en genre ("GNR") et nombre ("NBR") entre les deux mots. Cette vérification (prédicat `coraccord`) produit une liste `R1` des valeurs des variables s'il y a accord et un indicateur `Rep` de succès ou d'échec. Le prédicat somme ajoute les valeurs "art" (indiquant que le nom a pris son article) et "tre" (troisième personne) à la liste R1, produisant R qui est la liste de variables résultant de l'application de la règle.

Pour une paire (gauche * droit) donnée, le vérificateur n'applique que la première règle applicable. En théorie, une telle règle existe toujours, et l'absence de règles applicables pour une paire donnée indique : soit une incohérence entre la grammaire de dépendances de l'analyseur et la grammaire du vérificateur (l'analyseur construit des structures qui ne sont pas traitées par le vérificateur), soit que la structure est incorrecte (on a parlé au §1.4 des limites de l'analyseur de PILAF). Dans les deux cas, cela conduit à l'arrêt de la vérification.

L'existence d'un règle pour toute paire suppose donc que certaines règles ne servent qu'à éviter l'arrêt de l'analyseur (ce sont des règles permettant de poursuivre le parcours de l'arbre). L'inconvénient de ce type d'algorithme est qu'il faut une règle pour chaque paire de catégorie possible, c'est-à-dire au maximum n^2 règles si on a n catégories. Pour éviter cette combinatoire, les catégories utilisées pour l'analyse ont été regroupées en classes, chaque classe étant caractérisée par un comportement similaire.

Exemples :

La classe adjo regroupe tous les adjectifs (sauf les adjectifs possessifs considérés comme des déterminants).

La classe gnomo regroupe les noms communs et propres.

Ce regroupement est effectué en post-traitement de l'analyse syntaxique.

Ce post-traitement est encore un exemple des difficultés liées à la définition d'un ensemble de catégories qui réponde à tous les besoins (analyse précise et vérification). Les notions utilisées dans la première partie peuvent fournir une solution plus souple à ce problème :

- l'utilisation d'une hiérarchie de catégorie dispense de la définition des classes (qui sont définies dans la hiérarchie) ;
- l'utilisation dans les règles d'ensembles de catégories à la place de simples catégories facilite les regroupements des phénomènes d'accord communs à plusieurs catégories.

Exemple :

On pourrait écrire, pour couvrir l'accord du nom avec les adjectifs, une règle dont l'entête serait simplement (N, A). Grâce à la hiérarchie et au

mécanisme de l'unification, on pourrait, avec cette règle, traiter aussi l'accord du nom avec le participe passé.

Dans le système CORSYNT, outre le regroupement des catégories en classes, on a défini des *méta-règles* afin de regrouper les phénomènes d'accord communs. L'objectif de ces règles est de déterminer, pour une paire (gauche * droit) quelle règle d'accord doit être appliquée et quel est le mot auquel la liste des variables résultats doivent être attachée (la liste R dans l'exemple de règle donné ci-dessus).

Exemples de méta-règle :

(adjo * **gnomo**) → (gnomo * adjo) (1)

(**gnomo** * ppaso) → (gnomo * adjo) (2)

(popel * **xeto**) → (popel * verbo) (3)

Les méta-règles (1) et (2) renvoient toutes les deux à la règle d'accord (gnomo * adjo), avec pour résultat gnomo (noté en gras). Elles modélisent le fait que l'adjectif placé avant le nom ou le participe passé placé après le nom s'accordent de la même manière. La partie droite d'une méta-règle désigne une règle d'accord du type de celle donnée ci-dessus ; on notera que l'utilisation de ces méta-règles rend indifférent l'ordre des catégories dans les règles d'accord. Une règle d'accord dont l'entête est (gnomo * adjo) doit donc s'interpréter : "règle d'accord entre le nom et l'adjectif".
La méta-règle (3) traduit le fait que l'auxiliaire être s'accorde avec un pronom personnel de la même façon qu'un verbe.

6.3.1.2. Fonctionnement de CORSYNT

Les trois modules (analyse, vérification, génération) ont été intégrés à un programme unique (le prototype CORSYNT) qui gère l'interface utilisateur et qui pilote l'analyse, la détection et la correction des erreurs. Son fonctionnement peut être décrit par le schéma d'algorithme suivant :

1 - L'utilisateur entre une phrase au clavier (chaîne de caractères)
2 - Cette phrase (supposée correcte lexicalement) est soumise à l'analyseur morpho-syntaxique qui produit une (ou plusieurs) structure(s) de dépendances décorée(s).
3 - Cette structure (la première s'il y en a plusieurs) est soumise au vérificateur syntaxique, 2 cas se présentent :
4 - La structure ne comporte pas d'erreurs, le traitement s'arrête (on a une structure correcte, on ignore les éventuelles autres).
5 - Le vérificateur détecte une erreur sur une paire (gauche * droit)
6 - Il transmet au générateur morphologique un des deux mots, considéré comme le mot à corriger, avec les valeurs des variables de l'autre mot (limitées aux variables qui ont provoqué l'erreur). Le générateur produit alors une forme compatible du mot qui est proposée à l'utilisateur. Ce dernier a alors deux choix possibles :
7 - soit il accepte la correction ; l'arbre est alors mis à jour et le traitement reprend à l'étape 3.
8 - soit il refuse la correction ; ceci implique que les accords sont en fait corrects mais que c'est la structure qui ne l'est pas. Le traitement reprend alors à l'étape 3 avec la structure suivante.

Commentaires :

Le choix du mot à corriger dans l'étape 6 est fixé à priori pour deux catégories données : en cas d'erreur entre le déterminant et le nom par exemple, c'est le nom qui est soumis à correction. Cette méthode est également celle adoptée par [LAPALME 86]. Ainsi pour "le chatte" la correction proposée est "la chatte". Nous revenons plus loin de manière détaillée sur ces problèmes de stratégies de correction.

L'étape 8 peut être justifiée avec des phrases comme "la maison de l'oncle que nous avons vu". Si la structure traitée contient un rattachement de "que nous avons vu" à "maison", le système détecte une erreur d'accord qui n'en est pas une, mais qui est une erreur de construction. La structure suivante, rattachant "que nous avons vu" à "oncle", sera la bonne.

Signalons enfin le caractère indispensable du générateur morphologique dès qu'on veut dépasser le stade de la simple détection et effectuer des corrections, soit automatiques, soit soumises au contrôle de l'utilisateur.

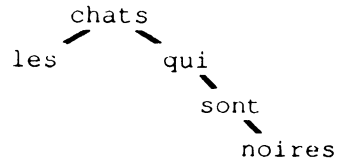
6.3.1.3. Problèmes

L'étude et la réalisation de ce prototype a rencontré un certain nombre de problèmes qui ont été parfois résolus mais qui ont permis de souligner les caractéristiques essentielles d'un système de détection/correction au niveau syntaxique.

D'un point de vue informatique d'abord, l'algorithme utilisé pour la vérification, qui a l'avantage d'être facile à mettre en œuvre, s'est montré limité pour le traitement de certaines structures. Ainsi, la vérification de l'accord entre le verbe d'une relative et son sujet (antécédent du pronom) est difficile avec la structure construite par l'analyseur de PILAF.

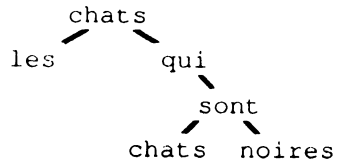
Exemple :

Pour la phrase "les chats qui sont noires", l'analyseur construit la structure :



et l'ordre de parcours (postfixé) examine les paires (sont * noires) et (qui * sont) avant la paire (chats * qui) et donc avant de connaître les variables morphologiques de "chats"

[STRUBE DE LIMA 90] propose de modifier la structure pour insérer une copie de l'antécédent du pronom relatif en position de sujet du verbe de la relative, on aurait alors la structure :



Cette modification de structure a été intégrée dans CORSYNT au post-traitement syntaxique (déjà chargé de regrouper les catégories en classes).

Ce problème aurait également pu être résolu en modifiant le mécanisme de transmission des variables, mais il montre le besoin de consulter le contexte de la paire (gauche * droit) à corriger. Ce besoin devient une nécessité si on envisage de mettre en œuvre des stratégies de correction assez fine (voir paragraphe suivant).

D'un point de vue linguistique, l'absence de certaines informations syntaxiques et sémantiques rend difficile voire impossible la détection de certaines erreurs. [STRUBE DE LIMA 90] suggère notamment :

- d'adopter une catégorisation plus fine afin de distinguer par exemple les pronoms relatifs qui induisent un accord (comme "que" dans "la tarte aux fraises que j'ai mangée") et ceux qui n'en induisent pas (comme "dont" dans "la réunion dont j'ai parlé").
- de stocker plus d'informations sur les verbes, comme la transitivité, la possibilité d'accepter une complétive, la situation "verbe d'état" (possédant un attribut qui s'accorde avec le sujet), etc...
- d'utiliser un minimum d'informations sémantiques (trait sémantique des noms, traits sémantiques du sujet et des compléments des verbes) afin de régler par exemple le problème de l'accord du participe passé suivi d'un infinitif dans une relative, qui dépend de la capacité du complément d'objet à effectuer l'action décrite par l'infinitif. Comparer :
  "Les chansons que j'ai entendu chanter"
  "Les chanteuses que j'ai entendues chanter"

6.3.2. Vers un vérificateur plus souple

Nous faisons dans ce paragraphe des propositions en vue d'obtenir un vérificateur syntaxique plus souple et plus performant.

Les problèmes linguistiques évoqués ci-dessus seront résolus grâce à l'utilisation de l'analyseur décrit dans la première partie de la thèse. Le modèle de représentation de la connaissance permet en effet de coder ce type d'information et l'utilisation de la hiérarchie de catégories facilite la définition de catégories très fines (dans les verbes on peut distinguer les verbes d'état, les verbes auxiliaires) sans que cela n'ait d'influence directe sur les règles d'analyse (on peut par exemple utiliser pour tous les verbes une règle générale de rattachement du sujet).

Il nous semble important de changer la méthode de parcours de l'arbre ainsi que l'unité minimale de correction. En effet, une paire de catégories complètement isolée de son contexte constitue une granularité trop faible qui rend impossible toute stratégie globale de correction. Bien qu'il nous semble impossible d'envisager une correction entièrement automatique à court terme, il nous paraît important de proposer à l'utilisateur non pas toutes les corrections possibles en vrac, mais ces mêmes corrections triées par ordre de vraisemblance décroissant. Un tel ordre suppose la mise en œuvre de mécanismes d'évaluation attribuant à chaque proposition un facteur de confiance.

[VERONIS 88c] considère par exemple les critères suivants :

- le nombre d'erreurs : corriger "le petits chien" en "le petit chien" paraît plus logique que de le corriger en "les petits chiens". On notera que ce critère nécessite la correction de plusieurs mots à la fois.
- le type de l'erreur : Véronis observe par exemple que les marques morpho-syntaxiques muettes (comme le "s" du pluriel) sont souvent oubliées mais rarement ajoutées là où elles ne devraient pas être. Il observe également que la prononciation joue un rôle important et qu'il vaut mieux privilégier les corrections qui ne modifient pas la prononciation : on corrigera "deux chiennes dressés" plutôt en "deux chiennes dressées" qu'en "deux chiens dressés"[1].

Il nous paraît donc préférable, même si c'est plus difficile à mettre en œuvre, de corriger des groupes de mots plutôt que de simple paires. On pourra utiliser des règles de vérification/correction, comme c'est le cas dans CORSYNT mais avec un langage de sélection/modification plus proche de celui défini pour l'écriture des règles de construction (voir §2.1.2).

Exemple :

On pourrait écrire des règles du type :

```
Groupe_nominal [(1:Det, $F:Adj) 2:Nom ($G:Adj)
      =>
      Correction([1, $F, 2, $G].[GNR, NBR]) ]
Attribut [(1:Nom) 2:xet (3:Adj)
      =>
      Correction([1, 3].[GNR, NBR], [1, 2].[NBR, PERS]) ]
```

L'opérateur `Correction` mettrait en œuvre la détection et des techniques de correction qui pourrait intégrer des critères heuristiques. On notera que l'application de telles règles à tout un arbre nécessite un algorithme d'instanciation des schémas, qui, contrairement à celui utilisé pour construire les structures, doit chercher les occurrences des schémas à tous les niveaux de l'arbre et pas seulement à la racine.

L'inhibition des contraintes d'accord en cours d'analyse va de toute évidence multiplier les ambiguïtés structurales. Il nous paraît important d'en tenir compte pendant la détection et surtout pendant la correction. On ne considérerait plus comme résultat d'analyse chaque structure indépendamment mais la liste de toutes les structures. La correction doit s'appuyer sur toutes les alternatives produites : il est en effet possible qu'une de ces alternatives soit la structure correcte de la phrase.

On pourrait se contenter, pour les phrases ambiguës, d'éliminer toutes les structures incorrectes pour ne garder que celles qui ne présentent pas d'erreurs d'accord. Mais on retombe alors dans le travers des systèmes qui utilisent les contraintes d'accord en cours d'analyse et qui ne lancent de détection/correction qu'en cas d'échec.

---

[1] Un tel critère peut facilement être mis en oeuvre dans un système intégrant à la fois la vérification syntaxique et des techniques de correction lexicales basées sur la phonétique comme celle décrite au chapitre 5.

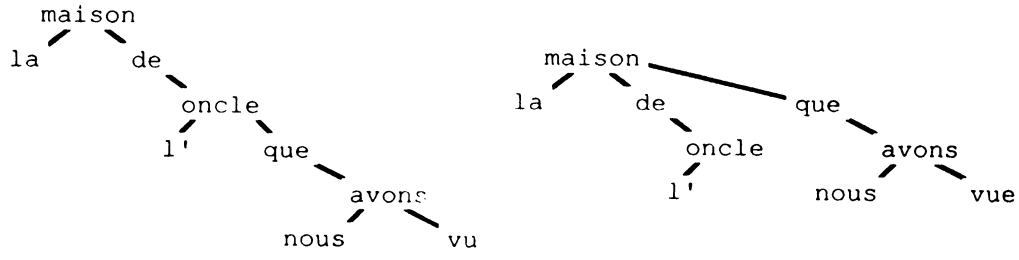
Nous suggérons d'utiliser là encore une stratégie globale :

1- vérifier toutes les structures produites et pour chaque erreur rencontrée choisir la correction ayant le facteur de confiance le plus grand et appliquer cette correction. On cumulera les facteurs de confiance à l'intérieur d'une même structure afin d'obtenir un facteur de confiance pour la structure globale (qui sera maximal si la structure est correcte).

2- soumettre à l'utilisateur les structures obtenues triées par facteur de confiance décroissant pour lui demander de choisir. On pourra également lui donner la possibilité, pour la structure choisie, de remettre en cause les corrections locales effectuées à l'étape 1.

Exemple :

Si on reprend l'exemple de "la maison de l'oncle que nous avons vu", l'utilisateur se verra proposer les deux structures :



Cette méthode présente l'avantage de cumuler deux traitements : la correction d'erreurs d'accord mais aussi la désambiguïsation interactive. C'est-à-dire que dans une phrase ambiguë mais sans erreur ("la belle ferme le voile"), l'ambiguïté sera soumise à l'utilisateur. Le plus simple est de lui soumettre les structures telles quelles mais l'idéal en terme d'ergonomie est bien sûr de disposer d'un système capable de faire des paraphrases (un générateur syntaxique "intelligent") pour discriminer les diverses interprétations.

Exemple :

Les deux interprétations de "la maison de l'oncle que nous avons vu" pourraient être paraphrasées :

"Nous avons vu la maison de l'oncle"

"La maison appartient à l'oncle que nous avons vu"

7

Propositions pour un système complet

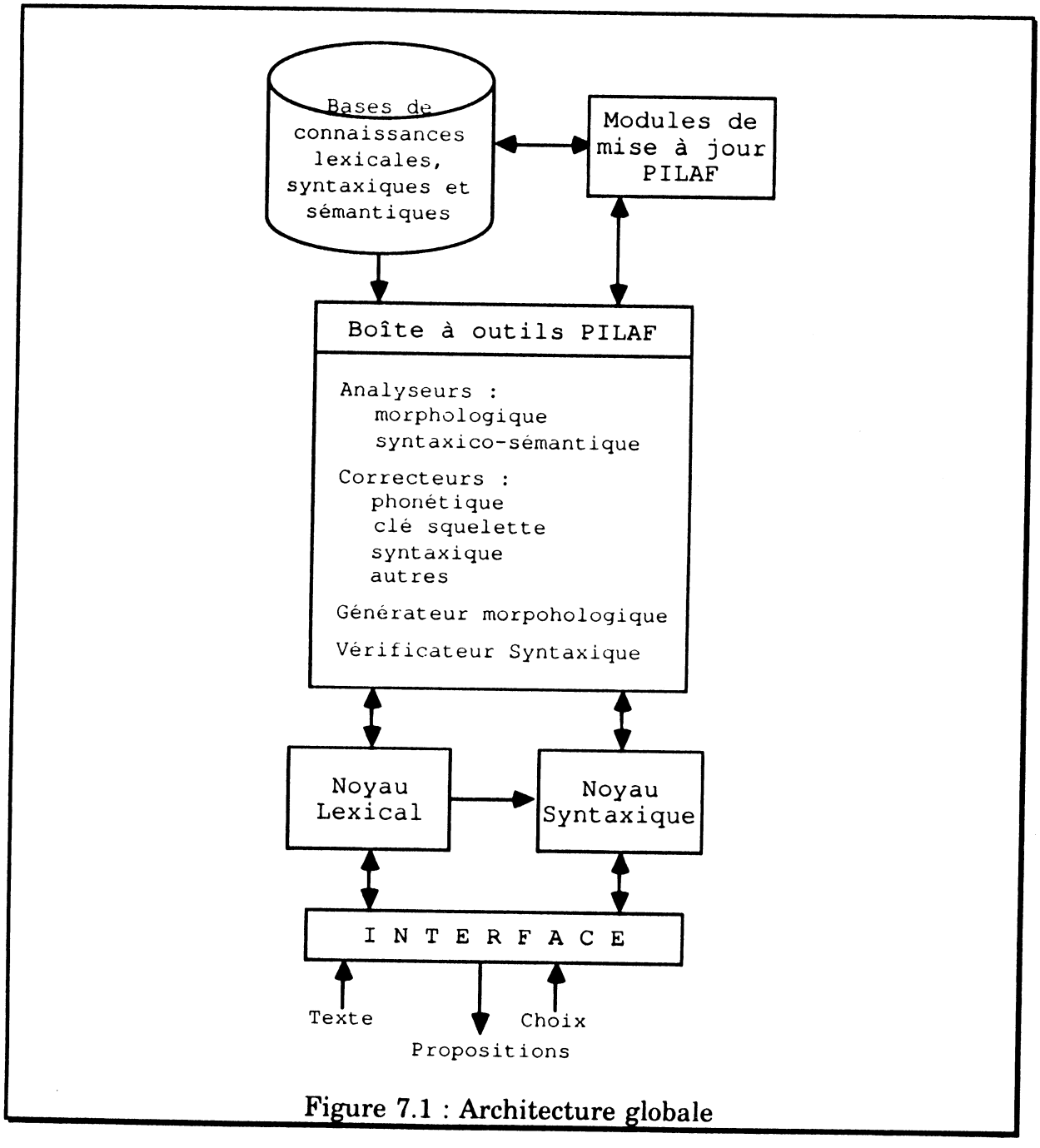


Figure 7.1 : Architecture globale

Nous proposons dans ce chapitre une architecture pour un système complet de détection et de correction d'erreurs qui intègre les techniques de correction lexicales présentées au chapitre 5 et les techniques de vérification et correction syntaxiques présentées au chapitre 6.

7.1. Architecture

L'objectif à long terme de l'équipe TRILAN est la construction d'une boîte à outils pour le traitement automatique du français. Le système proposé tient compte de cet objectif en séparant au maximum les traitements sur la langue des traitements propres à une application particulière. Il s'ensuit une architecture très modulaire dont une version grossière est donnée figure 7.1.

La *boîte à outils PILAF* (Procédures Interactives Linguistiques Appliquées au Français) est composée de modules logiciels de bas niveau exploitant la base de connaissances linguistiques. Ces modules sont au maximum indépendants les uns des autres mais aussi indépendants d'une application particulière. Cette caractéristique permet de les extraire de la boîte à outils pour les utiliser dans de nombreuses applications autres que la détection/correction (voir §7.2.2).

Les *modules de mise à jour* sont chargés de la maintenance de la base de connaissances. Ils utilisent les modules de base pour effectuer de manière automatique un certain nombre de mises à jour (voir §7.2.1).

Pour chaque niveau de correction (lexical et syntaxique), les modules sont exploités par un *noyau* qui assure le contrôle de haut niveau (stratégies de correction et communication avec l'utilisateur). Là encore, la relative indépendance des deux permet de les utiliser séparément (on pourra notamment n'utiliser que les outils de niveau lexical).

L'*interface* assure la communication avec l'utilisateur (entrée du texte, visualisation des propositions de corrections et saisie des choix éventuels). Elle est séparée des noyaux dans un souci de portabilité. L'idée est de rendre les noyaux et la boîte à outils portables sur le plus grand nombre de machines possibles sans modifications. Ainsi le portage du système complet n'impose que la réécriture de l'interface.

7.1.1. Le niveau lexical

La figure 7.2 donne l'architecture détaillée du niveau lexical. Le noyau reçoit du texte (chaîne de caractères) qui est transmis à l'analyseur morphologique. Ce dernier procède à l'analyse et calcule pour chaque unité lexicale reconnue un Ψ-terme qui contient toute l'information morphologique, syntaxique et sémantique associée à cette unité. Les Ψ-termes sont transmis en séquence au constructeur de structures de dépendances (voir §7.1.2.).

L'analyseur morphologique exploite une base de connaissances lexicales accessible par un dictionnaire de racines et désinences. L'assemblage des divers éléments d'une unité lexicale est contrôlé par les règles d'une grammaire à validations et saturations (voir chapitre 5).

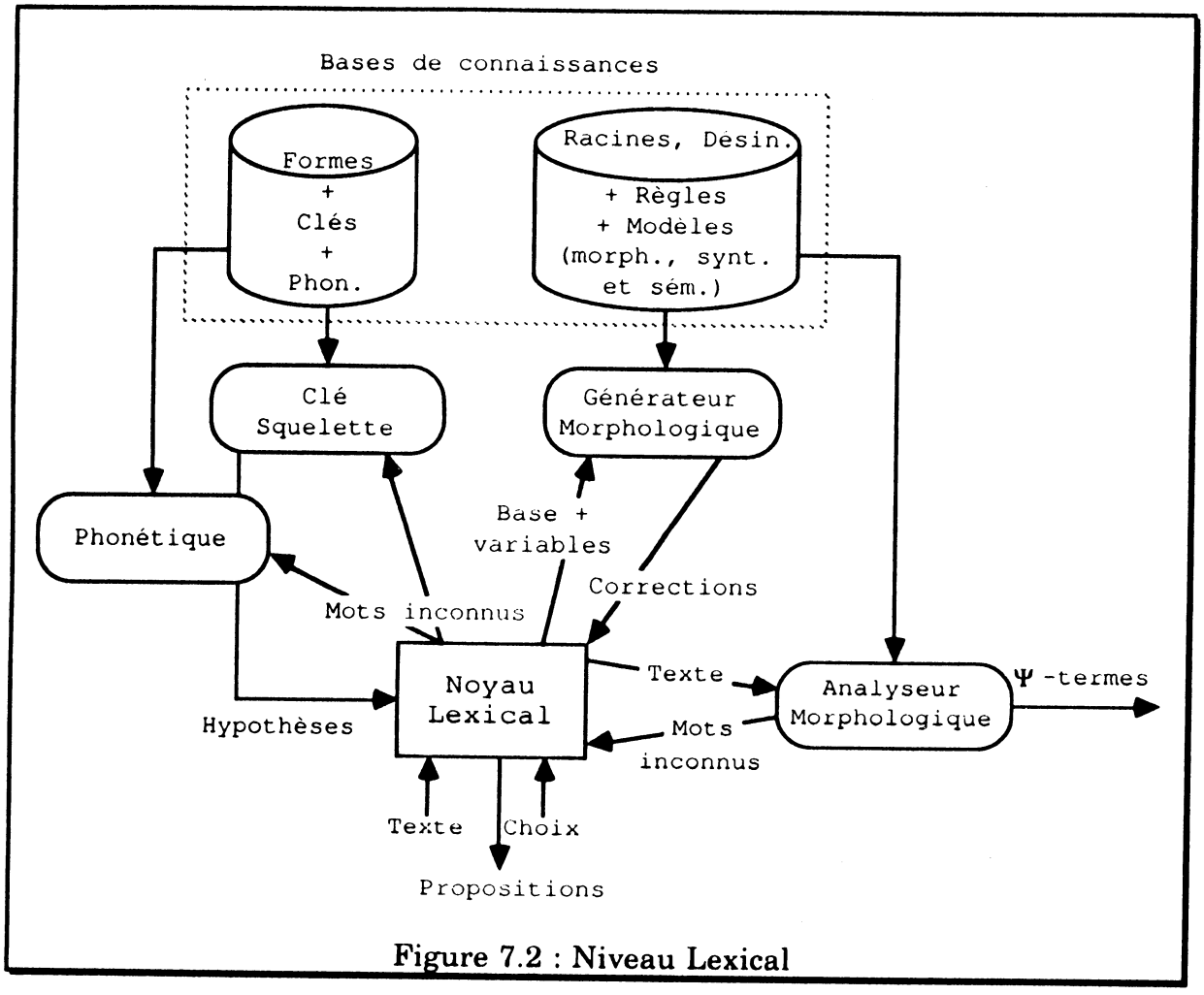


Figure 7.2 : Niveau Lexical

En cas d'échec de l'analyse, le segment de texte non reconnu, supposé incorrect, est transmis au noyau qui va se charger de calculer des corrections. Pour ce faire, il va exploiter les outils décrits au chapitre 5 :

- le générateur morphologique reçoit une racine et un ensemble de variables morphologiques (calculées par l'analyseur) et produit toutes les formes dérivables à partir de la racine et qui possèdent les mêmes variables. Il exploite pour ce faire la même base de connaissances que l'analyseur morphologique.
- le correcteur par clé squelette reçoit le segment de texte incorrect (la forme incorrecte) et produit une liste des formes ayant des clés proches et identiques à la forme incorrecte à une opération d'édition près. Ce module exploite un dictionnaire associant à chaque forme sa clé squelette.
- le correcteur phonétique reçoit également la forme incorrecte et produit les formes ayant la même représentation phonétique. Ce module exploite deux bases de règles de transcription (graphique/phonétique et phonétique/graphique) et utilise la liste des formes pour filtrer les chaînes graphiques obtenues.

Les deux derniers modules exploitent tous les deux la liste des formes mais le premier y accède par la clé squelette. Le partage de cette liste de formes par les deux modules nécessite donc une adaptation de la structure : on conserve une

liste de formes triées pour le filtrage post-phonétique et on construit une table d'accès par la clé squelette. La construction d'une telle table présente en outre l'avantage d'autoriser la factorisation des clés communes (dans le dictionnaire actuel 2/3 des clés ont plus de deux occurrences et 1/3 en ont plus de trois).

Le noyau peut utiliser une, deux où les trois techniques puis exploiter les hypothèses (formes) produites. C'est lui qui contrôle la stratégie de correction et on peut imaginer des stratégies complexes, ou des stratégies dépendantes de l'utilisateur [COURTIN 89c]. Ainsi on pourra privilégier la phonétique si le système est utilisé par un enfant ou un étranger en situation d'apprentissage de la langue (erreurs de compétences) ou au contraire privilégier la clé squelette si le système est utilisé par une personne maîtrisant bien la langue mais pas très bien le clavier (erreurs de performances). De même le noyau pourra choisir d'interroger l'utilisateur pour obtenir une correction unique ou bien de transmettre toutes les hypothèses au niveau syntaxique. Quoiqu'il en soit, les trois techniques de correction évoquées ci-dessus produisent des formes, il est donc nécessaire de passer ces formes à l'analyseur morphologique pour obtenir les informations linguistiques correspondantes, sous forme d'un Ψ-terme qui est ensuite transmis au niveau syntaxique.

7.1.2. Le niveau syntaxique

La figure 7.3 donne l'architecture détaillée du niveau syntaxique. Le module le plus important est bien sûr l'analyseur syntaxico-sémantique (le constructeur de structures de dépendances typées décrit dans la première partie). Ce module reçoit en séquence les Ψ-termes produit par l'analyseur morphologique et construit des arbres de dépendances en exploitant une base de règles de construction.

Ces arbres sont ensuite transmis au vérificateur syntaxique qui applique un ensemble de règles de vérification. Ce module attache à chaque arbre vérifié une liste des nœuds incompatibles avec l'origine de l'incompatibilité. Ces informations sont ensuite transmises au noyau.

Comme au niveau lexical, le noyau syntaxique est chargé de la stratégie de correction : soit une stratégie globale (correction de toutes les structures possibles d'une phrase donnée par exemple), soit une stratégie locale (correction indépendante de chaque arbre produit par l'analyseur). Le noyau décide également des heuristiques de correction pour un arbre donné : correction en fonction d'une catégorie dominante, en fonction du nombre d'erreurs, en fonction de critères phonétiques (voir §6.3.2.).

Les arbres à corriger ainsi que les heuristiques choisies sont transmis au correcteur syntaxique qui utilise le module phonétique et le générateur morphologique pour produire les versions corrigées des arbres.

En fonction de la stratégie choisie (globale ou locale) le noyau soumet des propositions à l'utilisateur qui décide quelle est la bonne structure.

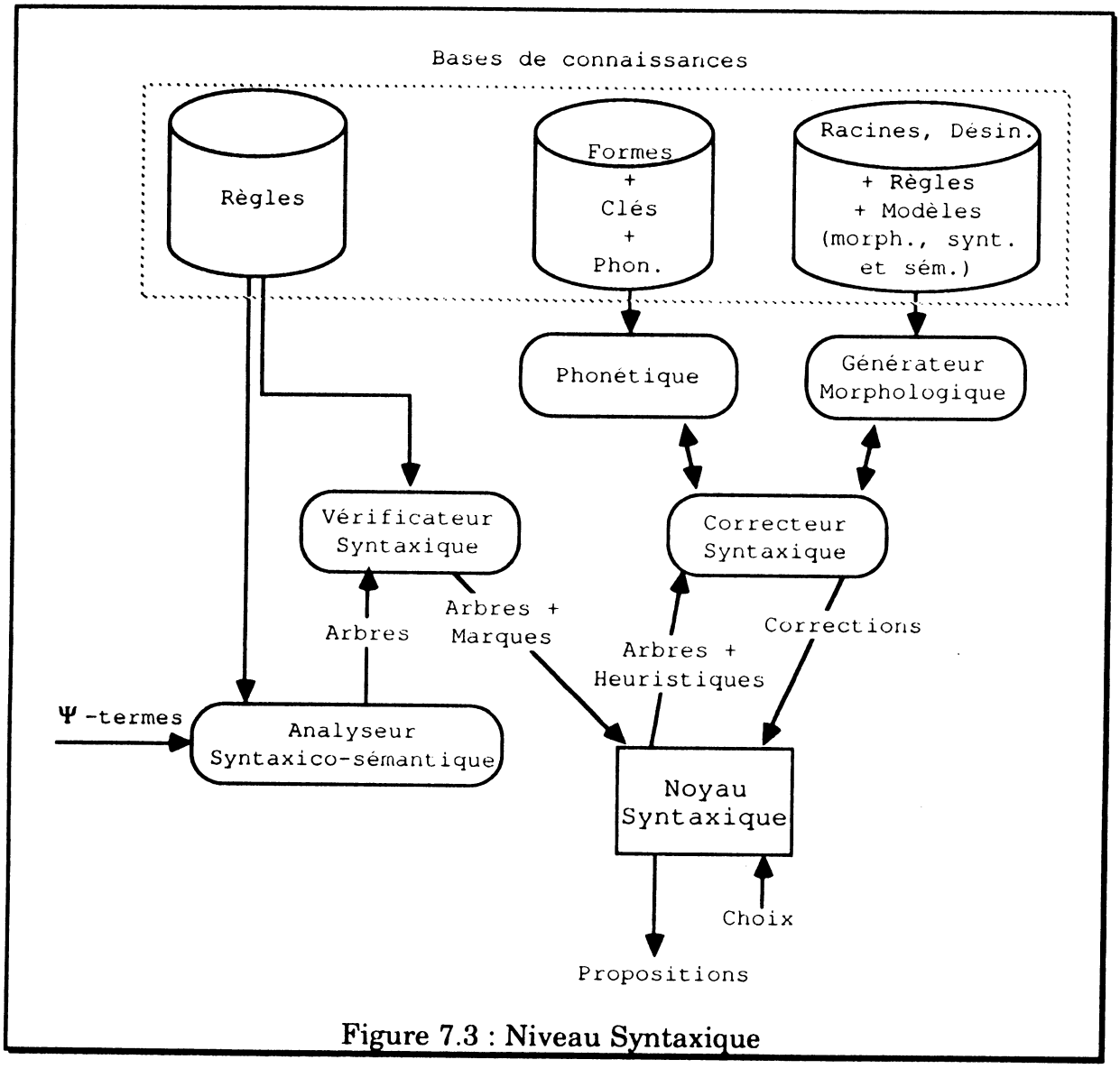


Figure 7.3 : Niveau Syntaxique

7.1.3. Parallélisme et architecture répartie

L'indépendance des différents modules permet en outre leur exécution simultanée. Ainsi, si le support matériel et logiciel le permet, le noyau pourra lancer tous les modules en parallèle en ayant pour charge d'assurer la synchronisation et la communication. Ainsi au niveau lexical, les trois méthodes de correction peuvent fonctionner en parallèle. L'analyseur morphologique doit par contre attendre les hypothèses de correction avant de poursuivre l'analyse. Les deux niveaux lexical et syntaxique peuvent fonctionner en mode "pipe-line" : dès que l'analyseur morphologique obtient un résultat, il le transmet à l'analyseur syntaxico-sémantique.

A l'heure des réseaux et des architectures réparties, on peut même imaginer de répartir les ressources sur différentes machines. On peut par exemple bâtir une architecture dans laquelle la boîte à outils et la base de connaissances sont mises en œuvre sur un serveur puissant, capable de temps de réponse très courts grâce à la vitesse de ses disques et à sa puissance de calcul. Ces

ressources sont partagées par un certain nombre de postes de travail (par exemple de simples micro-ordinateurs) connectés au serveur. Sur ces postes tournent les noyaux et les interfaces : chaque noyau est un client du serveur.

Outre le partage des ressources, cette architecture donne une plus grande souplesse d'utilisation : chaque noyau de chaque client peut ainsi mettre en œuvre une stratégie de correction différente et on peut même construire des noyaux utilisant les outils linguistiques du serveur pour d'autres applications que la détection/correction. Cependant, le partage impose une base de connaissances commune et rend donc plus difficile l'adaptation à un domaine particulier (vocabulaire terminologique, grammaire et sémantique particulières). Pour éviter ce problème, on peut concevoir une architecture dans laquelle le serveur contient les bases de connaissances générales et le client ses propres bases, spécifiques à l'application qu'il met en œuvre. Toutefois, on perd une partie du partage de ressources puisque les outils d'accès à une base doivent être présents à la fois sur le serveur et sur le client, de plus le noyau doit tenir compte de la disponibilité de deux bases différentes.

7.2. Utilisation et adaptation

7.2.1. Gestion de la base de connaissances

L'utilisation d'un tel système suppose la constitution d'une base de connaissances linguistiques très importante :

- règles de transcription graphique-phonétique et phonétique-graphique ;
- dictionnaire de racines et désinences associé aux règles et modèles pour l'analyse et la génération morphologique ;
- table de correspondance clé-forme ;
- signature et bases de Ψ-termes codant l'information syntaxico-sémantique (exploitée par l'analyseur morphologique) ;
- règles de construction et de vérification syntaxique.

La constitution d'une telle base pose bien sûr le problème de sa couverture : s'il paraît possible de constituer un dictionnaire de racines et désinences qui couvre 90% du français courant, il sera par contre très difficile à court terme de construire la base de Ψ-termes associée. De même il est très difficile d'écrire une grammaire exhaustive du français.

L'utilisation du système complet devra donc être limitée à des domaines particuliers, possédant un vocabulaire restreint, une grammaire limitée et une sémantique précise. Il est donc indispensable de fournir des outils pour adapter la base de connaissances à une application particulière.

Au niveau lexical, on peut modifier tous les paramètres liés à l'analyseur morphologique (partagés par le générateur morphologique) et mettre à jour le dictionnaire de racines et désinences. Sur la base de ce dictionnaire, on peut, grâce au générateur morphologique, engendrer de manière entièrement automatique le dictionnaire clé-forme. Les règles de transcription graphique-phonétique et phonétique-graphique peuvent également être modifiées grâce à des éditeurs interactifs.

Les informations au niveau lexical sont donc entièrement paramétrées mais la constitution d'une base de connaissances à partir de rien est un travail long et difficile qui ne peut être confié qu'à un linguiste ayant une bonne connaissance des mécanismes internes des divers modules de la boîte à outils. D'autre part, les différences d'une application à l'autre ne nécessitent par une reconstruction complète. Nous proposons donc de fournir les connaissances de base, celles qui ne changent pas d'une application à l'autre, ainsi que des outils de mise à jour qui permettent une adaptation aisée de ces connaissances.

Cette base de connaissances initiale contiendra :

- les règles de transcription graphique-phonétique qui peuvent être établies de manière exhaustive une fois pour toutes.
- les correspondances phonétique-graphique, également établies a priori et figées. Il s'agit ici de la donnée, pour un p-phonème, des p-graphèmes dans lesquels il peut se réécrire (voir §5.2.2.). Concernant les règles de transcription phonétique-graphique proprement dites, elles pourront être calculées automatiquement par analyse du dictionnaire de formes.
- un dictionnaire clé-forme, qui sera construit automatiquement sur la base du dictionnaire de racines et désinences et des règles morphologiques.
- un dictionnaire de racines et désinences sur lequel tout repose et qui contiendra au minimum toutes les classes fermées et surtout toutes les exceptions dont l'indexation requiert une connaissance approfondie des mécanismes internes de l'analyseur morphologique et de la structure du dictionnaire. Nous pensons notamment à tous les verbes irréguliers qui requièrent l'indexation de plusieurs bases chaînées entre elles. L'idée est qu'avec ce dictionnaire, l'ajout du vocabulaire propre au domaine puisse être réalisé par équivalence avec le vocabulaire existant. En effet, le gestionnaire du dictionnaire de racines et désinences fournit la possibilité d'insérer un mot par équivalence à un autre.
  Exemple :
  On peut insérer "compilera" comme "pilera" ou "mésopotamien" comme "chien".

Au niveau syntaxique, seules les règles de vérification peuvent être fixées a priori. Les règles de construction ainsi que les informations syntaxico-sémantiques attachées à chaque mot sont relativement dépendantes du domaine d'application. On donnera, pour chaque entrée du dictionnaire fondamental décrit ci-dessus, les informations syntaxico-sémantiques correspondant à l'usage courant. On fournira également une grammaire de dépendances en accord avec ces informations : on a vu au chapitre 2 qu'on pouvait écrire, pour une base lexicale donnée, des grammaires très générales qui peuvent ensuite être affinées pour une application donnée.

Quoiqu'il en soit, on devra fournir des outils permettant la modification des règles de construction et surtout la mise à jour de la base de Ψ-termes contenant les informations syntaxico-sémantiques associées aux entrées du dictionnaire de racines et désinences. On a vu au §5.3 qu'on pouvait se contenter de définir dans cette base le vocabulaire du domaine.

Si on peut fournir une telle base de connaissances fondamentales, l'adaptation du système à une application donnée sera grandement facilitée : on procédera à une étude de corpus afin de déterminer :

1- les mots à ajouter au niveau lexical et la spécificité syntaxico-sémantique de chaque mot ;
2- les règles de construction qu'il faudra ajouter (ou enlever) pour produire des structures adéquates.

Comme l'analyseur syntaxique produit toujours un résultat, on pourra l'utiliser pour l'analyse du corpus dans la deuxième étape : sur la base des résultats produits par la grammaire générale fournie, appliquée aux phrases du corpus, on ajoutera des règles plus spécifiques ou on éliminera certaines règles.

7.2.2. Applications

Les applications du système complet décrit ci-dessus devront être restreintes à des domaines où la syntaxe et surtout la sémantique sont réduites. On peut imaginer des applications dans les domaines suivants :

- enseignement du français assisté par ordinateur : pour les classes élémentaires on pourra avoir une grammaire relativement complète pour la description des constructions de base et l'utilisation d'un dictionnaire important permettra de donner à l'élève une grande liberté sur le choix des mots. Nous pensons que la compétence et les outils informatiques existent, il ne reste plus qu'à établir une collaboration avec des enseignants pour la réalisation d'une interface vraiment didactique.
- interfaces homme-machine en langue naturelle : pour ce type d'application les corrections peuvent être réalisées de manière entièrement automatique si le domaine est assez limité. En effet, si le dictionnaire est très réduit et si l'utilisateur n'est supposé employer que ce vocabulaire, on aura peu de corrections possibles pour un mot, voire une seule.
- outils d'aide à la rédaction de textes techniques pour lesquels la syntaxe est relativement pauvre et la sémantique précise. On pourrait utiliser le système complet pour la détection et la correction aux deux niveaux.

Si on n'utilise que le niveau lexical, pour lequel il est plus facile d'obtenir une couverture importante du français, on peut envisager l'intégration du système dans des traitements de textes généraux ou comme post-traitement d'un système de reconnaissance optique de caractères.

Conclusion de la deuxième partie

Nous avons proposé un système dont la mise en oeuvre effective est envisageable à très court terme. En effet, tous les modules de niveau lexical sont d'ores et déjà opérationnels sur micro-ordinateur, les modules de niveau syntaxico-sémantique le sont aussi, mais à l'état de prototypes.

Comme nous l'avons déjà dit dans la conclusion de la première partie, un travail assez important reste à faire pour rendre le constructeur de structures de dépendances vraiment opérationnel. Concernant le vérificateur syntaxique, il suffit de l'adapter au nouveau formalisme de représentation de la connaissance et de modifier la méthode de parcours des arbres (intégrer le marquage), mais ceci ne représente pas un énorme travail. Le correcteur est quand à lui quasi-opérationnel, moyennant une adaptation à la manipulation d'arbres décorés par des Ψ-termes.

Le travail le plus important concerne les noyaux : nous pensons qu'il n'y a pas de système de correction viable sans une stratégie de correction cohérente et efficace. Un système de correction doit être *ergonomique*.

Au niveau lexical, il va s'agir de limiter le nombre de corrections proposées et dans la mesure du possible d'améliorer ces corrections. On peut dans un premier temps envisager, pour limiter le nombre de solutions, d'utiliser des techniques de filtrage portant sur la catégorie du mot et sur son contexte. Nous pensons notamment aux méthodes statistiques du type de celles utilisées par [DEBILI 77] ou [KALLAS 87]. Un module utilisant ces techniques pourrait être intégré dans la boîte à outils et être utilisé :

- dans un système ne corrigeant que le niveau lexical : comme filtre pour limiter le nombre de solutions proposées à l'utilisateur ;
- dans un système complet, intercalé entre le niveau lexical et le niveau syntaxique et chargé de limiter le nombre d'hypothèses transmises au niveau syntaxique.

Dans un système ne corrigeant que le niveau lexical, on peut également utiliser l'analyseur syntaxique : on peut imaginer un fonctionnement en parallèle des analyseurs morphologique et syntaxique et en présence d'un mot inconnu, lancer la correction avec les techniques de niveau lexical, transmettre les hypothèses au niveau syntaxique, mais ne lancer d'interaction avec l'utilisateur qu'après les deux, trois ou quatre mots qui suivent le mot inconnu. L'idée est de laisser à l'analyseur le temps de calculer l'arbre syntaxique et donc d'éliminer certaines des hypothèses. On aurait alors un retour du niveau syntaxique vers le niveau lexical.

Au niveau syntaxique, il est indispensable, pour être ergonomique, de déterminer et de mettre en œuvre des stratégies capables, sinon de déterminer

où est l'erreur, en tous cas de poser à l'utilisateur la bonne question : celle qui permet de le déterminer. Il est également important de pouvoir effectivement poser une question et donc de doter le niveau syntaxique de moyens de paraphrase, ou plus généralement de génération de niveau syntaxique.

Quoiqu'il en soit, la réalisation d'un système complet avec une couverture importante suppose la disponibilité d'un dictionnnaire morphologique, syntaxique et sémantique qui ait cette couverture et d'un ensemble de règles de constructions qui couvre la syntaxe sous-jacente. Ceci ne peut être envisagé qu'à moyen ou long terme.

Pour le court terme, il nous paraît important de signaler que la modularité et l'indépendance des différents modules de la boîte à outils permet de les utiliser dans de nombreuses applications qui n'ont pas de rapport avec la détection/correction, mais qui manipulent la langue naturelle. Dans un traitement de texte, par exemple, on pourrait utiliser les analyseurs, le générateur morphologique et le correcteur syntaxique (avec une stratégie forcée) pour mettre en œuvre des fonctionnalités du type "mettre cette phrase au pluriel".
Signalons enfin deux projets en cours qui illustrent cette adaptabilité :

- le premier, en collaboration avec le GETA[1] [TOMASINO 90], vise à la construction d'un lemmatiseur intégré à un système général d'assistance linguistique sur Macintosh. Ce lemmatiseur doit permettre la mise en relation de textes en langue naturelle avec des dictionnaires usuels. Il est basé sur les outils morphologiques de PILAF (analyseur et générateur).
- le second, en collaboration avec l'ICP[2], utilise également les outils morphologiques pour réaliser une analyse de corpus. Il s'agit de déterminer, dans un corpus de textes qui sont des transcriptions de textes oraux, la catégorie de chaque mot, sa fréquence d'utilisation, la nature des phrases (nominales, verbales) ainsi que les types d'ambiguïtés lexicales les plus fréquentes. Le but final est la construction d'une interface homme-machine multi-média (utilisant notamment le langage parlé).

[1]Groupe d'Etude pour la Tradution Automatique, Imag-Campus, 150, Rue de la Chimie, BP53X, 38041 Grenoble CEDEX

[2]Institut de la Communication Parlée, 48 Ave Felix Viallet, 38031 Grenoble CEDEX

Références bibliographiques

Abréviations utilisées :

ACL	Association for Computational Linguistics
AI	Artificial Intelligence
AJCL	American Journal of Computational Linguistics
CACM	Communications of the Association for Computing Machinery
CoLing	International Conference on Computational Linguistics
IJCAI	International Joint Conference on Artificial Intelligence
JACM	Journal of the Association for Computing Machinery
JLP	Journal of Logic Programming
RFIA	Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle

[AHO & ULL 72] : Alfred D. Aho, Jeffrey D. Ullman
The theory of parsing, translation, and compiling. Volume I : Parsing.
Prentice Hall, 1972

[AHO & HOP & ULL 74] : Alfred D. Aho, J.E. Hopcroft, Jeffrey D. Ullman
The Design and Analysis of Computer Algorithms.
Addison Wesley, 1974

[AHO & ULL 77] : Alfred D. Aho, Jeffrey D. Ullman
Principles of compiler design.
Addison Wesley, 1977

[AIT KACI 84] : Hassan Aït Kaci
A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.
Ph.D. Thesis - Computer and Information Science, University of Pennsylvania, Philadelphia, USA, 1984

[AIT KACI 86a] : Hassan Aït Kaci
An Algebraïc Semantics Approach to the Effective Resolution of Type Equations.
Theoretical Computer Science 45, 1986, pp 293-351

[AIT KACI 86b] : Hassan Aït Kaci, Roger Nasr
LOGIN : a logic programming language with built-in inheritance.
JLP 3, 1986, pp 185-215

[AIT KACI 88] : Hassan Aït Kaci, Patrick Lincoln
LIFE : A natural language for natural language.
MCC Technical Report - Number ACA-ST-074-88, February 1988

[AIT KACI 89a] : Hassan Aït Kaci, Roger Nasr
Integrating logic and functional programming.
Lisp and Symbolic Computation 2, 1989, pp 51-89

[AIT KACI 89b] : Hassan Aït Kaci et al.
Efficient implementation of Lattice Operations.
ACM Transactions on Programming Languages and Systems 11:1, 1989, pp 116-146

[ALLEN 87] : James Allen
Natural Langage Understanding.
Benjamin/Cummings Publishing Company, 1987

[BAINIER 89] : Françoise Bainier
Détection et correction d'erreurs orthographiques.
Rapport de stage IUT II, Grenoble, Juin 89.

[BESCHERELLE 84] :
LE NOUVEAU BESCHERELLE - 3. La grammaire pour tous.
Librairie Hatier, Paris, Mai 1984

[BINOT 85] : Jean-Louis Binot
SABA : Vers un système portable d'analyse du français écrit.
Thèse de doctorat, Université de Liège - Faculté des sciences appliquées - Collection des publications n°101, Décembre 85

[BLANK 87a] : Ingeborg Blank
Etude des constructions syntaxiques du verbe en vue d'un traitement automatique.
Les Cahiers du CRISS, 1987

[BLANK 87b] : Ingeborg Blank, Rosalba Palermiti
Rapport sur le comportement syntaxique des verbes de la langue française.
Les Cahiers du CRISS, 1987

[BOBROW 77a] : D.G. Bobrow & T. Winograd
An Overview of KRL, a Knowledge Representation Language.
Cognition 1:1, 1977

[BOBROW 77b] : D.G. Bobrow & A.M. Collins Eds
Representation of understanding : Studies in Cognitive Science.
Academic Press, New York, 1975

[BOITET 76] : Christian Boitet
Un essai de réponse à quelques questions théoriques et pratiques liées à la traduction automatique, définition d'un système prototype.
Thèse d'état, Grenoble I, 1976

[BOITET 78] : Christian Boitet, Philippe Guillaume, Maurice Quezel-Ambrunaz
Manipulation d'arborescences et parallélisme : le système ROBRA.
7th CoLing, 1978

[BOITET 88] : Christian Boitet
Representation and computation of units of translation for Machine Interpretation of spoken texts.
GETA & ATR Technical Report TR-I-0035, August 88.
also in Computers and Artificial Intelligence (Bratislava), 8:6, 1989, pp 505-545

[BOLC 80] :
Série : Natural Communication with Computers. Natural Language Question Answering Systems
Edited by Leonard Bolc - Hanser, 1980

[BOUMA 88] : Gosse Bouma, Esther König, Hans Uszkoreit
A Flexible Graph-Unification Formalism and its Application to Natural Language Processing.
IBM Journal of Research and Development 32:2, 1988, pp 170-184

[BRESNAN 81a] : Joan Bresnan Ed.
The Mental Representation of Grammatical Relations.
MIT Press, Cambridge, Mass., 1981

[BRESNAN 81b] : Joan Bresnan, Ronald Kaplan
Lexical Functional Grammars ; a Formal System for Grammatical Relations.
In [BRESNAN 81a]

[BRACHMAN 79] : R.J. Brachman
On the epistemological status of semantic networks.
in [FINDLER 79].

[CALDER 88] : Jonathan Calder, Ewan Klein & Henk Zeevat
Unification Categorial Grammars : a concise, extendable grammar for natural language processing.
12th CoLing, Budapest, Hungary, August 1988

[CARBONNELL 81] : Jaime.G. Carbonell & Philip.J. Hayes
Dynamic Strategy Selection in Flexible Parsing.
19th Annual meeting of the ACL, 1981

[CARBONNELL 83] : Jaime.G. Carbonell & Philip.J. Hayes
Recovery Strategies for Parsing Extragrammatical Language.
AJCL 9:3-4, 1983, pp 123-146

[CARBONNELL 84] : Jaime.G. Carbonell & Philip.J. Hayes
Coping with Extragrammaticality.
10th CoLing, Stanford, USA, July 1984, pp 437-443

[CHAPPUY 83] : Sylviane Chappuy
Formalisation de la description des niveaux d'interprétation des langues naturelles. Etude menée en vue de l'analyse et de la génération au moyen de transducteurs.
Thèse de 3ème cycle, Grenoble 1, Juillet 1983

[CHARNIAK 76] : Eugene Charniak & Yorrick Wilks Eds
Computational Semantics : An Introduction to Artificial Intelligence and Natural Language Comprehension.
North Holland, Amsterdam, 1976

[CHARNIAK 78] : Eugene Charniak
On the use of framed knowledge for language comprehension.
AI 11, 1978

[CHAUCHE 74] : Jacques Chauché
Transducteurs et arborescences. Etudes et réalisations de systèmes appliqués aux grammaires transformationnelles.
Thèse d'état, Grenoble I, 1974.

[CHOMSKY 57] : Noam Chomsky
Syntactic Structures.
Mouton & Co, La Haye, 1957
Traduction :
Structures Syntaxiques.
Seuil, Paris, 1959

[CHOMSKY 65] : Noam Chomsky
Aspects of the theory of syntax.
MIT Press, Cambridge, Mass., 1965 (Traduction : *Aspects de la théorie syntaxique.* Seuil, Paris, 1971)

[CHOMSKY 72] : Noam Chomsky
Studies on Semantics in Generative Grammar
Mouton & Co, La Haye, 1972 (Traduction : *Questions de sémantique.* Edition du Seuil, Paris, 1975)

[CHOMSKY 75] : Noam Chomsky
Reflexions on language
Pantheon, New Jersey, 1975

[CHOMSKY 82] : Noam Chomsky
Some Concepts and Consequences of the Theory of Government and Binding.
MIT Press, 1982 (Traduction : *La nouvelle syntaxe.* Seuil, Paris, 87)

[COHARD 88] : Brigitte Cohard
Logiciel de détection et de correction des erreurs lexicales.
Mémoire CNAM, Grenoble, Mars 1988

[COLMERAUER 70] : Alain Colmerauer
Les systèmes-Q, un formalisme pour analyser et synthétiser des phrases sur ordinateur.
TAUM - Université de Montréal, 1970

[COLMERAUER 77] : Alain Colmerauer
Un sous-ensemble intéressant du français.
Actes des journées IRIA sur la compréhension, Arc et Senans, Mai 1977

[COULON 80] : Daniel Coulon , Daniel Kayser
Un système de raisonnement à profondeur variable.
3ème RFIA, Nancy 1981

[COURTIN 76] : Jacques Courtin, Danièle Dujardin
Paramètres linguistiques de la Morphologie Française dans le système PIAF.
Rapport interne du Laboratoire d'Informatique de Grenoble, Décembre 1976

[COURTIN 77] : Jacques Courtin
Algorithmes pour le traitement interactif des langues naturelles.
Thèse d'état, Grenoble I, Octobre 1977

[COURTIN 86] : Jacques Courtin, Danièle Dujardin, Irène Kowarski
Le système PILAF.
Table Ronde Franco-Allemande sur les langues naturelles, Stuttgart, RFA, Décembre 1986

[COURTIN 88] : Jacques Courtin, Danièle Dujardin, Irène Kowarski, Damien Genthial, Vera Lúcia Strube de Lima
Correção de erros de ortografia através da fonética em textos escritos em francês.
XIV Conferencia Latinoamericana de Informática, 17avas Jornadas Argentinas de Informática e Investigación Operativa, Buenos Aires, Sep. 1988, pp 873-891

[COURTIN 89a] : Jacques Courtin, Danièle Dujardin, Irène Kowarski, Damien Genthial, Vera Lúcia Strube de Lima
DECOR - detecção e correção léxico-sintática num texto escrito.
XV Conferencia Latinoamericana de Informática, IX Conferencia Internacional de la Sociedad Chilena de Ciência de la Computación, Santiago du Chili, Juillet 1989, pp 67-76

[COURTIN 89b] : Jacques Courtin, Danièle Dujardin, Irène Kowarski, Damien Genthial, Vera Lúcia Strube de Lima
Análise de textos escritos em português com PILAF - uma experiência e seus resultados.
18avas Jornadas Argentinas de Informática e Investigación Operativa, Buenos Aires, Août 1989, pp 9.29-9.46.

[COURTIN 89c] : Jacques Courtin, Danièle Dujardin, Irène Kowarski, Damien Genthial, Vera Lúcia Strube de Lima
Interactive Multi-Level Systems for Correction of Ill-Formed French Texts.
2nd Scandinavian Conference on Artificial Intelligence, Tampere, Finland, June 1989, pp 912-920

[COURTIN 90] : Jacques Courtin, Danièle Dujardin, Damien Genthial, Irène Kowarski
Creation And implementation on micro-computers of large scale French language dictionaries.
Conference on computational lexicography, Balatonszabadi , Hungary, September 1990

[DAMERAU 64] : F.J. Damerau
A technique for computer detection and correction of spelling errors.
CACM 7:3, March 1964, pp 171-176

[DANLOS 85] : Laurence Danlos
Génération automatique de textes en langue naturelle.
Masson - Collection Etudes et Recherches en Informatique, Paris, 1985

[DEBILI 77] : Fathi Debili
Traitements syntaxiques utilisant des matrices de précédence fréquentielles construites automatiquement par apprentissage.
Thèse de Docteur-Ingénieur, Paris VII, 1977

[DEBILI 82] : Fathi Debili
Analyse syntaxico-sémantique fondée sur une acquisition automatique de relations lexicales-sémantiques.
Thèse d'état, Paris XI, 1982

[DEBILI 86] : Fathi Debili
Le traitement des entrées non prévues dans les systèmes de compréhension du langage naturel : détection et correction des graphies fautives.
Rapport d'activités du PRC Communication Homme-Machine, Pôle Langage Naturel, Orsay, 1986

[DIK 78] : Simon Dik
Functional Grammar
Foris - Publications in Language Sciences, Dordrecht, Holland, 1978.

[DURAND 88] : Jean-Claude Durand
TTEDIT : Un éditeur transformationnel d'arbres.
Thèse de l'Universté Joseph Fourier, Grenoble I, Mars 88

[DURHAM 83] : I. Durham, D. Lamb, J. Saxe
Spelling correction in user interfaces.
CACM 26:10, 1983, pp 764-773

[EARLEY 70] : Jay Earley
An efficient context-free parsing algorithm.
CACM 6:8, 1970, pp 451-455

[EISELE 88] : Andreas Eisele, Jochen Dörre
Unification of Disjunctive Feature Descriptions.
26th meeting of the ACL, Buffalo, NY, USA, June 1988

[EMELE 90] : Martin Emele, Ulrich Heid, Stefan Momma, Rémi Zajac
Organizing linguistic knowledge for multilingual generation.
13th CoLing, Helsinki, Finland, August 1990, Vol. 3, pp 102-107.

[EMIRKANIAN 88a] : L. Emirkanian, L. Bouchard
Towards a knowledge-based tool for correcting French text.
IFIP European Conference on Computer in Education, Lausanne, Switzerland, July 1988, pp 583-588.

[EMIRKANIAN 88b] : L. Emirkanian, L. Bouchard
Knowledge integration in a robust and efficient morpho-syntactic analyser for French.
12th CoLing, Budapest, Hungary, August 1988, pp 166-171.

[FILLMORE 68] : C.J. Fillmore
The Case for Case.
In Bach & Harms Eds : Universal in Linguistic Theory, Holt, Rinehart & Winston Inc., 1968

[FILLMORE 71] : C.J. Fillmore
Some problems for case grammars.
In R.J. O'Brien (Ed.), Report of the twenty-second annual round table meeting on linguistics and language studies. Monograh Series on Languages and Linguistics, N° 24, Washington DC, Georgestown University Press, 1971.

[FINDLER 79] : N.V. Findler Eds.
Associative Networks : Representation and Use of Knowledge by Computers.
Academic Press, New York, 1979

[FOUQUERE 88] : Christophe Fouqueré
Systèmes d'analyse tolérante du langage naturel.
Thèse de Doctorat, Paris-Nord, Janvier 1988.

[FOURNIER 87] : Jean Pierre Fournier, P. Herman, Gérard Sabah, Anne Vilnat, N. Burgaud, M. Gilloux
Traitement des mots inconnus dans un système de questions-réponses en langue naturelle.
6ème RFIA, Antibes, Décembre 87, pp 653-667

[FOURNIER 88] : Jean Pierre Fournier, Jean Véronis
Traitement des erreurs dans la communication homme-machine en langage naturel.
Premières journées nationales du GRECO-PRC Communication Homme-Machine, Paris, Novembre 1988

[FRISCH 88] : Rudolf Frisch, Antonio Zamora
Spelling assistance for compound words.
IBM Journal of Research and Development 32:2, March 1988

[GAIFMAN 65] : H. Gaifman
Dependency Systems and Phrase Structure Systems.
Information and Control 8:3, 1965, pp 304-337

[GAL 84] : A. Gal, P. Levasseur, J.M. Walle
Compréhension du langage naturel et utilisation d'une base de connaissances.
Journées Applications Informatiques Conversationnelles et le Langage Naturel, Actes du colloque Traitement Automatique du Langage Naturel, Nantes, 1984

[GAL 86] : A. Gal, J.H. Jayez, P. Levasseur, M. Liscouet, J.M. Walle
Compréhension du langage naturel. Le cas de l'interrogation simple en français.
Masson - Collection Méthodes + Programmes, Paris, 1986

[GAL 89] : A. Gal, G. Lapalme, P. St Dizier
Prolog pour l'analyse automatique du langage naturel.
Eyrolles, Paris, 1989.

[GAZDAR 85] : Gerald Gazdar & al
Generalized Phrase Structure Grammar.
Stephen Austin & Sons Ltd, Hertford, 1985

[GAZDAR 86] : Gerald Gazdar
Generative Grammar.
Cognitive Science Research Report, University of Sussex, 1986

[GENTHIAL 86] : Damien Genthial
Interrogation de bases de données en langue naturelle.
Rapport de D.E.A., Grenoble I, Juin 1986

[GENTHIAL 90] : Damien Genthial, Jacques Courtin, Irène Kowarski
Contribution of a Category Hierarchy to the Robustness of Syntactic Parsing.
13th CoLing, Helsinki, Finland, August 1990, Vol. 2, pp 139-144

[GENTHIAL 91a] : Damien Genthial
Souplesse et robustesse de l'analyse syntaxique : apports d'une hiérarchie de catégories.
Colloque "Informatique & Langue Naturelle", Nantes, Janvier 91

[GENTHIAL 91b] : Damien Genthial, Jacques Courtin
Représentation des données lexicales : vers des traitements tolérants.
2èmes Journées Nationales du GRECO-PRC Communication Homme-Machine, Toulouse, Janvier 91

[GOESER 90] : Sebastian Goeser
A Linguistic Theory of Robustness.
13th CoLing, Helsinki, Finland, August 90, Vol. 2, pp 156-161

[GRANGER 77] : R.H. Granger
FOUL-UP : A Program that Figures out Meanings of Words from Context.
5th IJCAI, 1977

[GRANGER 83] : R.H. Granger
The NOMAD System : Expectation-Based Detection and Correction of Errors during Understanding of Syntactically and Semantically Ill-Formed Text.
AJCL 9:3-4, 1983, pp 188-196

[GREVISSE 69] : Maurice Grévisse
Le bon usage.
Hatier, Paris, 1969

[GRISHMAN 73] : Ralph Grishman
Implementation of a string parser of English.
in [RUSTIN 73]

[GRISHMAN 86] : Ralph Grishman
Computational linguistics, an introduction.
Cambridge Press University - Studies in NLP Series, Cambridge - Mass., USA, 1986

[GROSS 75a] : Maurice Gross
Grammaire transformationnelle du français : syntaxe du verbe.
Larousse, Paris, 1975

[GROSS 75b] : Maurice Gross
Méthodes en syntaxe.
Hermann, Paris, 1975

[GROSS 77] : Maurice Gross
Grammaire transformationnelle du français : syntaxe du nom.
Larousse, Paris, 1977

[GROSS 84] : Maurice Gross
Lexicon-Grammar and the Syntactic Analysis of French.
10th CoLing, Stanford, July 1984, pp 275-282

[HAJICOVA 88] : Eva Hajicova
Reasons Why We Use Dependency Grammars.
Panel of 12th CoLing, Budapest, Hungary, August 1988, p 451

[HALL 80] : P. Hall, G. Dowling
Approximate string matching.
Computing Surveys 12:4, 1980, pp 381-402

[HALLIDAY 61] : M.A.K. Halliday
Categories of the Theorie of Grammar.
Word 17, 1961

[HARRIS 68] : Z.S. Harris
Mathematical Structures of Language.
J. Wiley & Sons, New York, 1968,
Traduit par C. Fuchs :
Structures mathématiques du langage.
Dunod, Paris, 1971

[HAYES 80] : P.J. Hayes & G. Mouradian
Flexible Parsing.
18th Annual meeting of the ACL, 1980

[HAYES 84] : Philip J. Hayes
Entity-Oriented Parsing
10th CoLing, Stanford, USA, July 1984, pp 212-217

[HAYES 86] : Philip J. Hayes, Alexander J. Hauptman, Jaime J. Carbonnell, Masaru Tomita
Parsing Spoken Language : a Semantic Caseframe Approach.
11th CoLing, Bonn, FRG, August 1986, pp 587-592

[HAYS 64] : D. Hays
Dependency theory : a formalism and some observations.
Language 40, 1964, pp 511-525

[HEIDORN 82] : G. E. Heidorn et al.
The EPISTLE text-critiquing system.
IBM Systems Journal 21:3, 1982.

[HELLWIG 80] : Peter Hellwig
PLAIN : A Program System for Dependency Analysis and for Simulating Natural Language Inference.
in [BOLC 80], pp 271-376

[HELLWIG 86] : Peter Hellwig
Dependency Unification Grammar
11th CoLing, Bonn, FRG, August 1986, 195-198.

[HENDRIX 79] : G.G. Hendrix
Encoding Knowledge in partitioned networks.
in [FINDLER 79].

[HENDRIX 82] : G.G. Hendrix
Natural Language Interface.
Summary of a Panel of the Whorkshop on Applied Computational Linguistics in Perspective. JACL 8: 2, 1982.

[HOFFMAN 82] : C.M. Hoffman, M.J. O'Donnell
Pattern Matching in trees.
JACM 29:1, 1982, pp 68-95

[HUDSON 71] : R.A. Hudson
English Complex Sentences : An Introduction to Systemic Grammar.
North Holland, Amsterdam, 1971

[JACKENDOFF 75] : Ray Jackendoff
A System of Semantic Primitives.
In Theorical Issues in Natural Language Processing, Eds Nash-Vebber & Schank, MIT Press, 1975

[JÄPPINEN 88] : H. Jäppinen, E. Lassila, A. Lehtola
Locally Governed Trees and Dependency Parsing.
12th CoLing, Budapest, Hungary, August 1988, pp 275-277

[JAYEZ 79] : Jacques-Henri Jayez
Une approche de la compréhension par machine du langage naturel.
Thèse d'état, Paris VII, 1979

[JAYEZ 82] : Jacques-Henri Jayez
Compréhension automatique du langage naturel, le cas du groupe nominal en français.
Masson - Collection Méthode+Programmes, Paris, 1982.

[JAYEZ 88] : Jacques Jayez
L'inférence en langue naturelle.
Hermès, Paris, 1988.

[JENSEN 83] : K. Jensen, G.E. Heidorn, L.A. Miller, Y. Ravin
Parse Fitting and Prose Fixing : Getting a Hold on Ill-formedness.
AJCL 9:3-4, 1983, pp 147-160

[KALLAS 87] : G. Kallas
Résolution des solutions multiples en analyse morphologique automatique des langues naturelles. Utilisation des modèles de Markov.
Thèse de 3ème cycle, Grenoble II, 1987

[KAMP 81] : H. Kamp
A theory of truth and sematic representation.
In Formal methods in the study of language 136, Groenendijk J.A.G., Jansen T.M.V. and Stockhof M.J.B. Ed., 1981, pp 277-322

[KAPLAN 72] : R.M. Kaplan
Augmented Transition Networks as Psychological Models of Sentence Comprehension.
AI 3, 1972

[KAPLAN 73] : R.M. Kaplan
A General Syntactic Processor.
In [RUSTIN 73]

[KARTTUNEN 86] : Lauri Karttunen
D-PATR : a Development Environment for Unification-Based Grammars.
11th CoLing, Bonn, FRG, August 1986

[KASPER 87] : Robert T. Kasper
A Unification Method for Disjunctive Feature Description.
25th Annual Meeting of the ACL, 1987, pp 235-242

[KASPER 88] : Robert T. Kasper
Conditional Descriptions in Functional Unification Grammar.
26th Annual Meeting of the ACL, Buffalo, USA, June 88

[KATZ 63] : J. Katz & J. Fodor
The Structure of Semantic Theory.
Language 39, 1963

[KAY 84] : Martin Kay
Functional Unification Grammar : a Formalism for Machine Translation.
10th CoLing, Stanford, USA, July 1984, pp 75-78

[KAY 81] : Martin Kay
Unification Grammars
Xerox Publication, 1981

[KAYSER 82] : D. Kayser, D. Coulon
La compréhension : un processus à profondeur variable.
Bulletin de psychologie 35, 1982

[KNUTH 65] : D. E. Knuth
On the translation of language from left to right.
Information and Control, 8, 1965

[KNUTH 73] : D. E. Knuth
The Art of Computer Programming.
Addison Wesley, Vol. 1, 3, 1973.

[KULAGINA 67] : O.S. Kulagina, Igor Mel'Cuk
Automatic Translation : Some theoretical aspects and the design of a translation system.
In Machine Translation, A.D. Both Ed., North Holland, 1967

[KUNZE 75] : Jürgen Kunze
Abhängigkeitsgrammatik. (Grammaires de dépendances)
Akademie Verlag - Studia Grammatica XII, Berlin, 1975

[LACOUTURE 88] : Roxane Lacouture, Guy Lapalme
Une implantation informatique du français fondamental.
Techniques et Sciences Informatiques 7:5, 1988

[LAHENS 87] : François Lahens
Un modèle stochastique pour la vérification et la correstion automatique de textes : le système VORTEX.
Thèse de 3ème cycle, Université Paul Sabatier Toulouse, Janvier 1987

[LALLICH-BOIDIN 86] : Geneviève Lallich-Boidin
Analyse syntaxique automatique du français. Applications à l'indexation automatique.
Thèse de 3ème cycle, Grenoble II, 1986.

[LAPALME 86] : Guy Lapalme, Danièle Richard
Un système de correction automatique des accords des participes passés.
Techniques et Sciences Informatiques 4, 1986

[LAPALME 89] : Guy Lapalme, Diane Goupil
Un analyseur déterministe de la langue naturelle.
Techniques et Sciences Informatiques 8:4, 1989, pp 361-376

[LESMO 84] : Leonardo Lesmo, Pietro Torasso
Interpreting Syntactically Ill-formed Sentences.
10th CoLing, Stanford, USA, July 1984, pp 534-539

[LEHNERT 82] : Wendy G. Lehnert, Martin H. Ringle (Ed)
Strategies for Natural Language Processing.
Lawrence Erlbaum Associates Publishers, 1982

[LEVENSHTEIN 66] : V.I. Levenshtein
Binary codes capable of correcting deletions, insertions or reversals.
Soviet Physics Doklady 10:8, 1966, pp 707-710

[LOPEZ 90] : Eric Lopez
Compactage d'un gros dictionnaire.
Rapport de stage IUT II, Grenoble, Juin 90.

[LOWRANCE 75] : R. Lowrance, R., R. Wagner
An extension of the string-to-string correction problem.
JACM 22:2, 1975, pp 177-183

[LU 86] : Lu Chengren
Détection et correction des erreurs dans un texte écrit en langue naturelle.
Rapport de DEA, Grenoble I, Septembre 1986

[MARCUS 80] : M.P. Marcus
A Theory of Syntactic Recognition for Natural Language.
MIT Press, Cambridge, Mass., 1980

[MARET 87] : Dominique Maret
Comparaisons de chaînes de caractères, accès lexicaux tolérants et applications.
Thèse de 3ème cycle, Grenoble II, Mai 1987

[Mc CORD 82] : Michael Mc Cord
Using Slots and Modifiers in Logic Grammars for Natural Language.
AI 18, 1982

[MEL'CUK 71] : Igor Mel'cuk et A.K. Zolkovskij
Vers un modèle "sens-texte" du langage.
Documents de Linguistique quantitative 10, 1971

[MERLE 82] : A. Merle
Un analyseur présyntaxique pour la levée des ambiguïtés dans des documents écrits en langue naturelle : application à l'indexation automatique.
Thèse de doctorat, INPG, 1982

[NACHAWATI 81] : H. Nachawati
Une méthode probabiliste pour l'élimination d'ambiguïtés en sortie d'un analyseur morphologique de langue naturelle.
Thèse de 3ème cycle, Université Claude Bernard, Lyon I, 1981

[OKUDA 76] : Teruo Okuda, Eiichi Tanaka, Tamotsu Kasai
A method for the correction of garbled words based on the Levenshtein Metric.
IEEE Transactions on Computers C-25 :2, 1976, pp 172-177

[OWOLABI 88] : O. Owolabi, D. R. McGregor
Fast approximate string matching.
SOFTWARE, Practice and experience 18:4, 1988, pp 387-393.

[PALMER 90] : Patrick Palmer
Etude d'un analyseur de surface de la langue naturelle. Application à l'indexation automatique de textes.
Thèse de l'Université Joseph Fourier, Grenoble I, Septembre 90

[PEQUEGNAT 87] : Catherine Péquegnat
Compréhension automatique de requêtes en langue naturelle. Aspects linguistiques dans le contexte d'un système de recherche d'informations.
Rapport de DEA, Grenoble I, Septembre 1987

[PERENNOU 86] : Guy Pérennou, Pierre Daubeze, François Lahens
La vérification et la correction automatique de textes : le système VORTEX.
Techniques et Sciences Informatiques 5:4, 1986, pp 285-304

[PEREIRA 80] : Fernando Pereira, David Warren
Definite Clause Grammars for Language Analysis. A Survey of the Formalism and a Comparison with Augmented Transition Networks.
AI 13:3, 1980, pp 231-278

[PEREIRA 85] : Fernando Pereira
A Structure Sharing Representation for Unification-Based Formalisms.
23rd Annual Meeting of the ACL, 1985, pp 137-144

[PETERSON 80] : J.L. Peterson
Computers programs for detecting and correcting spelling errors.
CACM 23, 1980, pp 676-687

[PIERREL 87] : Jean-Marie Pierrel
Dialogue oral homme-machine.
Hermès, Paris, 1987.

[PITRAT 83] : Jacques Pitrat
Réalisation d'un analyseur-générateur lexicographique général.
Rapport n° 79-2 du GR22, Paris VI, 1983

[POLLOCK 84] : Joseph J. Pollock & Antonio Zamora
Automatic spelling correction in scientific and scholarly text
CACM 27:4, 1984

[PRATT 75] : V.R. Pratt
Lingol : A Progress Report.
4th IJCAI, 1975

[RADY 83] : Mohamed Rady
L'ambiguïté du langage naturel est-elle la source du non déterminisme des procédures de traitement ?
Thèse d'état, Paris VI, Juin 83

[RICHARDSON 85] : S. D. Richardson
Enhanced Text Critiquing using a Natural Language Parser : the CRITIQUE System.
IBM Research Report RC 11332, Yorktown Heights, USA, 1985

[RISEMAN 71] : E. Riseman, R. Ehrich
Contextual word recognition using binary digrams.
IEEE Transactions on Computers C-20:4, 1971, pp 397-403

[RISEMAN 74] : E. Riseman, A. Hanson
A contextual postprocessing system for error correction using binary n-grams.
IEEE Transactions on Computers, 23:5, 1974, pp 480-493

[ROBERT 78] : Paul Robert
Dictionnaire alphabétique et analogique de la langue française.
Ed. Société du Nouveau Littré, Paris, 1978

[ROBINSON 70] : J. Robinson
Dependency structures and transformational rules.
Language 46:2, 1970

[RUSTIN 73] : R. Rustin Eds
Natural Language Processing.
Algorithmics Press, New York, 1973

[SABAH 88] : Gérard Sabah
L'intelligence artificielle et le langage. Vol 1 : Représentation des connaissances.
Hermès, Paris, 1988

[SABAH 89] : Gérard Sabah
L'intelligence artificielle et le langage. Vol 2 : processus de compréhension.
Hermès, Paris, 1989

[SAGER 73] : Naomi Sager
The string parser for scientific litterature.
in [RUSTIN 73]

[SALKOFF 73] : Maurice Salkoff
Une grammaire en chaîne du français.
Dunod, Paris, 1973

[SCHABES 88] : Yves Schabes, Anne Abeille, A.K. Joshi
Parsing strategies with "lexicalized" grammars : application to Tree Adjoining Grammars.
12th CoLing, Budapest, Hungary, August 1988

[SCHANK 75] : R.C. Schank
Conceptual Information Processing.
North Holland, New York, 1975

[SCHANK 77] : R.C. Schank & R.P. Abelson
Scripts, Plans, Goals and Understanding.
Lawrence Erlbaum Ass., Hillsdale, New Jersey, 1977

[SELLS 85] : Sells
Lectures on Go&Bi, GPSG and LFG.
CSLI Lecture Notes 3, Stanford, 85

[SHIEBER 84] : Stuart M. Shieber
The Design of a Computer Language for Linguistic Information
10th CoLing, Stanford, USA July 1984, pp 362-366

[SHIEBER 86] : Stuart M. Shieber
An Introduction to Unification-Based Approach to Grammar.
CSLI Lecture Notes 4, 1986

[SLOCUM 81] : J. Slocum
A Practical Comparison of Parsing Strategies.
19th Annual meeting of the ACL, 1981

[SMOLKA 89] : Gert Smolka and Hassan Aït-Kaci
Inheritance Hierarchies : Semantics and Unification.
Journal of Symbolic Computation 7, 1989, pp 343-370

[SOWA 84] : J.F. Sowa
Conceptual Structures. Information processing in mind and machine.
Addison Wesley - The System Programming Series, 1984

[SPARCK-JONES 87] : Karen Sparck-Jones & Branimir Boguraev
A Note on a Study of Cases
AJCL 13:1-2,1987, pp 65-68

[STEYAERT 83] : J.M. Steyaert
Patterns and Pattern-Matching in Trees : an Analysis.
Information and Control 58, 1983, pp 19-58

[ST-DIZIER 84] : Patrick St-Dizier.
Etude et réalisation de ESOPE : un système paramétrable pour la traduction de sous-ensembles du français en logique.
Thèse de 3ème cycle, Université de Rennes I, 1984

[STRUBE DE LIMA 89] : Vera Lúcia Strube de Lima, Damien Genthial
Vers un système complet de détection/correction.
Rapport interne 01/1989 - Equipe TRILAN - LGI-Imag, Grenoble, Mars 1989.

[STRUBE DE LIMA 90] : Vera Lucia Strube de Lima.
Contribution à l'étude du traitement des erreurs au niveau lexico-syntaxique dans un texte écrit en français.
Thèse de l'Université Joseph Fourier, Grenoble I, Mars 1990

[TESNIERE 59] : Lucien Tesnière
Eléments de syntaxe structurale.
Klincksiek, Paris, 1959

[TOMASINO 90] : Isabelle Tomasino
ODILE : Un Outil d'Intégration Extensible de Dictionnaires et de Lemmatiseurs.
Thèse CNAM, Grenoble, Décembre 90.

[TOMITA 84a] : Masaru Tomita
LR Parsers for Natural Language.
10th CoLing, Stanford, USA, July 1984, pp 354-357

[TOMITA 84b] : Masaru Tomita
Disambiguating Grammatically Ambiguous Sentences By Asking.
10th CoLing, Stanford, USA, July 1984, pp 476-480

[TOMITA 87] : Masaru Tomita
An Efficient Augmented-Context-Free Parsing Algorithm.
AJCL 13:1-2, 1987, pp 31-46

[TOMITA 88a] : Masaru Tomita &al
The Generalized LR Parser Compiler Version 8.1 User's Guide.
CMT, CMU Memo, January 88

[TOMITA 88b] : Masaru Tomita
Graph-structured Stack and Natural Language Parsing.
26th Annual Meeting of the ACL, Buffalo, USA, June 88

[USZKOREIT 86] : Hans Uszkoreit
Categorial Unification Grammars.
11th CoLing, Bonn, FRG, August 1986

[VALKONEN 87] : K. Valkonen, H. Jäppinen, A. Lehtola
Blackboard based dependency parsing.
IJCAI, Milan, Italy, 1987.

[VAUCHER 80] : J.G. Vaucher
Pretty-Printing of Trees.
Software Practice and Experience, 10, 1980, pp 553-561

[VAUQUOIS 75] : Bernard Vauquois
La traduction automatique à Grenoble.
Documents de linguistique quantitative 24, Dunod, Paris, 1975

[VEILLON 70] : Veillon
Modèles et algorithmes pour la traduction automatique.
Thèse d'état, Grenoble I, 1970

[VERGNES 90] : Jacques Vergnes
A parser without a dictionary as a tool for research into French syntax.
13th CoLing, Helsinki, Finland, August 1990, Vol. 1, pp 70-72

[VERONIS 87] : Jean Véronis
Phonographic and typographical correction in Natural Language Interfaces.
Fifth International Symposium in Applied Informatics, Grindelwald-Suisse, February 1987

[VERONIS 88a] : Jean Véronis
Correction of phonographic errors in natural language interfaces.
11th International Conference on Research & Development in Information Retrieval, Grenoble, France, Juin 1988

[VERONIS 88b] : Jean Véronis
Morphosyntactic correction in natural language interfaces.
12th CoLing, Budapest, Hungary, August 1988, pp 708-713

[VERONIS 88c] : Jean Véronis
Contribution à l'étude de l'erreur dans le dialogue homme-machine en langage naturel.
Thèse de Doctorat, Aix-Marseille III, Octobre 1988

[WAGNER 74] : C. K.Wagner, M.J. Fischer
The string-to-string correction problem.
JACM 21:1, 1974, pp 168-173

[WEISCHEDEL 80] : Ralph.H. Weischedel & J.E. Block
Responding Intelligently to Unparsable Inputs.
AJCL 6:2, 1980

[WEISCHEDEL 83] : Ralph.H. Weischedel
Meta-Rules as a Basis for Processing Ill-formed Input.
AJCL 9:3-4, 1983, pp 161-177

[WINOGRAD 72] : Terry Winograd
Understanding Natural Language.
Academic Press, 1972

[WINOGRAD 83] : Terry Winograd
Language as a cognitive process. Vol 1 : Syntaxe
Addison Wesley, 1983

[WOODS 70] : William A. Woods
Transition Network Grammars for Natural Language Analysis.
CACM 13:10, 1970, pp 591-606

[ZAJAC 89] : Rémi Zajac, Martin Emele
Multiple Inheritance in RETIF.
Report of the ATR Interpreting Telephony Research Laboratories, TR-I-0114.

[ZAJAC 90] : Rémi Zajac, Martin Emele
Typed Unification Grammars.
13th CoLing, Helsinki, Finland, August 1990, Vol. 3, pp 293-298.

Annexes

Annexe A

Eléments de théorie des langages

Cette annexe donne quelques définitions fondamentales[1] de la théorie des langages formels. Nous nous sommes limité aux définitions et aux notions utiles à la compréhension du texte de la thèse. Nous présentons également de manière plus détaillée les deux méthodes d'analyse les plus utilisées pour les les langages de programmation : analyse prédictive LL et analyse ascendante LR.

A.1. Définitions générales

A.1.1. Langage

Un *vocabulaire* V est un ensemble fini et non vide de symboles. Un *mot* ou une *phrase* est une chaîne de longueur finie composée avec les symboles de V.

On note V^+ l'ensemble de toutes les chaînes engendrées par le vocabulaire V et V^* le même ensemble incluant la chaîne vide, notée ε.

Si X et Y sont deux chaînes, la chaîne Z = XY est le *produit* de X et de Y. On définit par analogie l'opération *puissance* : X^n est le produit de n occurrences de X.

On appelle *langage formel* sur un vocabulaire V, ou tout simplement *langage*, un sous ensemble de V^*.

A.1.2. Grammaire

Une *grammaire syntagmatique* G est définie par la donnée d'un quadruplet :

$$G = (V_T, V_N, S, P)$$

où V_T est un vocabulaire, dit *vocabulaire terminal* ; V_N est également un vocabulaire, dit *vocabulaire non-terminal* ou *vocabulaire auxiliaire* ; $S \in V_N$ est dit *axiome* de la grammaire ; P est un ensemble fini de *productions* (ou règles de réécritures).
On a la propriété $V_T \cap V_N = \varnothing$ et on note $V = V_T \cup V_N$.

---

[1]Une grande partie du contenu de cette annexe s'inspire du polycopié de Bernard Vauquois : "Calculabilité des langages", utilisé comme support du cours sur les langages formels en licence d'informatique à l'université de Grenoble I.

On note L(G) le langage caractérisé par la grammaire, on a $L(G) \subset V_T^*$.

Une production est de la forme :
$\alpha \rightarrow \beta$, où $\alpha \in V^+$ et contient au moins un symbole de V_N et $\beta \in V^*$.
Les productions servent à construire les dérivations.

A.1.3. Dérivation

Soit X et Y deux mots de V^*. On dit que Y *dérive directement* de X s'il existe une règle $\alpha \rightarrow \beta$ de G et deux mots ψ et ϕ de V^* tels que $X = \psi\alpha\phi$ et $Y = \psi\beta\phi$.
On dit que Y *dérive* de X dans G et on note $X \Rightarrow Y$ s'il existe une suite de dérivations directes permettant d'obtenir Y à partir de X. Autrement dit :
$X \Rightarrow Y$, s'il existe $\phi_0, \phi_1, ..., \phi_{n+1} \in V^*$ tels que $\phi_0 = X$, $\phi_{n+1} = Y$ et :
$\forall i, 0 \leq i \leq n$, ϕ_{i+1} dérive directement de ϕ_i.

A.1.4. Langage engendré par une grammaire G

On note L(G) et on appelle *langage engendré par la grammaire G* le sous ensemble de V_T^* tel que : $\forall \omega \in L(G)$, $S \Rightarrow \omega$. L(G) est donc l'ensemble de toutes les chaînes de V_T^* qui dérivent de l'axiome S de la grammaire.

A.1.5. Classification de Chomsky

Chomsky définit quatre types de grammaires syntagmatiques. Les grammaires définies ci-dessus sont dites de *type 0*. On obtient les grammaires de type 1, 2 et 3 en imposant des contraintes de plus en plus sévères sur les productions.

Grammaires de type 1 et grammaires sous-contextes :
Les grammaires de type 1 sont des grammaires dans lesquelles on impose, pour toute production $\alpha \rightarrow \beta$, que $|\alpha| \leq |\beta|$ (où $|\alpha|$ dénote la longueur de α). Cette contrainte interdisant de produire la chaîne vide ε, on définit les grammaires de *type 1 étendu* dans lesquelles on autorise la présence de la production $S \rightarrow \varepsilon$, sous la contrainte que S n'apparaît pas en partie droite d'une production.
Toute grammaire de type 1 est équivalente à une grammaire (de type 1), dite grammaire *sous-contexte*, où toute production est de la forme : $\psi A\phi \rightarrow \psi\omega\phi$ avec $\omega \in V^+$, $\psi, \phi \in V^*$ et $A \in V_N$. On définit les grammaires sous-contextes étendues de la même façon que ci-dessus.

Grammaires de type 2 ou hors-contextes :
Elles sont définies comme les grammaires sous-contextes mais on ajoute la contrainte : $\psi = \phi = \varepsilon$. Les productions sont donc de la forme $A \rightarrow \omega$ avec $\omega \in V^+$, $A \in V_N$. On définit le type 2 étendu (hors-contexte étendue) de la même façon que pour les grammaires de type 1. On définit également le *type 2 généralisé* (hors-contexte généralisée) dans lequel on permet $\omega = \varepsilon$. En général, quand on parle de grammaire hors-contexte, il s'agit de grammaire hors-contexte généralisée.

Grammaires de type 3 :
Elles sont définies comme les grammaires de type 2 en ajoutant la contrainte : $\omega = a$ ou $\omega = aB$ avec $a \in V_T$ et $B \in V_N$. On définit, comme pour le type 2, le type 3 étendu ($S \rightarrow \varepsilon$, sous la contrainte que S n'apparaît pas en partie droite d'une production) et le type 3 généralisé (dans lequel on permet $\omega = \varepsilon$).

Les grammaires les plus utilisées pour le traitement des langues naturelles sont les grammaires de type 2. On va dans la suite se limiter à cette classe de grammaires. On appelle langage hors-contexte un langage qui peut être engendré par une grammaire hors-contexte.

A.1.6. Arbre de dérivation

Un *arbre* A est un graphe fini (N,R) où:
N est un ensemble de sommets (nœuds) fini et non vide
R est une relation, $R \subset N \times N$ telle que :
i) le graphe est connexe (on a toujours un chemin d'un nœud à un autre)
ii) $\forall x \in N$, il existe au plus un sommet $y \in N$ tel que xRy. y est appelé *père* de x et x est un *fils* de y. Tout sommet a donc au plus un père.
iii) il existe un unique sommet $r \in N$ tel qu'il n'existe aucun sommet $y \in N$ tel que rRy. Ce sommet est appelé *racine* de l'arbre.
Les éléments x de N qui n'ont pas de fils sont appelés *feuilles* de l'arbre.

Un *arbre étiqueté* sur E est un arbre muni d'une application de N dans E.

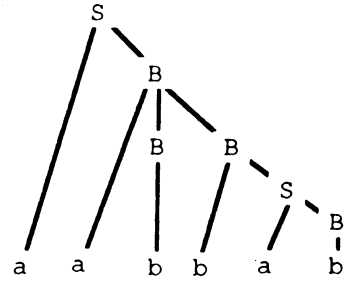
Un *arbre de dérivation* sur une grammaire hors-contexte $G = (V_T, V_N, S, P)$ est un arbre ordonné et étiqueté sur $V \cup \{\varepsilon\}$ (où $V = V_T \cup V_N$) construit de la manière suivante :
- on étiquette la racine de l'arbre avec S ;
- pour toute dérivation directe $A \rightarrow a_1a_2...a_n$, on crée n fils du nœud étiqueté par A : $(x_1, x_2, ..., x_n)$ et on étiquette chaque x_i avec a_i.

Il résulte de cette définition que si un nœud porte un symbole terminal (un élément de V_T) ou ε alors ce nœud est une feuille.
Si la dérivation donne une chaîne du langage L(G) alors toute feuille de l'arbre est étiquetée par un symbole de $V_T \cup \{\varepsilon\}$.

Exemple :
$G = (\{a, b\}, \{S, A, B\}, S, P)$ où P est l'ensemble des règles :

$S \rightarrow aB$ (1); $S \rightarrow bA$ (2); $A \rightarrow aS$ (3); $B \rightarrow bS$ (4)
$A \rightarrow bAA$ (5); $A \rightarrow a$ (6); $B \rightarrow aBB$ (7); $B \rightarrow b$ (8)

A la dérivation :
$S \xrightarrow{(1)} aB \xrightarrow{(7)} aaBB \xrightarrow{(8)} aabB \xrightarrow{(4)} aabbS \xrightarrow{(1)} aabbaB \xrightarrow{(8)} aabbab$
on associe l'arbre :



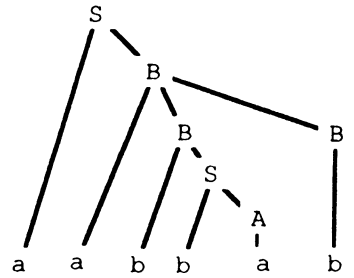
L'arbre de dérivation d'une chaîne du langage est aussi appelé *structure syntaxique* de la chaîne.

A.1.7. Ambiguïté

Etant donnée une grammaire hors contexte G, s'il existe au moins un mot de L(G) qui possède au moins deux structures syntaxiques différentes, on dit que la grammaire G est ambiguë. Un langage L est dit ambigu si on ne peut pas trouver une grammaire G non ambiguë telle L(G) = L.

Exemple :

La grammaire de l'exemple ci-dessus est ambiguë, en effet le mot aabbab, dont on a donné ci-dessus une structure syntaxique, possède également la structure syntaxique suivante :



correspondant à la dérivation :

$S \xrightarrow{(1)} aB \xrightarrow{(7)} aaBB \xrightarrow{(4)} aabSB \xrightarrow{(2)} aabbAB \xrightarrow{(6)} aabbaB \xrightarrow{(8)} aabbab$

Un langage pour lequel il n'est pas possible de trouver une grammaire non ambigüe est dit *intrinsèquement ambigu*. Ce n'est pas le cas du langage ci dessus puisque qu'il peut être engendré par la grammaire non ambigüe contenant l'ensemble de productions :

$S \rightarrow aBS$; $S \rightarrow aB$; $S \rightarrow bAS$; $S \rightarrow bA$

$A \rightarrow bAA$; $A \rightarrow a$; $B \rightarrow aBB$; $B \rightarrow b$

A.1.8. Automates à pile

Une grammaire sert à décrire (ou plus simplement à engendrer) un langage. La question qu'on se pose est la suivante : étant donnée une grammaire G et une chaîne α de symboles de V_T, est-ce que α est un mot de L(G) ? Une première méthode pour répondre à cette question est d'engendrer toutes les chaînes ayant la même longueur que ω et de tester si ω est dans l'ensemble obtenu. Une deuxième méthode, plus efficace, consiste à définir un dispositif abstrait, appelé *automate à pile*.

Un automate à pile M est un 7-uple (V_T, Q, q_1, F, V_P, P_0, T) où :

V_T est le *vocabulaire terminal* (dit aussi *vocabulaire d'entrée*),

Q est l'ensemble des *états* de l'automate,

$q_1 \in Q$ est *l'état initial*,

$F \subset Q$ est l'ensemble des *états finaux*,

V_P est le *vocabulaire de pile*,

$P_0 \in V_P$ est le *symbole initial de pile*,

T est une application de $Q \times V_T \cup \{\varepsilon\} \times V_P$ dans l'ensemble des parties finies de $Q \times V_P^*$, ainsi $T(q, a, P) = \{ (r_1, \omega_1), (r_2, \omega_2), ..., (r_n, \omega_n)\}$.

Une représentation commode pour comprendre le fonctionnement d'un automate à pile est donnée figure A.1.

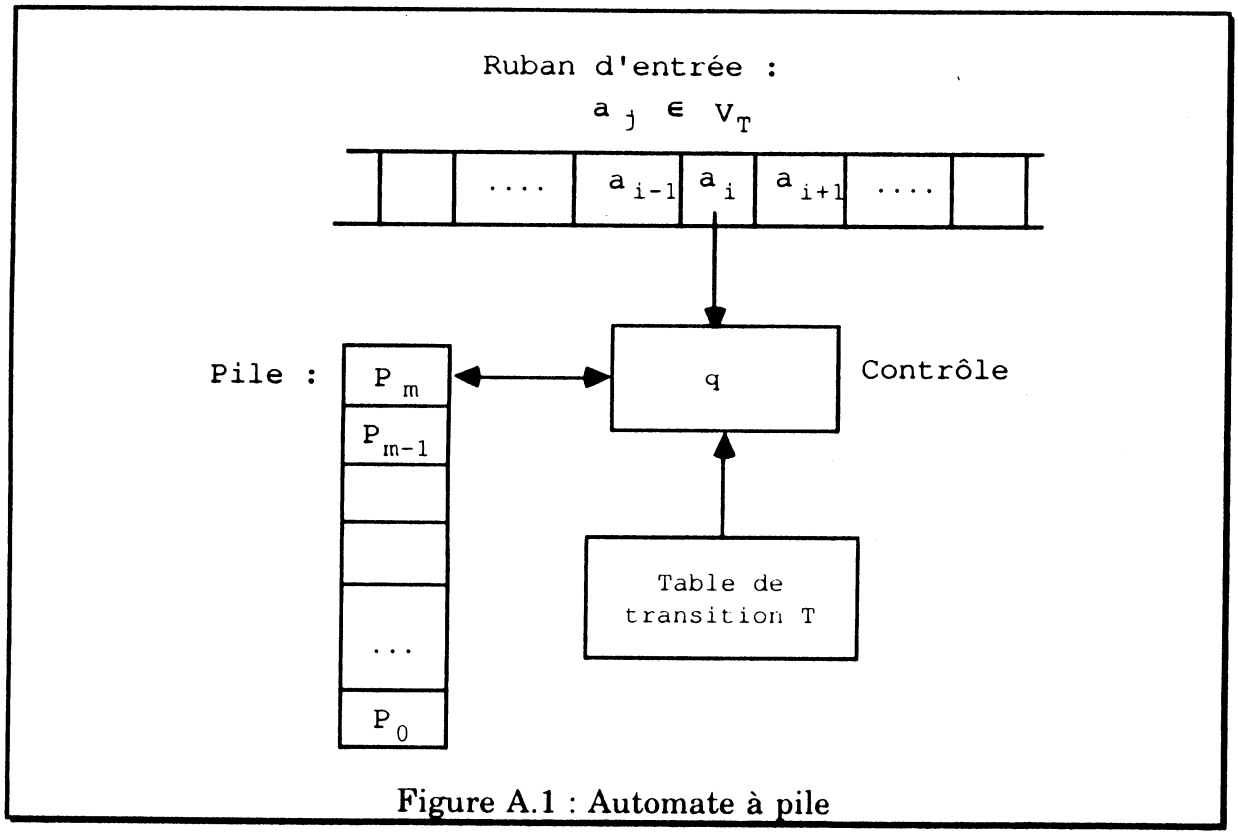


Figure A.1 : Automate à pile

L'unité de contrôle, qui se trouve dans un état $q \in Q$, est munie d'une tête de lecture sur le ruban d'entrée, qui contient une chaîne $\omega \in V_T^*$ à reconnaître et d'une tête de lecture-écriture sur un autre ruban appelé pile. L'unité de contrôle peut consulter le ruban d'entrée et lire le symbole sous la tête de lecture, dit *symbole courant*. Elle peut également commander l'avance de la tête de lecture sur le symbole suivant. Le fonctionnement de la pile est celui d'une pile classique. On appelle *mot de pile* la chaîne se trouvant dans la pile. La table de transition de l'application T permet à l'unité de contrôle de décider, en fonction de la configuration courante, quelle sera la configuration successeur.

Une configuration de l'automate est un quadruplet (u, q, v, β) où uv est le contenu du ruban d'entrée ($uv \in V_T^*$), q est l'état courant de l'automate et β est le mot de pile ($\beta \in V_P^*$). Une configuration C_{i+1} est une configuration successeur d'une configuration C_i si :

- $C_i = (u, q, av, P_m\beta)$, où $a \in V_T$, $P_m \in V_P$ et :
  $(r, \omega) \in T(q, a, P_m)$ et $C_{i+1} = (ua, r, v, \omega\beta)$, intuitivement cela veut dire qu'on avance la tête de lecture du ruban d'entrée sur le symbole suivant et qu'on remplace le symbole en sommet de pile par ω ;
- ou bien $C_i = (u, q, v, P_m\beta)$, où $P_m \in V_P$ et :
  $(r, \omega) \in T(q, \varepsilon, P_m)$ et $C_{i+1} = (u, r, v, \omega\beta)$, dans ce cas, on empile simplement ω sans avancer sur le ruban d'entrée.

On note $C_i \triangleright C_{i+1}$. On note également $C \triangleright D$ s'il existe une suite $C_0, C_1, ..., C_{n+1}$, telle que $C_0 = C$, $C_{n+1} = D$ et $\forall i, 0 \leq i \leq n, C_i \triangleright C_{i+1}$. Etant donnée une configuration, il peut exister 0, 1 ou plusieurs configurations successeurs.
La *configuration initiale*, notée CI est égale à $(\varepsilon, q_1, \omega, P_0)$.

On peut définir deux langages reconnus par un automate à pile M :

- Acceptation par critère d'état final :
  $L(M) = \{x / CI \triangleright (x, q', \varepsilon, \beta) \text{ et } q' \in F\}$
- Acceptation par critère de pile vide (le langage est noté Nul(M)) :
  $Nul(M) = \{x / CI \triangleright (x, q', \varepsilon, \varepsilon) \text{ et } q' \in Q\}$

On peut montrer que pour tout automate à pile M, il existe un automate M' tel que Nul(M) = L(M') et un automate M'' tel que L(M) = Nul(M'').

Un automate à pile M est dit *déterministe* si les conditions suivantes sont vérifiées :

1- $\forall q \in Q, P \in V_P, a \in V_T \cup \{\varepsilon\}$ $T(q, a, P)$ contient au plus un élément.

2- $\forall q \in Q, P \in V_P, T(q, \varepsilon, P) \neq \varnothing$ entraîne $T(q, a, P) = \varnothing$ pour tout $a \in V_T$.

Exemple :
Soit $L = a^n b^n$ le langage défini sur $V_T = \{a, b\}$.
L'automate $M = (V_T, \{q_1, q_2\}, q_1, \varnothing, \{P_0, A\}, P_0, T)$ avec :
$T(q_1, a, P_0) = \{(q_1, A)\}$
$T(q_1, a, A) = \{(q_1, AA)\}$
$T(q_1, b, A) = \{(q_2, \varepsilon)\}$
$T(q_2, b, A) = \{(q_2, \varepsilon)\}$
est tel que Nul(M) = L.
Si le ruban d'entrée contient aabb, on va avoir la suite de configurations :
$(\varepsilon, q_1, aabb, P_0) \triangleright (a, q_1, abb, A) \triangleright (aa, q_1, bb, AA)$
$\triangleright (aab, q_2, b, A) \triangleright (aabb, q_2, b, \varepsilon)$

On peut montrer que la famille des langages engendrés par les grammaires hors-contextes est équivalente à la famille des langages reconnus par les automates à pile. Le problème est de déterminer, étant donné une grammaire G, l'automate à pile M qui permet de reconnaître L(G). Nous allons voir dans les deux paragraphes suivants deux méthodes permettant de trouver de manière systématique l'automate associé à une grammaire. Dans les deux cas la méthode est limitée à une certaine classe de grammaires hors-contextes.

A.2. Analyse descendante LL

Appelée également *analyse prédictive*, cette méthode permet de construire des automates à pile déterministes pour une certaine classe de grammaires, dites grammaires LL(k) pour *Left to right scanning, Leftmost derivation*, et lecture de k symboles du ruban d'entrée. Nous nous limitons ici à la présentation de la méthode pour les grammaires LL(1).

Il s'agit d'une méthode descendante, c'est-à-dire qu'on utilise la grammaire de manière générative : on construit une dérivation en partant de l'axiome et en descendant vers les symboles terminaux. L'idée est de pouvoir déterminer, étant donné un symbole non terminal A à réécrire et un symbole de la chaîne d'entrée, quelle est la règle à appliquer si plusieurs sont possibles ($A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$,..., $A \rightarrow \alpha_n$).

Soit une grammaire $G = (V_T, V_N, S, P)$, on construit un automate à pile déterministe un peu particulier de la manière suivante : on ajoute le symbole \$ à la fin du ruban d'entrée et on initialise la pile avec la chaîne S\$. On suppose donnée une table de transition notée T (par analogie avec l'application T ci-dessus) qui détermine, pour un terminal a et un non-terminal A donnés, quelle est la production $A \rightarrow \alpha$ à utiliser. On a $T : V_T \cup \{\$\} \times V_N \rightarrow P$. L'automate n'a qu'un seul état et une configuration sera donc un triplet $(u, av, A\beta)$; si ω est la chaîne à reconnaître, la configuration initiale sera $(\varepsilon, \omega\$, S\$)$.

Le fonctionnement de l'automate est alors le suivant :
Soit $(u, av, A\beta)$ la configuration courante, on a trois cas :

1 - si $A = a = \$$, alors succès (la chaîne u est une chaîne du langage, on a $v = \varepsilon$)
2 - si $A = a \neq \$$ alors on avance sur le symbole suivant du ruban d'entrée et on dépile A (on a reconnu le symbole terminal a), on obtient donc la configuration (ua, v, β)
2 - si $A \in V_N$ alors on consulte T(a, A) : si T(a, A) n'est pas définie alors erreur sinon on a $T(a, A) = A \rightarrow \alpha$, on dépile A et on empile α. On obtient donc la configuration $(u, av, \alpha\beta)$.

Pour construire la table T, on définit deux applications *Premier* et *Suivant*.

A.2.1. Application Premier

On note $V = V_N \cup V_T$. Premier est une application de V^+ dans l'ensemble des parties finies de $V_T \cup \{\varepsilon\}$ définie pour tout α de V^+ par :
$Premier(\alpha) = \{x \in V_T / \exists \beta \in V_T^* / \alpha \Rightarrow x\beta$ et ε si $\alpha \Rightarrow \varepsilon\}$
Premier(α) est donc l'ensemble des symboles terminaux qui peuvent commencer une chaîne dérivable à partir de α.

On définit de manière plus précise l'algorithme de calcul de Premier(A) pour $A \in V$:
1 - si $A \in V_T$, Premier(A) := {A}
2 - si $A \in V_N$ alors
  3 - Premier(A) := $\varnothing$
  4 - si $A \rightarrow \varepsilon$ alors Premier(A) := {ε}
  5 - pour chaque production $A \rightarrow X_1X_2...X_n$ de G, $X_i \in V$ faire
  Soit le plus grand entier k, $1 \leq k \leq n$ tel que $X_1X_2...X_k \Rightarrow \varepsilon$,
    6 - Premier(A) := Premier(A) $\cup$ ($\overset{k}{\underset{i=1}{\cup}}$Premier($X_i$))
    7 - si k = n alors Premier(A) := Premier(A) $\cup$ {ε}

Pour calculer Premier(α), $\alpha \in V^+$, on utilise l'algorithme :
Soit $\alpha = X_1X_2...X_n$, $X_i \in V$ et soit le plus grand entier k, $1 \leq k \leq n$ tel que $X_1X_2...X_k \Rightarrow \varepsilon$,
1 - Premier(α) := ($\overset{k}{\underset{i=1}{\cup}}$Premier($X_i$))
2 - si k = n alors Premier(α) := Premier(α) $\cup$ {ε}

Exemple :
Soit la grammaire $G = (V_T, V_N, E, P)$ avec :
V_T = {id, +, *, (,)} V_N = {E, T, E', F, T'}
On donne les productions de P ci-dessous en les numérotant :

1 : $E \rightarrow TE'$	2 : $E' \rightarrow +TE'$	3 : $E' \rightarrow \varepsilon$
4 : $T \rightarrow FT'$	5 : $T' \rightarrow *FT'$	6 : $T' \rightarrow \varepsilon$
7 : $F \rightarrow$ (E)	8 : $F \rightarrow$ id	

On a bien sûr, pour tout $a \in V_T$, Premier(a) = {a}.
Premier(E) est obtenu en calculant Premier(T) (règle 1) et éventuellement (si $T \Rightarrow \varepsilon$), Premier(E').
Premier(T) est obtenu en calculant Premier(F) (règle 4) et éventuellement (si $F \Rightarrow \varepsilon$), Premier(T').
Premier(F) = {(, id} (règles 7 et 8), on en déduit :
Premier(T) = Premier(F) = {(, id}
Premier(E) = Premier(T) = {(, id}.
Premier(E') = {+, ε} (règles 2 et 3)
Premier(T') = {*, ε} (règles 5 et 6)

A.2.2. Application Suivant

Suivant est une application de V_N dans l'ensemble des parties finies de $V_T \cup \{\$\}$ définie pour tout A de V_N par :

Suivant(A) = $\{a \in V_T / S \Rightarrow \alpha A a \beta, \alpha, \beta \in V^*\}$

Il s'agit donc, pour un non terminal donné, de l'ensemble des terminaux (ou \$), susceptibles de suivre ce terminal dans une chaîne dérivable à partir de l'axiome.

On définit ci dessous l'algorithme permettant de calculer les ensembles Suivant pour tous les éléments de V_N :

0 - Initialiser à $\varnothing$ tous les ensembles Suivant, sauf Suivant(S) := {\$}
1 - Tant que on ajoute un symbole à un quelconque des ensembles Suivant faire
  2 - si $A \rightarrow \alpha B \beta \in P$ alors
    3 - Suivant(B) := Suivant(B) $\cup$ Premier(β) - {ε}
  4 - si $A \rightarrow \alpha B \in P$ ou si $A \rightarrow \alpha B \beta \in P$ avec $\varepsilon \in$ Premier(β) alors
    5 - Suivant(B) := Suivant(B) $\cup$ Suivant(A)

Exemple :
Avec la grammaire de l'exemple ci-dessus et les ensembles Premier obtenus, on a :

Suivant(E) = {\$,)} (Etape 0 et règle 7)
Suivant(E') = Suivant(E) (règles 1 et 2)
Suivant(T) = Premier(E')-{ε} $\cup$ Suivant(E') $\cup$ Suivant(E) (règles 1 et 2, et E' $\Rightarrow \varepsilon$)
= {+, \$,)}
Suivant(T') = Suivant(T) (règle 4)
Suivant(F) = Premier(T')-{ε} $\cup$ Suivant(T') $\cup$ Suivant(T) (règles 4 et 5, et T' $\Rightarrow \varepsilon$)
= {*, +, \$,)}

A.2.3. Construction de la table de transition T

On construit l'application T en fonction des applications Premier et Suivant en exploitant l'idée suivante :

si $A \rightarrow \alpha$ et que $a \in$ Premier(α) alors T(a, A) = $A \rightarrow \alpha$,

si $\alpha \Rightarrow \varepsilon$, T(a, A) = $A \rightarrow \alpha$ si $a \in$ Suivant(A).

Algorithme de construction de l'application T :
Données :
  Une grammaire G et les applications Premier et Suivant
Résulats :
  La table de transition T
1 - Pour chaque production $A \rightarrow \alpha$ de G faire
    2 - pour chaque $a \in$ Premier(α) faire
      3 - T(a, A) := $A \rightarrow \alpha$
    4 - si $\varepsilon \in$ Premier(α) ou si $\alpha = \varepsilon$ alors
      5 - pour chaque $b \in$ Suivant(A) faire
        6 - T(b, A) := $A \rightarrow \alpha$

Si dans les étapes 3 ou 6 il y a un conflit, alors l'algorithme échoue et la grammaire n'est pas LL(1). Toutes les entrées non-définies par l'algorithme ci-dessus donneront une erreur si l'automate tente de les utiliser.

Exemple :
On reprend la grammaire et les ensembles Premier et Suivant des exemples précédents. On a alors l'application T :

T	id	+	*	(	)	$
E	E → TE'			E → TE'		
T	T → FT'			T → FT'		
E'		E' → +TE'			E' → ε	E' → ε
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Exemple d'évolution de l'automate :
On donne ci-dessous les configurations successives de l'automate pour la reconnaissance de la chaîne : `id+id*id$`

CI = (ε, `id+id*id$`, E$) ⊳ (ε, `id+id*id$`, TE'$) ⊳ (ε, `id+id*id$`, FT'E'$)
⊳ (ε, `id+id*id$`, `id`T'E'$) ⊳ (`id`, `+id*id$`, T'E'$) ⊳ (`id`, `+id*id$`, E'$)
⊳ (`id`, `+id*id$`, +TE'$) ⊳ (`id+`, `id*id$`, TE'$) ⊳ (`id+`, `id*id$`, FT'E'$)
⊳ (`id+`, `id*id$`, FT'E'$) ⊳ (`id+`, `id*id$`, `id`T'E'$) ⊳ (`id+id`, `*id$`, T'E'$)
⊳ (`id+id`, `*id$`, *FT'E'$) ⊳ (`id+id*`, `id$`, FT'E'$) ⊳ (`id+id*`, `id$`, `id`T'E'$)
⊳ (`id+id*id`, $, T'E'$) ⊳ (`id+id*id`, $, E'$) ⊳ (`id+id*id`, $, $) ⊳ **Succès**

A.3. Analyse ascendante LR

Il s'agit d'une méthode d'analyse ascendante dite analyse par décalage-réduction. On l'appelle analyse LR pour *Left to right scanning* et *Rightmost derivation in reverse*. Cette méthode est très efficace et accepte une plus grande classe de grammaires que la méthode LL mais elle est très difficile à mettre en œuvre à la main. Cependant, on peut automatiser la production d'analyseurs LR à partir d'une grammaire pour peu que cette grammaire ait certaines propriétés. Dans le cadre de l'analyse des langages de programmation, on obtient des analyseurs déterministes sans retour arrière.

Un analyseur LR pour une grammaire $G = (V_T, V_N, S, P)$, est un automate à pile particulier. Comme pour la méthode LL, on ajoute un $ à la fin de la chaîne d'entrée. Le vocabulaire de pile est constitué de $Q \cup V_N \cup V_T$ où Q est l'ensemble des états de l'automate.

L'application de transition T est traditionnellement décomposée en deux tables Action et Successeur. Ces tables sont en fait la représentation des fonctions :
Action :

$$Q \times V_T \cup \{\$\} \rightarrow \{\text{(décaler q, q} \in Q\text{), (réduire par A}\rightarrow\alpha\text{), Succès, Erreur)}$$

Successeur :

$$Q \times V_N \rightarrow Q$$

Nous donnons ci-dessous la description de l'algorithme de l'automate ; nous nous somme limité à un accepteur produisant comme résultat une indication de succès ($\omega \in L(G)$) ou d'erreur ($\omega \notin L(G)$).

Algorithme d'analyse LR :
Données :

une chaîne à analyser ω à laquelle on ajoute un \$.
les tables Action et Successeur pour une grammaire G.

Résultat :

Succès ou Erreur

0 - Initialiser avec q_0 (état initial) en sommet de pile et ω\$ sur le ruban d'entrée.
1 - Tanque ni succès ni erreur faire :
Soit q l'état en sommet de pile et a le symbole courant du ruban d'entrée.
2 - Si Action[q, a] a pour valeur :

(décaler q') alors empiler a puis q', avancer la tête de lecture du ruban d'entrée

(réduire $A \rightarrow \alpha$) alors dépiler 2 fois autant d'éléments de la pile que de symboles de α (les symboles de α et les états intermédiaires).
Soit q' le nouvel état en sommet de pile, empiler A puis Successeur[q', A].

Succès alors arrêt.
Erreur alors arrêt.

La construction d'un analyseur de ce type repose donc sur la construction des tables Action et Successeur. Nous allons décrire ci-dessous une méthode de construction connue sous le nom de SLR (pour *Simple* LR). Les analyseurs construit avec cette méthode sont appelés *analyseurs SLR(1)* ou parfois *analyseurs LR(0)*.

A.3.1. Construction des tables SLR

On appelle *item LR(0)* ou tout simplement *item* d'une grammaire G, une production de G à laquelle on ajoute un point pour repérer une position dans la partie droite. Tous les symboles à gauche du point sont les symboles déjà reconnus alors que les symboles à droite sont ceux qui restent à reconnaître.
Exemples :

Une production $A \rightarrow XYZ$ fournit 4 items :
$A \rightarrow .XYZ$, $A \rightarrow X.YZ$, $A \rightarrow XY.Z$, $A \rightarrow XYZ.$
$A \rightarrow \varepsilon$ fournit un seul item $A \rightarrow \varepsilon$.

L'idée de la méthode SLR est de regrouper les items de la grammaire en ensembles où chaque ensemble représente un ensemble de préfixes valides pour les chaînes du langage. Chaque ensemble d'items représentera un état différent de l'analyseur.

On construit une collection d'ensembles d'items LR(0), appelée *collection canonique LR(0)*, qui permet ensuite de construire les tables de l'analyseur

SLR. Pour construire cette collection C, on augmente la grammaire G initiale en une grammaire G' de la manière suivante :

- on ajoute un symbole non terminal S' qui devient l'axiome de la grammaire ;
- on ajoute la production S' → S à l'ensemble P des productions de G.

On peut montrer facilement que L(G') = L(G). Le but de cette opération est de pouvoir arrêter l'analyse avec succès dès qu'on réduit S en S'.

Pour construire la collection canonique LR(0) pour une grammaire augmentée G', on définit deux opérations : *Fermeture* et *Transition*.

On définit l'opération *Fermeture* sur les ensembles d'items :

Algorithme de calcul d'une Fermeture :
Données :
une grammaire G et l'ensemble I(G) des items LR(0) de G
un ensemble I d'items LR(0).
Résultat :
un ensemble d'items Fermeture(I) qui est la fermeture de l'ensemble I.
0 - Fermeture(I) := I
1 - Tant que on ajoute de nouveaux items à Fermeture(I) faire
2 - Si A → α.Bβ ∈ Fermeture(I), α, β ∈ V^*, B ∈ V_N alors
3 - pour toute production B → ω de P faire
4 - Fermeture(I) := Fermeture(I) ∪ { B → .ω }

Intuitivement, si A → α.Bβ est dans Fermeture(I), cela veut dire qu'on se trouve dans une situation où on a reconnu une sous-chaîne de la chaîne d'entrée qui dérive de α et le début de la chaîne restant à analyser dérive de Bβ et donc peut dériver de tout ω tel que B → ω.

Exemple :
Soit la grammaire augmentée G', définie par :
V_T = {+, *, (,), id }, V_N = {E', E, T, F}, l'axiome est E'
P = { E' → E,
E → E+T, E → T,
T → T*F, T → F,
F → (E), F → id }
Cette grammaire permet d'engendrer des expressions arithmétiques simples comme `id + id * id * (id + id)`.
Si I = {E' → .E} alors Fermeture(I) prend les valeurs successives :

{E' → .E}	Etape 0 de l'algorithme
{E' → .E, E → .E+T, E → .T}	Etapes 2 et 3 avec B = E
{E' → .E, E → .E+T, E → .T, T → .T*F, T → .F}	Etapes 2 et 3 avec B = T
{E' → .E, E → .E+T, E → .T, T → .T*F, T → .F, F → . (E) , F → .id}	Etapes 2 et 3 avec B = F

Cette fermeture traduit le fait que pour reconnaître une expression E on peut être amené à reconnaître l'une des chaînes : E+T, T, T*F, F, (E) ou id.

On définit l'opération *Transition* qui détermine, étant donné un ensemble d'items I et un symbole X de V un autre ensemble d'items J, qui est le successeur de I si on reconnaît le symbole X.
Transition(I, X) est définie comme suit : soit I_X le sous-ensemble de I des items de la forme $A \rightarrow \alpha.X\beta$, $\alpha, \beta \in V^*$, $X \in V$. Ces items indiquent que l'on est en passe de reconnaître un X. On transforme les éléments de I_X en décalant le point d'un cran vers la droite (on indique qu'on a reconnu le X). On obtient un ensemble d'items de la forme $A \rightarrow \alpha X.\beta$. Transition(I, X) est alors défini comme la fermeture de cet ensemble d'items.

Exemple :

Avec la grammaire de l'exemple ci-dessus, si I = { E' → E., E → E .+T } alors Transition(I, +) est définie comme Fermeture({ E → E+.T }), c'est-à-dire comme l'ensemble d'items :
{ E → E+.T, T → .T*F, T → .F, F → . (E) , F → .id }
Cette opération traduit le fait qu'après avoir reconnu un +, on peut être amené à reconnaître une des chaînes : T, T*F, F, (E) ou id.

Algorithme de construction d'une collection canonique LR(0) pour une grammaire augmentée G'.

Données :

Une grammaire G'
Les deux opérations Fermeture et Transition

Résultats :

Une collection C = {I_0, I_1, ..., I_n } d'ensembles d'items LR(0)

0 - C := { I_0 } avec I_0 = Fermeture({ S' → .S }).
1 - Tant que on ajoute un nouvel ensemble d'items à C faire
  2 - Pour chaque paire (I, X) où I ∈ C et X ∈ V faire
    3 - Si Transition(I, X) ≠ ∅ alors
      4 - C := C ∪ Transition(I, X)

Exemple :

On donne ci-dessous les différentes transitions calculées pour la construction de la collection canonique LR(0) associée à la grammaire d'expressions des exemples ci-dessus.

I_0 = { E' → .E, E → .E+T, E → .T, Etape 0
T → .T*F, T → .F, F → . (E) , F → .id }

Les transitions possibles à partir de I_0 concernent les symboles E, T, F, (et id. On calcule donc :

Transition(I_0, E) = Fermeture({E' → E., E → E.+T})
= {E' → E., E → E.+T} = I_1

Transition(I_0, T) = Fermeture({T → T.*F, E → T.})
= {T → T.*F, E → T.} = I_2

Transition(I_0, F) = Fermeture({T → F.}) = {T → F.} = I_3
Transition(I_0, () = Fermeture({F → (.E)})
= {F → (.E), E → .E+T, E → .T, T → .T*F, T → .F,
F → .(E), F → .id} = I_4
Transition(I_0, id) = Fermeture({T → id.}) = {T → id.} = I_5
On a donc ajouté 5 ensembles d'items à C, on répète le processus pour I_1 :
Transition(I_1, +) = Fermeture({E → E+.T})
= {E → E+.T, T → .T*F, T → .F, F → .(E), F → .id} = I_6
puis pour I_2 :
Transition(I_2, *) = Fermeture({T → T*.F})
= {T → T*.F, F → .(E), F → .id} = I_7
Aucune transition n'est possible pour I_3, on traite donc I_4 :
Transition(I_4, E) = Fermeture({F → (E.), E → E.+T})
= {F → (E.), E → E.+T} = I_8
Transition(I_4, T) = Fermeture({E → T., T → T.*F}) = I_2
Transition(I_4, F) = Fermeture({T → F.}) = I_3
Transition(I_4, () = Fermeture({F → (.E)}) = I_4
Transition(I_4, id) = Fermeture({F → id.}) = I_5
I_5 ne donne rien, on traite I_6 :
Transition(I_6, T) = Fermeture({E → E+T., T → T.*F})
= {E → E+T., T → T.*F} = I_9
Transition(I_6, F) = Fermeture({T → F.}) = I_3
Transition(I_6, () = Fermeture({F → (.E)}) = I_4
Transition(I_6, id) = Fermeture({F → id.}) = I_5
puis I_7 :
Transition(I_7, F) = Fermeture({T → T*F.}) = {T → T*F.} = I_{10}
Transition(I_7, () = I_4
Transition(I_7, id) = Fermeture({F → id.}) = I_5
et enfin I_8 et I_9 (I_{10} et I_{11} ne donneront rien) :
Transition(I_8,)) = Fermeture({F → (E).}) = {F → (E).} = I_4
Transition(I_8, +) = Fermeture({E → E+.T}) = I_6
Transition(I_9, *) = Fermeture({T → T*.F}) = I_7

Pour construire les tables Action et Successeur de l'analyseur SLR, on va exploiter la collection canonique de la grammaire et la fonction Transition. On utilise également la fonction Suivant qui calcule les symboles terminaux qui peuvent suivre un symbole non terminal (voir §A.2.2.). Chaque élément de la collection représente un état de l'automate, l'état initial étant donné par I_0.

Algorithme de construction des tables d'analyse SLR
Données :
Une grammaire augmentée G'
La collection canonique LR(0) C de G'

Les fonctions Suivant et Transition

Résultat :

Les tables Action et Successeur pour G'.

0 - Initialiser Action et Successeur avec Erreur partout.

1 - Construire l'état q à partir de $I_q \in C$, pour chaque q, $0 \leq q \leq n$

2 - Pour chaque état q :

3 - si $A \rightarrow \alpha.a\beta \in I_q$, $a \in V_T$, et Transition(I_q, a) = I_s, alors

4 - Action[q, a] := (décaler s)

5 - si $A \rightarrow \alpha. \in I_q$, $A \neq S'$ alors

6 - Pour tout a dans Suivant(A) faire

7 - Action[q, a] := (réduire par $A \rightarrow \alpha$)

8 - si $S \rightarrow S'. \in I_q$, $A \neq S'$ alors

9 - Action[q, $] := Succès

10 - Pour chaque A, $A \in V_N$ faire

11- si Transition(I_q, A) = I_s, alors

12 - Successeur[q, A] := s

Si pour un état q et un terminal a, deux au moins des trois étapes 4, 7 et 9 s'appliquent, alors l'algorithme échoue et on dit que la grammaire n'est pas SLR(1).

Exemple :

On donne figure A.2 les tables d'analyse SLR de la grammaire d'expressions utilisée dans les exemples. On numérote les règles de la grammaire :

0 : E' → E,

1 : E → E+T, 2 : E → T,

3 : T → T*F, 4 : T → F,

5 : F → (E), 6 : F → `id`

On note `Re n` pour (réduire par $A \rightarrow \alpha$) si la règle $A \rightarrow \alpha$ porte le numéro `n` et `Emp s` pour (décaler s). On a également :

Suivant(E) = {+,), $}, Suivant(F) = Suivant(T) = {*, +,), $}

A.3.2. Construction des tables pour une grammaire ambiguë

On décrit ci-dessous la construction détaillée de la table d'analyse de la grammaire utilisée au chapitre 1, figure 1.6. Cette grammaire n'est pas SLR(1) puisqu'elle entraîne un conflit dans le calcul de la table Action[1].

On a la grammaire augmentée G' :

V_T = {Nom, Det, Prep, Verbe}

V_N = {S', S, GN, GV, GP}, l'axiome est S' et on a les productions (numérotées pour la construction de la table) :

0 : S' → S	1 : S → GN GV	2 : S → S GP
3 : GN → Nom	4 : GN → Det Nom	5 : GN → GN GP

[1]On se contente de décrire ici la construction de cette table, pour l'usage qui en est fait, et notamment la manipulation de l'ambiguïté, voir chapitre 1, §1.1.3.2.

6 : GP → Prep GN 7 : GV → Verbe GN

Etat	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	Emp5			Emp4			1	2	3
1		Emp6				Succ			
2		Re2	Emp7		Re2	Re2			
3		Re4	Re4		Re4	Re4			
4	Emp5			Emp4			8	2	3
5		Re6	Re6		Re6	Re6			
6	Emp5			Emp4				9	3
7	Emp5			Emp4					10
8		Emp6		Emp11					
9		Re1	Emp7		Re1	Re1			
10		Re3	Re3		Re3	Re3			
11		Re5	Re5		Re5	Re5			

Figure A.2 : Exemple de tables d'analyse SLR

On calcule la collection canonique C de cette grammaire en commençant avec

I_0 = Fermeture({S' → .S})
= { S' → .S, S → .GN GV, S → .S GP,
GN → .Nom, GN → .Det Nom, GN → .GN GP}

Transition(I_0, S) = Fermeture({S' → S., S → S. GP})
= {S' → S., S → S. GP, GP → .Prep GN} = I_1

Transition(I_0, GN) = Fermeture({S → GN. GV, GN → GN. GP})
= { S → GN. GV, GN → GN. GP,
GP → .Prep GN, GV → .Verbe GN} = I_2

Transition(I_0, Det) = {GN → Det. Nom} = I_3

Transition(I_0, Nom) = {GN → Nom.} = I_4

Transition(I_1, GP) = {S → S GP.} = I_5

Transition(I_1, Prep) = Fermeture({GP → Prep. GN})
= { GP → Prep. GN,
GN → .Nom, GN → .Det Nom, GN → .GN GP} = I_6

Transition(I_2, Prep) = Fermeture({GP → Prep. GN}) = I_6

Transition(I_2, Verbe) = Fermeture({GV → Verbe. GN})
= { GV → Verbe. GN,
GN → .Nom, GN → .Det Nom, GN → .GN GP} = I_7

Transition(I_2, GV) = {S → GN GV.} = I_8
Transition(I_2, GP) = {GN → GN GP.} = I_9
Transition(I_3, Nom) = {GN → Det Nom.} = I_{10}
Transition(I_6, GN) = Fermeture({GP → Prep GN., GN → GN. GP})
= { GP → Prep GN., GN → GN. GP,
GP → .Prep GN} = I_{11}
Transition(I_6, Nom) = I_4
Transition(I_6, Det) = I_3
Transition(I_7, GN) = Fermeture({GV → Verbe GN., GN → GN. GP})
= { GV → Verbe GN., GN → GN. GP,
GP → .Prep GN} = I_{12}
Transition(I_7, Nom) = I_4
Transition(I_7, Det) = I_3
Transition(I_{11}, GP) = I_9
Transition(I_{11}, Prep) = I_6
Transition(I_{12}, GP) = I_4
Transition(I_{12}, Prep) = I_9

Etat	Det	Nom	Verbe	Prep	$	S	GN	GV	GP
0	Emp 3	Emp 4				1	2		
1				Emp 6	Succès				5
2			Emp 7	Emp 6				8	9
3		Emp 10							
4			Re 3	Re 3	Re 3				
5				Re 2	Re 2				
6	Emp 3	Emp 4					11		
7	Emp 3	Emp 4					12		
8				Re 1	Re 1				
9			Re 5	Re 5	Re 5				
10			Re 4	Re 4	Re 4				
11			Re 6	Re 6, Emp 6	Re 6				9
12				Re 7, Emp 6	Re 7				9

Figure A.3 : Tables Action-Successeur à entrées multiples

Pour déterminer les tables Action et Successeur il manque encore les suivants de chaque non-terminal, on a :
Suivant(S) = Suivant(GV) = {$, Prep}
Suivant(GN) = Suivant(GP) = {Verbe, Prep, $}

On construit alors la table en suivant l'algorithme donné au paragraphe précédent. Un problème se pose quand on arrive à l'état 11. En effet on a I_{11} = {GP → Prep GN., GN → GN. GP, GP → .Prep GN}, et donc, d'après l'étape 3 de l'algorithme, on a Action[11, Prep] = (décaler 6) puisque Transition(I_{11}, Prep) = I_6. Mais d'après l'étape 5 de ce même algorithme, on a Action[11, a] = (réduire par GP → Prep GN) pour tout a dans Suivant(GP) et donc pour Verbe, $ et Prep ! On retrouve le même type de conflit pour l'état 12 de l'automate. La méthode n'est donc pas applicable telle quelle sur cette grammaire car on a des entrées de la table à valeur multiple :

Action[11, Prep] = {(décaler 6), (réduire par GP → Prep GN)

Action[12, Prep] = {(décaler 6), (réduire par GV → Verbe GN)

On obtient donc la table de la figure A.3.

Annexe B

Exemples de grammaires

Cette annexe contient la liste de la signature et de la grammaire que nous avons développées pour les tests ainsi que la table d'accès aux règles. On donne également une signature et une grammaire pour l'analyse du langage sous-contexte $a^n b^n c^n$, $n \geq 1$.

B.1. Analyse de phrases simples

B.1.1. Signature

```
-- Essai de définition d'une signature pour l'analyse du
-- français.

----------------------------
-- Hiérarchie de catégories
----------------------------
-- Premier niveau :
{ Nom, Det, Adj, Pron, Verbe, Adv, Prep, Conj } < CLS

-- Noms :
{ subc,    -- Substantifs communs
  subp }   -- Substantifs propres
       < Nom

-- Déterminants
{ Art,     -- Articles
  NonQ }   -- Adjectifs non-qualificatifs
       < Det

{ Def,     -- Articles définis (le, la, les)
  Indef,   -- Articles indéfinis (une, une, des)
  Part }   -- Articles partitifs (du, de la, des)
       < Art

-- Adjectifs :
{ NonQ,    -- Adjectifs non-qualificatifs (possessifs,
           -- démonstratifs,...)
  Qual }   -- qualificatifs (incluant participes)
       < Adj
```

```
{ adjq,    -- Ajectifs qualificatifs
  PPass,   -- Participe passé
  PPres }  -- Participe présent
       < Qual

{ APoss,   -- Possessifs (mon, ton, ...)
  ADem,    -- Démonstratifs (ce, cette, ces)
  AIntEx,  -- Interrogatifs et exclamatifs (quel, quelle, ....)
  AIndef,  -- Indéfinis (certain, quelque, chaque, plusieurs,
           -- tout, beaucoup de, bien des, ...)
  AOrd,    -- Numéraux ordinaux (premier, second, troisième,...)
  ACard }  --  Numéraux  cardinaux  (un,  deux,  trois,  quatre-
vingt,...)
       < NonQ

-- Pronoms : (les pronoms relatifs sont classés avec les
-- conjonctions de subordination)
{ PPer,    -- Personnels (je, te, me, ..., lui, en, y)
  PDem,    -- Démonstratifs (celui, celle, celui-ci, ...)
  PPoss,   -- Possessifs (mien, tien, sien (tjrs précédés d'un
           -- Def))
  PInt }   -- Interrogatifs (qui, que, quoi)
       < Pron

-- Verbes :
{ Verb,    -- Forme finie d'un verbe
  Etre,    -- Auxiliaire être
  Avoir,   -- Auxiliaire avoir
  Aller,   -- Auxiliaire aller
  PPass,   -- Participe passé
  PPres,   -- Participe présent
  Infi }   -- Infinitif des verbes
       < Verbe

{ VAction,      -- Verbes qui ne sont dans aucune des catégories
                -- suivantes
  VEtat,        -- Verbes d'état (être, sembler, paraître)
  VPerc,        -- Verbes de perception (regarder, voir,
                -- sentir,...)
  VModal }      -- Verbes modaux (aimer, vouloir, pouvoir,...)
       < Verb

-- Adverbes :
{ AdvV,    -- Modifiant un verbe mais pas un adjectif (adv de
           -- manière, de temps, de lieu, de négation)
  AdvA,    -- Modifiant un adjectif mais pas un verbe (adv
           -- d'intensité)
  Ne }     -- le mot ne
       < Adv

-- Prépositions : (pas de décomposition pour l'instant)

-- Conjonctions :
{ CoCoS,   -- Conjonction de coordination simple (et, ou, donc,
           -- mais...)
  CoCoD,   -- Coordination double (ni...ni, soit...soit,...)
```

```
  CoSub } -- Conjonctions de subordination (incluant les pronoms
          -- relatifs) : que, qui, duquel, parce que, ...
        < Conj

{ CoSubV,       -- Subordination verbale invariable
  CoSubN }      -- Subordination nominale (pronom relatifs),
                -- s'accordent en genre et nombre avec
                -- l'antécédant.
         < CoSub

-----------------------------------------
-- Variables morphologiques et syntaxiques
-----------------------------------------
{ uno, dos, tre }                < personne
{ sin, plu }                     < nombre
{ mas, fem, neu }                < genre
{ mimp, msub, mind, mcond }      < mode
{ tpre, timp, tpas, tfut,
  tpas_comp, tpas_ant, tpqp,
  tfut_ant, tpas2 }              < tps
{ suj, cod, coi }                < fct_syn -- Pour les PPer
{ active, passive }              < voix

----------------------
-- Groupes syntaxiques (utilité à discuter)
----------------------
{ GN, PP }      < UL

--------------------
-- Traits sémantiques (version très provisoire)
--------------------
{ ANIMAL, HUMAIN }               < ANIME
{ DONNER, INGERER, SAVOIR }      < ACTION
COMESTIBLE                       < OBJET
EVENEMENT
NEGATION
MANIERE
LIEU
TEMPS
```

B.1.2. Grammaire

On a ajouté à cette liste en face de chaque nom de règle le numéro de la règle obtenu après compilation. Cela permet de retrouver plus facilement une règle dans la lecture de la trace donnée en annexe C.

```
-- Grammaire de construction d'arbres de dépendances du
-- français.
-- Grammaire construite de manière descendante (ceci explique le
-- caractère très général de certaines règles)
-- SIGNATURE : donnees\cadet.txs

--------------------
-- Dépendants du Nom
--------------------
```

```
-- Déterminants : (les cas particuliers de tout et autre ne sont
-- pas traités
-- Un troisième, notre deuxième, ces quelques,...
Det_Det0 [                                                          (1)
     (1:{Art;APoss;ADem}(), 2:{AIndef;AOrd;ACard})
     //
=>   (1 (2))
]

-- les trois mêmes livres, les quelques quinzes hommes,...
Det_Det1 [                                                          (2)
     (1:{Art;APoss;ADem}(0, 2:{AIndef;AOrd}),
         3:{AIndef;AOrd;ACard})
     //
=>   (1 (2, 3))
]

-- un homme, deux jolies femmes, quel beau petit homme !,...
Det_Nom [                                                           (3)
     (1:{Det}, $A:{adjq}, ()2:{Nom})
     //
=>   ((1, $A) 2)
]

-- voiture jaune, de couleur jaune, enfant remuant, livres
-- dispersés,...
Nom_Qual [                                                          (4)
     (1:{Nom} ?{Nom}, 2:{Qual})
     //
=>   (1(2))
]

-- voiture de course, difficultés à faire,...
Nom_Prep [                                                          (5)
     (1:{Nom} ?{Verbe}, 2:{Prep}(3:{Nom;Infi}))
     //
=>   (1(2(3)))
]

-- pierre qui roule,
Nom_CoSubN_Sujet [                                                  (6)
     (1:{Nom} ?{Verbe}, 2:{CoSubN}(()3))
     /Unif(3.syn.sujet, 1)/
=>   (1(2(3))) ;
     Affect(3.syn.sujet, Unif(1, 3.syn.sujet)) ;
     Affect(1, 3.syn.sujet)
]

-- la voiture que j'achèterai,...
Nom_CoSubN_Objet [                                                  (7)
     (1:{Nom} ?{Verbe}, 2:{CoSubN}((3:{Nom ; PPer}, $F)4))
     /Unif(4.syn.objet, 1)/
=>   (1(2((3, $F)4))) ;
     Affect(4.syn.objet, Unif(1, 4.syn.objet)) ;
```

```
      Affect(1, 4.syn.objet)
]

----------------------
-- Dépendants de l'Adj
----------------------
-- très clair, peu cher,...
AdvA_Adj [                                                    (8)
      (1:{AdvA}, 2:{Adj})
      //
=>    ((1) 2)
]

-- capable de lire, bon pour le service
Qual_Prep [                                                   (9)
      (1:{Qual} ?{Nom ; Verbe}, 2:{Prep}(3:{Nom;Infi}))
      //
=>    (1(2(3)))
]

----------------------
-- Dépendants de l'Adv
----------------------
-- beaucoup trop, trop peu,...
AdvA_Adv [                                                    (10)
      (1:{AdvA}, 2:{Adv})
      //
=>    ((1) 2)
]

----------------------
-- Dépendants du Verbe
----------------------
-- il vient, il n'est venu personne,...
sujet [                                                       (11)
      (1:{Nom ; PPer},
       (0, $N:{Ne}, $F:{PPer ; Etre ; Avoir}) 2:{Verb})
      /Unif(1.fonc, suj), Unif(2.syn.sujet, 1)/
=>    ((1, $N, $F) 2) ;
      Affect(2.syn.sujet, Unif(1, 2.syn.sujet)) ;
      Affect(1, 2.syn.sujet)
]

-- ne marche pas, ne vient jamais, ne lui le donne pas,...
Ne_Verbe_Adv [                                                (12)
      (1:{Ne}, (0, $F:{PPer})2:{Verbe}, 3:{AdvV})
      /Unif(3.sem, NEGATION)/
=>    ((1) 2 (3)) ;
      Affect(2.modalite, NEGATION)
]

-- ne vient que si tu es d'accord, n'aime que la lecture
Ne_Verbe_CoSubV [ -- il semble nécessaire de distinguer      (13)
                  -- "que" des autres conjonctions de
                  -- subordinations verbales
```

```
      (1:{Ne}, (0, $F:{PPer})2:{Verbe}, 3:{CoSubV}())
      //
=>    ((1) 2 (3())) ;
      Affect(2.modalite, NEGATION)
]

-- ferma violemment, est rarement reçu, ...
Verbe_AdvV [                                                    (14)
      (1:{Verbe}() ?{Verbe}, 2:{AdvV})
      /Non(Unif(2.sem, NEGATION))/
=>    (1 (2))
]

-- mange la soupe, aperçoit une voiture, ...
objet_direct [                                                  (15)
      (1:{VAction; VPerc}(0, $F:{Adv ; Prep}) ?{Verb}, 2:{Nom})
      /1.syn.objet, Unif(1.syn.objet, 2)/
=>    (1 ($F, 2)) ;
      Affect(1.syn.objet, Unif(1.syn.objet, 2)) ;
      Affect(2, 1.syn.objet)
]

-- je le donne à Pierre,...
objet_direct_avant [                                            (16)
      (1:{PPer}, ()2:{VAction; VPerc})
      /2.syn.objet, Unif(1.fonc, cod), Unif(1, 2.syn.objet)/
=>    ((1) 2) ;
      Affect(2.syn.objet, Unif(1, 2.syn.objet)) ;
      Affect(1, 2.syn.objet)
]

-- je lui donne une voiture,...
objet_indirect_avant [                                          (17)
      (1:{PPer}, ($F:{PPer})2:{VAction; VPerc})
      /2.syn.objetind, Unif(1.fonc, coi),
       Unif(1, 2.syn.objetind)/
=>    ((1, $F) 2) ;
      Affect(2.syn.objetind, Unif(1, 2.syn.objetind)) ;
      Affect(1, 2.syn.objetind)
]

-- regarde passer, doit faire,...
Verb_Infi [                                                     (18)
      (1:{VModal; VPerc} ?{Verbe}, 2:{Infi})
      //
=>    (1(2))
]

-- vient à Paris, mange avec appétit, tente de faire,...
Verbe_Prep [                                                    (19)
      (1:{Verbe} ?{Verbe}, 2:{Prep}(3))
      //
=>    (1(2(3)))
]
```

```
-- je viendrais si je peux, je mange quand j'ai faim, je sais
-- que tu peux,...
Verbe_CoSub [                                                    (20)
     (1:{Verbe} ?{Verbe}, 2:{CoSub}(3))
     //
=>   (1(2(3)))
]

-- est venu, a mangé,...
Aux_PPass [  -- Formation des temps composés                     (21)
     (($G)1:{Etre ; Avoir}($D) ?{Etre ; Avoir}, 2:{PPass})
     /Unif(2.aux, 1.cat)/
=>   (($G, 1, $D) 2) ;
     Affect(2.cat, 2.catverb)
]

-- vais faire, vais courir,...
Aller_Infi [                                                     (22)
     (($G)1:{Aller}($D) ?{CoSub}, 2:{Infi})
     //
=>   (($G, 1, $D) 2) ;
     Affect(2.cat, 2.catverb)
]

-- Etre_PPass [  -- Formation du passif
--   (1:{Etre} ?{Etre}, 2:{PPass})
--   /Unif(2.aux, Avoir)/
-- =>     ((1) 2) ;
--   Affect(2.cat, 2.catverb) ;
--   Affect(2.voix, passive)
-- ]

VEtat_Qual [                                                     (23)
     (1:{VEtat} ?{VEtat}, 2:{Qual})
     //
=>   (1 (2))
]

------------------------------
-- Dépendants de la préposition
------------------------------
-- de la voisine, de bon augure,...
Prep_Nom [                                                       (24)
     (1:{Prep}(), 2:{Nom})
     //
=>   (1 (2))
]

-- de faire, pour rire,...
Prep_Infi [                                                      (25)
     (1:{Prep}(), 2:{Infi})
     //
=>   (1 (2))
]
```

```
-- de lui, à elle
Prep_Pron [                                                          (26)
      (1:{Prep}(), 2:{Pron})
      /Unif(2.fonc, coi)/
=>    (1 (2))
]

------------------------------
-- Dépendants de la Conjonction
------------------------------

-- Subordination

-- qui roule, dont je parle,...
CoSubN_Verb [                                                        (27)
      (1:{CoSubN}(), 2:{Verb})
      //
=>    (1 (2))
]

-- que faire, je crois qu'arrachée et replantée elle
-- repousserait mieux
CoSubV_Verbe [                                                       (28)
      (1:{CoSubV}(), 2:{Verbe})
      //
=>    (1 (2))
]

-- Coordination

-- le chien et le chat, la belle voiture et le bus rouge, Pierre
-- et elle,...
CoCoS_Nom [                                                          (29)
      (1:{Nom ; Pron} ?{Nom}, ()2:{CoCoS}(), 3:{Nom ; Pron})
      /Unif(1.sem, 3.sem)/
=>    ((1) 2 (3)) ;
      AFFECT(2.cat, Nom) ;
      Affect(2.sem, Unif(1.sem, 3.sem))
]

-- nourris et logés, mangent et boivent, suant et soufflant,...
CoCoS_Verbe [                                                        (30)
      (1:{Verbe} ?{Verbe}, ()2:{CoCoS}(), 3:{Verbe})
      //
=>    ((1) 2 (3)) ;
      AFFECT(2.cat, Verbe)
]

-- de bric et de broc, de faire et de défaire,...
CoCoS_Prep [                                                         (31)
      (1:{Prep}(2) ?{Prep}, ()3:{CoCoS}(), 4:{Prep}(5))
      //
=>    ((1(2)) 3 (4(5))) ;
      Affect(3.cat, Prep)
]
```

```
-- galamment mais fermement,...
CoCoS_Adv [                                                          (32)
     (1:{Adv} ?{Adv}, ()2:{CoCoS}(), 3:{Adv})
     //
=>   ((1) 2 (3)) ;
     Affect(2.cat, Adv)
]

-- trois ou quatre,...
CoCoS_Adj [                                                          (33)
     (1:{Adj} ?{Adj}, ()2:{CoCoS}(), 3:{Adj})
     /Unif(1.cat, 3.cat)/
=>   ((1) 2 (3)) ;
     Affect(2.cat, Unif(1.cat, 3.cat))
]

-- soit la voiture soit le vélo, ni elle ni Pierre
CoCoD_Nom [                                                          (34)
     (1:{CoCoD}(), 2:{Nom ; Pron}, 3:{CoCoD}(), 4:{Nom ; Pron})
     /Unif(2.sem, 4.sem)/
=>   ((1(2)) 3 (4)) ;
     Affect(2.cat, Nom) ;
     Affect(2.sem, Unif(2.sem, 4.sem))
]

-- ni trois ni quatre, ni par la route ni par le train, soit un
-- soit plusieurs,...
CoCoD_Autre [                                                        (35)
     (1:{CoCoD}(), 2, 3:{CoCoD}(), 4)
     /Unif(2.cat, 4.cat)/
=>   ((1(2)) 3 (4)) ;
     Affect(2.cat, Unif(2.cat, 4.cat))
]

-----------------------------
-- Règles de désambiguïsation
-----------------------------
PPer_Nom [ (1:{PPer}, $F, 2:{Nom}) // => () ]                        (36)
Det_Verbe [ (1:{Det}, ()2:{Verbe}) // => () ]                        (37)
Nom_CoSubV [ (1:{Nom}, 2:{CoSubV}(3)) // => ()]                      (38)
```

B.1.3. Table d'accès aux règles

```
Ne          AdvA_Adv, CoCoS_Adv, CoCoD_Autre.
Adj         Det_Det0, Det_Det1, Nom_Qual, AdvA_Adj, Aux_PPass,
            VEtat_Qual, CoSubV_Verbe, CoCoS_Verbe, CoCoS_Adj,
            CoCoD_Autre, Det_Verbe.
Adv         AdvA_Adv, Ne_Verbe_Adv, Verbe_AdvV, CoCoS_Adv,
            CoCoD_Autre.
Det         Det_Det0, Det_Det1, AdvA_Adj, CoCoS_Adj,
            CoCoD_Autre.
CLS         Det_Det0, Det_Det1, Det_Nom, Nom_Qual, Nom_Prep,
            Nom_CoSubN_Sujet, Nom_CoSubN_Objet, AdvA_Adj,
            Qual_Prep, AdvA_Adv, sujet, Ne_Verbe_Adv,
```

	Ne_Verbe_CoSubV, Verbe_AdvV, objet_direct, objet_direct_avant, objet_indirect_avant, Verb_Infi, Verbe_Prep, Verbe_CoSub, Aux_PPass, Aller_Infi, VEtat_Qual, Prep_Nom, Prep_Infi, Prep_Pron, CoSubN_Verb, CoSubV_Verbe, CoCoS_Nom, CoCoS_Verbe, CoCoS_Prep, CoCoS_Adv, CoCoS_Adj, CoCoD_Nom, CoCoD_Autre, PPer_Nom, Det_Verbe, Nom_CoSubV.
Art	CoCoD_Autre.
Nom	Det_Nom, objet_direct, Prep_Nom, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre, PPer_Nom.
ADem	AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
AdvA	AdvA_Adv, CoCoS_Adv, CoCoD_Autre.
adjq	Nom_Qual, AdvA_Adj, VEtat_Qual, CoCoS_Adj, CoCoD_Autre.
AOrd	Det_Det0, Det_Det1, AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
Conj	Nom_CoSubN_Sujet, Nom_CoSubN_Objet, Ne_Verbe_CoSubV, Verbe_CoSub, CoCoD_Autre, Nom_CoSubV.
subc	Det_Nom, objet_direct, Prep_Nom, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre, PPer_Nom.
Verb	sujet, objet_direct_avant, objet_indirect_avant, CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
Etre	CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
AdvV	AdvA_Adv, Ne_Verbe_Adv, Verbe_AdvV, CoCoS_Adv, CoCoD_Autre.
Qual	Nom_Qual, AdvA_Adj, Aux_PPass, VEtat_Qual, CoSubV_Verbe, CoCoS_Verbe, CoCoS_Adj, CoCoD_Autre, Det_Verbe.
Prep	Nom_Prep, Qual_Prep, Verbe_Prep, CoCoS_Prep, CoCoD_Autre.
subp	Det_Nom, objet_direct, Prep_Nom, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre, PPer_Nom.
PInt	Prep_Pron, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre.
NonQ	Det_Det0, Det_Det1, AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
Pron	Prep_Pron, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre.
ACard	Det_Det0, Det_Det1, AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
Indef	CoCoD_Autre.
CoCoD	CoCoD_Autre.
Aller	CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
Verbe	Nom_Qual, AdvA_Adj, sujet, objet_direct_avant, objet_indirect_avant, Verb_Infi, Aux_PPass, Aller_Infi, VEtat_Qual, Prep_Infi, CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe, CoCoS_Adj, CoCoD_Autre, Det_Verbe.
CoCoS	CoCoD_Autre.
CoSub	Nom_CoSubN_Sujet, Nom_CoSubN_Objet, Ne_Verbe_CoSubV, Verbe_CoSub, CoCoD_Autre, Nom_CoSubV.
VPerc	sujet, objet_direct_avant, objet_indirect_avant, CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
Avoir	CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
VEtat	sujet, CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre, Det_Verbe.

```
APoss       AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
PPass       Nom_Qual, AdvA_Adj, Aux_PPass, VEtat_Qual,
            CoSubV_Verbe, CoCoS_Verbe, CoCoS_Adj, CoCoD_Autre,
            Det_Verbe.
PPres       Nom_Qual, AdvA_Adj, VEtat_Qual, CoSubV_Verbe,
            CoCoS_Verbe, CoCoS_Adj, CoCoD_Autre, Det_Verbe.
Def         CoCoD_Autre.
PPoss       Prep_Pron, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre.
AIndef      Det_Det0, Det_Det1, AdvA_Adj, CoCoS_Adj,
            CoCoD_Autre.
VModal      sujet, CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe,
            CoCoD_Autre, Det_Verbe.
AIntEx      AdvA_Adj, CoCoS_Adj, CoCoD_Autre.
CoSubN      Nom_CoSubN_Sujet, Nom_CoSubN_Objet, Verbe_CoSub,
            CoCoD_Autre.
CoSubV      Ne_Verbe_CoSubV, Verbe_CoSub, CoCoD_Autre,
            Nom_CoSubV.
PDem        Prep_Pron, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre.
Part        CoCoD_Autre.
VAction     sujet, objet_direct_avant, objet_indirect_avant,
            CoSubN_Verb, CoSubV_Verbe, CoCoS_Verbe, CoCoD_Autre,
            Det_Verbe.
Infi        Verb_Infi, Aller_Infi, Prep_Infi, CoSubV_Verbe,
            CoCoS_Verbe, CoCoD_Autre, Det_Verbe.
PPer        Prep_Pron, CoCoS_Nom, CoCoD_Nom, CoCoD_Autre.
```

B.2. Analyse de $a^n b^n c^n$, $n \geq 1$

B.2.1. Signature

```
-- Signature pour l'analyse de anbncn
{ a, b, c } < CLS
```

B.2.2. Grammaire

```
-- Grammaire d'analyse de anbncn

-- On ne fait rien tant qu'on n'a pas un c.
-- Le critère de succès de l'analyse est que la pile ne
-- contienne qu'un arbre.

-- Construction d'une forêt de (((a)b)c))
abc [ ($A:{a}, 1:{a}, $B:{b}, 2:{b}, 3:{c}) //
=>
(((1)2)3, $A, $B)
]

-- Réduction de la forêt de (((a)b)c)) en un seul arbre :
arbre [ (((1:{a})2:{b})3:{c}, ((4:{a})5:{b})6:{c}) //
=>
( ((4)5) 6 ( ((1)2) 3) )
]
```

B.2.3. Exemple d'analyse : aabbcc

```
-- Test de la grammaire anbncn (DONNEES)
-- Phrase a2b2c2
-- Signature donnees\anbncn
_TOUT_(CAT => a)
_TOUT_(CAT => a)
_TOUT_(CAT => b)
_TOUT_(CAT => b)
_TOUT_(CAT => c)
_TOUT_(CAT => c)
```

On donne ci-dessous la sortie du transducteur en mode trace pour l'analyse de $a^2b^2c^2$, mais seulement à partir de l'introduction du premier c.

```
--------- Analyse Syntaxico-Sémantique --------
[......]
Avant Analyse :
+- 1    _TOUT_(cat => a )
+- 2    _TOUT_(cat => a )
+- 3    _TOUT_(cat => b )
+- 4    _TOUT_(cat => b )
+- 5    _TOUT_(cat => c )


Application :
+- 1    _TOUT_(cat => a )
+- 2    _TOUT_(cat => a )
+- 3    _TOUT_(cat => b )
+- 4    _TOUT_(cat => b )
+- 5    _TOUT_(cat => c )

Règle : 1
abc=>
(
           +- 2    _TOUT_(cat => {a} )
      +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
+- 1    _TOUT_(cat => {a} )
+- 3    _TOUT_(cat => {b} )

)
Application :
           +- 2    _TOUT_(cat => {a} )
      +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
+- 1    _TOUT_(cat => {a} )
+- 3    _TOUT_(cat => {b} )

Après Analyse :
           +- 2    _TOUT_(cat => {a} )
      +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
+- 1    _TOUT_(cat => {a} )
+- 3    _TOUT_(cat => {b} )
```

```
Avant Analyse :
          +- 2    _TOUT_(cat => {a} )
     +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
+- 1    _TOUT_(cat => {a} )
+- 3    _TOUT_(cat => {b} )
+- 6    _TOUT_(cat => c )


Application :
          +- 2    _TOUT_(cat => {a} )
     +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
+- 1    _TOUT_(cat => {a} )
+- 3    _TOUT_(cat => {b} )
+- 6    _TOUT_(cat => c )

Règle : 1
abc=>
(
          +- 2    _TOUT_(cat => {a} )
     +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
|         +- 1    _TOUT_(cat => {a} )
|    +- 3-+ _TOUT_(cat => {b} )
+- 6-+ _TOUT_(cat => {c} )

)
Application :
          +- 2    _TOUT_(cat => {a} )
     +- 4-+ _TOUT_(cat => {b} )
+- 5-+ _TOUT_(cat => {c} )
|         +- 1    _TOUT_(cat => {a} )
|    +- 3-+ _TOUT_(cat => {b} )
+- 6-+ _TOUT_(cat => {c} )

Règle : 2
arbre=>
(
          +- 1    _TOUT_(cat => {a} )
     +- 3-+ _TOUT_(cat => {b} )
-- 6-+ _TOUT_(cat => {c} )
     |         +- 2    _TOUT_(cat => {a} )
     |    +- 4-+ _TOUT_(cat => {b} )
     +- 5-+ _TOUT_(cat => {c} )

)
Application :
          +- 1    _TOUT_(cat => {a} )
     +- 3-+ _TOUT_(cat => {b} )
-- 6-+ _TOUT_(cat => {c} )
     |         +- 2    _TOUT_(cat => {a} )
     |    +- 4-+ _TOUT_(cat => {b} )
     +- 5-+ _TOUT_(cat => {c} )
```

```
Après Analyse :
          +- 1    _TOUT_(cat => {a} )
     +- 3-+ _TOUT_(cat => {b} )
-- 6-+ _TOUT_(cat => {c} )
     |          +- 2    _TOUT_(cat => {a} )
     |     +- 4-+ _TOUT_(cat => {b} )
     +- 5-+ _TOUT_(cat => {c} )

-------------------------------------
Résultat :
          +- 1    _TOUT_(cat => {a} )
     +- 3-+ _TOUT_(cat => {b} )
-- 6-+ _TOUT_(cat => {c} )
     |          +- 2    _TOUT_(cat => {a} )
     |     +- 4-+ _TOUT_(cat => {b} )
     +- 5-+ _TOUT_(cat => {c} )
```

Annexe C

Trace d'analyse

On donne ci-dessous la trace du transducteur présenté au chapitre 4 pour la phrase : "les trois ou quatre personnes que je connais à Paris travaillent à l'école de commerce". La signature et la grammaire utilisées sont données dans l'annexe B. A cette phrase on associe la liste de Ψ-termes :

```
{ UL(cat => Def) ;          -- les
  UL(cat => PPer ; fonc => cod) }
UL(cat => ACard)            -- trois
UL(cat => CoCoS)            -- ou
UL(cat => ACard)            -- quatre
UL(cat => subc ;            -- personnes
   sem => HUMAIN)
UL(cat => CoSub)            -- que
UL(cat => PPer ;            -- je
   fonc => suj ;
   sem => ANIME)
UL(cat => VAction ;         -- connais
   syn => PP(sujet => GN(sem => @S:ANIME) ;
             objet => GN(sem => @O:{ANIME ; OBJET})) ;
   sem => SAVOIR(détenteur => @S ; objet => @O)
  )
UL(cat => Prep)             -- à
UL(cat => subp)             -- Paris
UL(cat => VAction ;         -- travaillent
   syn => PP(sujet => GN(sem => @S:ANIME)) ;
   sem => ACTION(agent => @S)
  )
UL(cat => Prep)             -- à
{ UL(cat => Def) ;          -- l'
  UL(cat => PPer ; fonc => cod) }
UL(cat => subc ;            -- école
   sem => LIEU)
UL(cat => Prep)             -- de
UL(cat => subc)             -- commerce
```

En mode trace, le transducteur affiche la forêt à analyser avant analyse et après analyse. De plus avant chaque application des règles de la grammaire à un arbre, il affiche l'arbre concerné (Application). Pour simplifier la trace, nous avons éliminé l'affichage de la forêt après analyse. Nous avons, ajouté quelques commentaires *en italique*.

```
------- Analyse Syntaxico-Sémantique -------
Avant Analyse :
```
On a introduit le premier mot "les", qui est ambigu, on a donc 1 arbre pour chaque interprétation :
```
—— 1    UL(cat => Def )

—— 1    UL(cat => PPer ;fonc => cod )


Application :
```
au premier arbre, aucune règle ne s'applique à un arbre seul
```
—— 1    UL(cat => Def )

Application :
```
au deuxième arbre
```
—— 1    UL(cat => PPer ;fonc => cod )

Avant Analyse :
```
on introduit le deuxième mot "trois", on a toujours deux arbres
```
+— 1    UL(cat => Def )
+— 2    UL(cat => ACard )

+— 1    UL(cat => PPer ;fonc => cod )
+— 2    UL(cat => ACard )


Application :
+— 1    UL(cat => Def )
+— 2    UL(cat => ACard )

Règle : 1
```
Elle remplace l'arbre ci-dessus par l'arbre donné à sa suite
```
Det_Det0=>
(
—— 1—+ UL(cat => {Def} )
       +— 2    UL(cat => {ACard} )

)
Application :
+— 1    UL(cat => PPer ;fonc => cod )
+— 2    UL(cat => ACard )
```
Aucune règle ne s'applique

```
Application :
```
On essaye d'appliquer la grammaire à l'arbre produit par `Det_Det0`
```
—— 1—+ UL(cat => {Def} )
       +— 2    UL(cat => {ACard} )
```
Aucune règle ne s'applique

```
Avant Analyse :
```
On introduit le troisième mot "ou"
```
+— 1    UL(cat => PPer ;fonc => cod )
+— 2    UL(cat => ACard )
+— 3    UL(cat => CoCoS )

+— 1—+ UL(cat => {Def} )
|      +— 2    UL(cat => {ACard} )
+— 3    UL(cat => CoCoS )


Application :
```

```
+- 1   UL(cat => PPer ;fonc => cod )
+- 2   UL(cat => ACard )
+- 3   UL(cat => CoCoS )

Application :
+- 1-+ UL(cat => {Def} )
|    +- 2   UL(cat => {ACard} )
+- 3   UL(cat => CoCoS )


Avant Analyse : on introduit "quatre"
+- 1   UL(cat => PPer ;fonc => cod )
+- 2   UL(cat => ACard )
+- 3   UL(cat => CoCoS )
+- 4   UL(cat => ACard )

+- 1-+ UL(cat => {Def} )
|    +- 2   UL(cat => {ACard} )
+- 3   UL(cat => CoCoS )
+- 4   UL(cat => ACard )


Application :
+- 1   UL(cat => PPer ;fonc => cod )
+- 2   UL(cat => ACard )
+- 3   UL(cat => CoCoS )
+- 4   UL(cat => ACard )

Règle : 33
CoCoS_Adj=>
(
+- 1   UL(cat => PPer ;fonc => cod )
|    +- 2   UL(cat => {ACard} )
+- 3-+ UL(cat => {ACard} )
     +- 4   UL(cat => {ACard} )

)
Application :
+- 1-+ UL(cat => {Def} )
|    +- 2   UL(cat => {ACard} )
+- 3   UL(cat => CoCoS )
+- 4   UL(cat => ACard )

Règle : 33
CoCoS_Adj=>
(
-- 1-+ UL(cat => {Def} )
     |    +- 2   UL(cat => {ACard} )
     +- 3-+ UL(cat => {ACard} )
          +- 4   UL(cat => {ACard} )

)
Application :
+- 1   UL(cat => PPer ;fonc => cod )
|    +- 2   UL(cat => {ACard} )
```

```
+- 3-+ UL(cat => {ACard} )
     +- 4   UL(cat => {ACard} )

Application :
-- 1-+ UL(cat => {Def} )
     |    +- 2   UL(cat => {ACard} )
     +- 3-+ UL(cat => {ACard} )
          +- 4   UL(cat => {ACard} )


Avant Analyse : on introduit "personnes"
+- 1   UL(cat => PPer ;fonc => cod )
|    +- 2   UL(cat => {ACard} )
+- 3-+ UL(cat => {ACard} )
|    +- 4   UL(cat => {ACard} )
+- 5   UL(cat => subc ;sem => HUMAIN )

+- 1-+ UL(cat => {Def} )
|    |    +- 2   UL(cat => {ACard} )
|    +- 3-+ UL(cat => {ACard} )
|         +- 4   UL(cat => {ACard} )
+- 5   UL(cat => subc ;sem => HUMAIN )


Application :
+- 1   UL(cat => PPer ;fonc => cod )
|    +- 2   UL(cat => {ACard} )
+- 3-+ UL(cat => {ACard} )
|    +- 4   UL(cat => {ACard} )
+- 5   UL(cat => subc ;sem => HUMAIN )
On a ici un exemple où deux règles s'appliquent à la même forêt, la règle
Det_Nom produit une nouvelle forêt, tandis que la règle de désambiguïsation
PPer_Nom élimine la forêt initiale.

Règle : 3
Det_Nom=>
(
+- 1   UL(cat => PPer ;fonc => cod )
|         +- 2   UL(cat => {ACard} )
|    +- 3-+ UL(cat => {ACard} )
|    |    +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )

)
Règle : 36
PPer_Nom=>
(
)
Application :
+- 1-+ UL(cat => {Def} )
|    |    +- 2   UL(cat => {ACard} )
|    +- 3-+ UL(cat => {ACard} )
|         +- 4   UL(cat => {ACard} )
+- 5   UL(cat => subc ;sem => HUMAIN )

Règle : 3
```

```
Det_Nom=>
(
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
-- 5-+ UL(sem => HUMAIN ;cat => {subc} )

)
Application :
+- 1   UL(cat => PPer ;fonc => cod )
|        +- 2   UL(cat => {ACard} )
|    +- 3-+ UL(cat => {ACard} )
|    |    +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )

Règle : 36
PPer_Nom=>
(
)
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
-- 5-+ UL(sem => HUMAIN ;cat => {subc} )


Avant Analyse : On n'a plus qu'une seule forêt , on introduit "que"
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )


Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
```

Aucune règle ne s'applique

```
Avant Analyse : on introduit "je"
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
+- 7   UL(cat => PPer ;fonc => suj ;sem => ANIME )
```

```
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
+- 7   UL(cat => PPer ;fonc => suj ;sem => ANIME )
```

```
Avant Analyse : on introduit "connais"
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
+- 7   UL(cat => PPer ;fonc => suj ;sem => ANIME )
+- 8   UL(cat => VAction ;syn => PP(sujet => GN(sem => @36:ANIME
) ;objet => GN(sem => @38:{ANIME ;OBJET} ) ) ;sem =>
SAVOIR(détenteur => @36 ;objet => @38 ) )
```

```
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
+- 7   UL(cat => PPer ;fonc => suj ;sem => ANIME )
+- 8   UL(cat => VAction ;syn => PP(sujet => GN(sem => @36:ANIME
) ;objet => GN(sem => @38:{ANIME ;OBJET} ) ) ;sem =>
SAVOIR(détenteur => @36 ;objet => @38 ) )
```

Règle : 11 *Rattache le sujet "je" à "connais"*

```
sujet=>
(
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
|    +- 7   GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
+- 8-+ UL(sem => SAVOIR(objet => @137 ;détenteur => @135 ) ;syn
=> PP(objet => GN(sem => @137:{OBJET ;ANIME} ) ;sujet => GN(fonc
=> suj ;cat => {PPer} ;sem => @135:ANIME ) ) ;cat => {VAction} )

)
```

Les règles 16 et 17 ont des schémas qui s'instancient avec la forêt donnée mais les conditions de la partie droite de la règle ne sont pas vérifiées (fonction syntaxique du pronom personnel).

Règle : 16

```
Règle : 17
Application : au résultat de la règle sujet ci-dessus
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6   UL(cat => CoSub )
|    +- 7   GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
+- 8-+ UL(sem => SAVOIR(objet => @137 ;détenteur => @135 ) ;syn
=> PP(objet => GN(sem => @137:{OBJET ;ANIME} ) ;sujet => GN(fonc
=> suj ;cat => {PPer} ;sem => @135:ANIME ) ) ;cat => {VAction} )
```

Encore un cas où deux règles s'appliquent à la même forêt, mais cette fois chaque règle crée une nouvelle forêt :

```
Règle : 27
CoSubN_Verb=> "que" est interprété comme un pronom relatif
(
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6-+ UL(cat => {CoSubN} )
     |    +- 7   GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
     +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem =>
@175 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @172 ) )
;sem => SAVOIR(détenteur => @175:ANIME ;objet => @172:{ANIME
;OBJET} ) )

)
Règle : 28
CoSubV_Verbe=> "que" est interprété comme une conjonction de subordination
(
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6-+ UL(cat => {CoSubV} )
     |    +- 7   GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
     +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem =>
@194 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @191 ) )
;sem => SAVOIR(détenteur => @194:ANIME ;objet => @191:{ANIME
;OBJET} ) )

)
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2   UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4   UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6-+ UL(cat => {CoSubN} )
     |    +- 7   GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
```

```
      +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem =>
@175 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @172 ) )
;sem => SAVOIR(détenteur => @175:ANIME ;objet => @172:{ANIME
;OBJET} ) )

Règle : 7
Nom_CoSubN_Objet=> Rattachement d'une relative dont l'antécédant est objet du verbe
(
      +- 1-+ UL(cat => {Def} )
      |     |     +- 2    UL(cat => {ACard} )
      |     +- 3-+ UL(cat => {ACard} )
      |           +- 4    UL(cat => {ACard} )
--- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
      +- 6-+ UL(cat => {CoSubN} )
            |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
            +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )

)
Application :
      +- 1-+ UL(cat => {Def} )
      |     |     +- 2    UL(cat => {ACard} )
      |     +- 3-+ UL(cat => {ACard} )
      |           +- 4    UL(cat => {ACard} )
+- 5-+ UL(sem => HUMAIN ;cat => {subc} )
+- 6-+ UL(cat => {CoSubV} )
      |     +- 7    GN(sem => ANIME ;fonc => suj ;cat => {PPer} )
      +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem =>
@194 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @191 ) )
;sem => SAVOIR(détenteur => @194:ANIME ;objet => @191:{ANIME
;OBJET} ) )

Règle : 38
Nom_CoSubV=> Elimination de la mauvaise interprétation du "que"
(
)
Application :
      +- 1-+ UL(cat => {Def} )
      |     |     +- 2    UL(cat => {ACard} )
      |     +- 3-+ UL(cat => {ACard} )
      |           +- 4    UL(cat => {ACard} )
--- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
      +- 6-+ UL(cat => {CoSubN} )
            |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
            +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )


Avant Analyse : Introduction de "à"
```

```
      +- 1-+ UL(cat => {Def} )
      |    |    +- 2   UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |         +- 4   UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|         +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9   UL(cat => Prep )
```

```
Application :
      +- 1-+ UL(cat => {Def} )
      |    |    +- 2   UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |         +- 4   UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|         +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9   UL(cat => Prep )
```

Avant Analyse : *Introduction de "Paris"*

```
      +- 1-+ UL(cat => {Def} )
      |    |    +- 2   UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |         +- 4   UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|         +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9   UL(cat => Prep )
+-10   UL(cat => subp )
```

```
Application :
      +- 1-+ UL(cat => {Def} )
      |    |    +- 2   UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |         +- 4   UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
```

```
|          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|          +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9    UL(cat => Prep )
+-10    UL(cat => subp )

Règle : 24
Prep_Nom=>
(
      +- 1-+ UL(cat => {Def} )
      |    |     +- 2    UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |          +- 4    UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|          +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9-+ UL(cat => {Prep} )
     +-10    UL(cat => {subp} )

)
Application :
      +- 1-+ UL(cat => {Def} )
      |    |     +- 2    UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |          +- 4    UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|          +- 8-+ UL(sem => SAVOIR(objet => @227 ;détenteur =>
@222 ) ;syn => PP(objet => GN(cat => {subc} ;sem =>
@227:{HUMAIN} ) ;sujet => GN(fonc => suj ;cat => {PPer} ;sem =>
@222:ANIME ) ) ;cat => {VAction} )
+- 9-+ UL(cat => {Prep} )
     +-10    UL(cat => {subp} )

Règle : 19
Rattachement du groupe prépositionnel "à Paris" au verbe "connais", la limite dans la règle
Nom_Prep interdit le rattachement à "personnes"
Verbe_Prep=>
(
      +- 1-+ UL(cat => {Def} )
      |    |     +- 2    UL(cat => {ACard} )
      |    +- 3-+ UL(cat => {ACard} )
      |          +- 4    UL(cat => {ACard} )
—— 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
     +- 6-+ UL(cat => {CoSubN} )
```

```
          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem
=> @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @265
;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME ;objet
=> @265:{HUMAIN} ) )
               +- 9-+ UL(cat => {Prep} )
                    +-10    UL(cat => {subp} )

)
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2    UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4    UL(cat => {ACard} )
-- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
     +- 6-+ UL(cat => {CoSubN} )
          |    +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem
=> @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @265
;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME ;objet
=> @265:{HUMAIN} ) )
               +- 9-+ UL(cat => {Prep} )
                    +-10    UL(cat => {subp} )
```

Avant Analyse : *Introduction de "travaillent"*

```
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2    UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4    UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|         |    +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
|         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem
=> @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @265
;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME ;objet
=> @265:{HUMAIN} ) )
|              +- 9-+ UL(cat => {Prep} )
|                   +-10    UL(cat => {subp} )
+-11   UL(cat => VAction ;syn => PP(sujet => GN(sem => @55:ANIME
) ) ;sem => ACTION(agent => @55 ) )
```

```
Application :
     +- 1-+ UL(cat => {Def} )
     |    |    +- 2    UL(cat => {ACard} )
     |    +- 3-+ UL(cat => {ACard} )
     |         +- 4    UL(cat => {ACard} )
+- 5-+ GN(cat => {subc} ;sem => {HUMAIN} )
|    +- 6-+ UL(cat => {CoSubN} )
|         |    +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
```

```
|          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem
=> @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem => @265
;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME ;objet
=> @265:{HUMAIN} ) )
|               +- 9-+ UL(cat => {Prep} )
|                    +-10   UL(cat => {subp} )
+-11   UL(cat => VAction ;syn => PP(sujet => GN(sem => @55:ANIME
) ) ;sem => ACTION(agent => @55 ) )

Règle : 11
sujet=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )


Avant Analyse : Introduction de "à"
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
```

```
     |     +- 6-+ UL(cat => {CoSubN} )
     |          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |                +- 9-+ UL(cat => {Prep} )
     |                     +-10    UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12    UL(cat => Prep )
```

```
Application :
          +- 1-+ UL(cat => {Def} )
          |    |     +- 2    UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4    UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |                +- 9-+ UL(cat => {Prep} )
     |                     +-10    UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12    UL(cat => Prep )
```

Avant Analyse : *Introduction de "l'", ambigü et donc donnant deux forêts :*

```
          +- 1-+ UL(cat => {Def} )
          |    |     +- 2    UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4    UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |                +- 9-+ UL(cat => {Prep} )
     |                     +-10    UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12    UL(cat => Prep )
+-13    UL(cat => Def )

          +- 1-+ UL(cat => {Def} )
```

```
          |     |     +- 2   UL(cat => {ACard} )
          |     +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |     +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
+-13   UL(cat => PPer ;fonc => cod )


Application :
          +- 1-+ UL(cat => {Def} )
          |     |     +- 2   UL(cat => {ACard} )
          |     +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |     +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
+-13   UL(cat => Def )

Application :
          +- 1-+ UL(cat => {Def} )
          |     |     +- 2   UL(cat => {ACard} )
          |     +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |     +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
```

```
+-12   UL(cat => Prep )
+-13   UL(cat => PPer ;fonc => cod )
```

Règle : 26 *Prep_Pron, ne s'applique pas car le* `PPer` *n'a pas la valeur* `dat` *pour le trait fonc.*

Avant Analyse : *Introduction de "école".*

```
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
+-13   UL(cat => Def )
+-14   UL(cat => subc ;sem => LIEU )

          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
+-13   UL(cat => PPer ;fonc => cod )
+-14   UL(cat => subc ;sem => LIEU )
```

Application :

```
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
```

```
     |          |     +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
     |                    +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
+-13   UL(cat => Def )
+-14   UL(cat => subc ;sem => LIEU )

Règle : 3
Det_Nom=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |          |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
     |                    +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
|    +-13   UL(cat => {Def} )
+-14-+ UL(sem => LIEU ;cat => {subc} )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |          |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
     |                    +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
```

```
+-13   UL(cat => PPer ;fonc => cod )
+-14   UL(cat => subc ;sem => LIEU )

Règle : 36
PPer_Nom=>
(
)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12   UL(cat => Prep )
|    +-13   UL(cat => {Def} )
+-14-+ UL(sem => LIEU ;cat => {subc} )

Règle : 24
Prep_Nom=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12-+ UL(cat => {Prep} )
     |    +-13   UL(cat => {Def} )
     +-14-+ UL(cat => {subc} ;sem => LIEU )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
```

```
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(sem => ACTION(agent => @296 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @296:{HUMAIN} ) ) ;cat => {VAction} )
+-12-+ UL(cat => {Prep} )
     |    +-13   UL(cat => {Def} )
     +-14-+ UL(cat => {subc} ;sem => LIEU )

Règle : 19
Verbe_Prep=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
     +-12-+ UL(cat => {Prep} )
          |    +-13   UL(cat => {Def} )
          +-14-+ UL(sem => LIEU ;cat => {subc} )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
```

```
—11—+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
     +—12—+ UL(cat => {Prep} )
          |    +—13   UL(cat => {Def} )
          +—14—+ UL(sem => LIEU ;cat => {subc} )
```

Avant Analyse : *Introudction de "de"*

```
          +— 1—+ UL(cat => {Def} )
          |    |    +— 2   UL(cat => {ACard} )
          |    +— 3—+ UL(cat => {ACard} )
          |         +— 4   UL(cat => {ACard} )
     +— 5—+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +— 6—+ UL(cat => {CoSubN} )
     |         |    +— 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +— 8—+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +— 9—+ UL(cat => {Prep} )
     |                   +—10   UL(cat => {subp} )
+—11—+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +—12—+ UL(cat => {Prep} )
|         |    +—13   UL(cat => {Def} )
|         +—14—+ UL(sem => LIEU ;cat => {subc} )
+—15   UL(cat => Prep )
```

Application :

```
          +— 1—+ UL(cat => {Def} )
          |    |    +— 2   UL(cat => {ACard} )
          |    +— 3—+ UL(cat => {ACard} )
          |         +— 4   UL(cat => {ACard} )
     +— 5—+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +— 6—+ UL(cat => {CoSubN} )
     |         |    +— 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +— 8—+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +— 9—+ UL(cat => {Prep} )
     |                   +—10   UL(cat => {subp} )
+—11—+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +—12—+ UL(cat => {Prep} )
|         |    +—13   UL(cat => {Def} )
|         +—14—+ UL(sem => LIEU ;cat => {subc} )
+—15   UL(cat => Prep )
```

Avant Analyse : *Introduction de "commerce".*

```
          +— 1—+ UL(cat => {Def} )
```

```
          |     |     +- 2   UL(cat => {ACard} )
          |     +- 3-+ UL(cat => {ACard} )
          |          +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |     +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +-12-+ UL(cat => {Prep} )
|         |    +-13   UL(cat => {Def} )
|         +-14-+ UL(sem => LIEU ;cat => {subc} )
+-15   UL(cat => Prep )
+-16   UL(cat => subc )


Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
+-11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +-12-+ UL(cat => {Prep} )
|         |    +-13   UL(cat => {Def} )
|         +-14-+ UL(sem => LIEU ;cat => {subc} )
+-15   UL(cat => Prep )
+-16   UL(cat => subc )

Règle : 24
Prep_Nom=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
```

```
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
     |                    +-10   UL(cat => {subp} )
+-11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +-12-+ UL(cat => {Prep} )
|         |    +-13   UL(cat => {Def} )
|         +-14-+ UL(sem => LIEU ;cat => {subc} )
+-15-+ UL(cat => {Prep} )
     +-16   UL(cat => {subc} )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
     |                    +-10   UL(cat => {subp} )
+-11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
|    +-12-+ UL(cat => {Prep} )
|         |    +-13   UL(cat => {Def} )
|         +-14-+ UL(sem => LIEU ;cat => {subc} )
+-15-+ UL(cat => {Prep} )
     +-16   UL(cat => {subc} )
```

Cette fois ci, le groupe prépostitionnel peut être attaché au nom "école" ou au verbe "travaillent", on a donc deux règles qui s'appliquent, produisant deux arbres :

```
Règle : 5
Nom_Prep=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |               +- 9-+ UL(cat => {Prep} )
```

```
     |                    +-10   UL(cat => {subp} )
--11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
     +-12-+ UL(cat => {Prep} )
          |    +-13   UL(cat => {Def} )
          +-14-+ UL(cat => {subc} ;sem => LIEU )
               +-15-+ UL(cat => {Prep} )
                    +-16   UL(cat => {subc} )

)
Règle : 19
Verbe_Prep=>
(
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(sem => ACTION(agent => @376 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @376:{HUMAIN} ) ) ;cat => {VAction} )
     +-12-+ UL(cat => {Prep} )
     |    |    +-13   UL(cat => {Def} )
     |    +-14-+ UL(sem => LIEU ;cat => {subc} )
     +-15-+ UL(cat => {Prep} )
          +-16   UL(cat => {subc} )

)
Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
     +-12-+ UL(cat => {Prep} )
          |    +-13   UL(cat => {Def} )
          +-14-+ UL(cat => {subc} ;sem => LIEU )
```

```
                +-15-+ UL(cat => {Prep} )
                     +-16   UL(cat => {subc} )

Application :
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(sem => ACTION(agent => @376 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @376:{HUMAIN} ) ) ;cat => {VAction} )
     +-12-+ UL(cat => {Prep} )
     |    |    +-13   UL(cat => {Def} )
     |    +-14-+ UL(sem => LIEU ;cat => {subc} )
     +-15-+ UL(cat => {Prep} )
          +-16   UL(cat => {subc} )
```

Résultat : *On obtient deux arbres, qui se distinguent par le rattachement du groupe "de commerce" au verbe "travaillent" ou au nom "école".*

```
          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
          |         +- 4   UL(cat => {ACard} )
     +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
     |    +- 6-+ UL(cat => {CoSubN} )
     |         |    +- 7   GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
     |         +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
     |              +- 9-+ UL(cat => {Prep} )
     |                   +-10   UL(cat => {subp} )
--11-+ UL(cat => {VAction} ;syn => PP(sujet => GN(sem => @339
;cat => {subc} ) ) ;sem => ACTION(agent => @339:{HUMAIN} ) )
     +-12-+ UL(cat => {Prep} )
          |    +-13   UL(cat => {Def} )
          +-14-+ UL(cat => {subc} ;sem => LIEU )
               +-15-+ UL(cat => {Prep} )
                    +-16   UL(cat => {subc} )

          +- 1-+ UL(cat => {Def} )
          |    |    +- 2   UL(cat => {ACard} )
          |    +- 3-+ UL(cat => {ACard} )
```

```
          |          +- 4    UL(cat => {ACard} )
      +- 5-+ GN(sem => {HUMAIN} ;cat => {subc} )
      |     +- 6-+ UL(cat => {CoSubN} )
      |          |     +- 7    GN(cat => {PPer} ;fonc => suj ;sem =>
ANIME )
      |          +- 8-+ UL(cat => {VAction} ;syn => PP(sujet =>
GN(sem => @267 ;cat => {PPer} ;fonc => suj ) ;objet => GN(sem =>
@265 ;cat => {subc} ) ) ;sem => SAVOIR(détenteur => @267:ANIME
;objet => @265:{HUMAIN} ) )
      |                +- 9-+ UL(cat => {Prep} )
      |                     +-10    UL(cat => {subp} )
--11-+ UL(sem => ACTION(agent => @376 ) ;syn => PP(sujet =>
GN(cat => {subc} ;sem => @376:{HUMAIN} ) ) ;cat => {VAction} )
     +-12-+ UL(cat => {Prep} )
     |     |     +-13    UL(cat => {Def} )
     |     +-14-+ UL(sem => LIEU ;cat => {subc} )
     +-15-+ UL(cat => {Prep} )
          +-16    UL(cat => {subc} )
```

Thèse de Docteur de l'Université Joseph Fourier (Grenoble I),

Spécialité Informatique

Auteur : Damien Genthial

Titre : Contribution à la construction d'un système robuste d'analyse du français.

Date : 10 Janvier 1991

Mots-clés : analyse syntaxique, hiérarchie de catégories, structures de dépendances, transduction d'arbres, structures de traits, traits sémantiques, correction d'erreurs lexicales, correction d'erreurs syntaxiques

Résumé : La première partie aborde la conception et la mise en œuvre d'un outil d'analyse syntaxique capable de manipuler des informations syntaxiques et sémantiques. La problématique de l'analyse d'une langue naturelle est d'abord présentée : nous essayons de montrer quels sont les invariants de quelques formalismes récents et comment ces invariants ont motivé nos choix. Nous décrivons ensuite le constructeur de structures de dépendances que nous proposons et les apports d'une hiérarchie de catégories à la souplesse et à la tolérance de l'analyse. Les arbres de dépendances produits sont décorées grâce à un formalisme de représentation de la connaissance basé sur des structures de traits intégrant un mécanisme d'héritage. Nous terminons en présentant le prototype d'analyseur que nous avons réalisé.

La deuxième partie définit une architecture pour un système de détection et de correction qui exploite de manière cohérente tous les outils dont nous disposons. Les outils de niveau lexical comprennent un analyseur et un générateur morphologique et des modules de correction lexicale utilisant trois techniques : phonétique, morphologie et clé squelette. Après avoir décrit les objectifs fixés pour le niveau syntaxique, nous donnons un aperçu du vérificateur syntaxique dont nous disposons et nous soulignons les apports des concepts et outils de la première partie à la robustesse des traitements. Enfin, nous proposons l'architecture d'un système complet de détection et correction d'erreurs dans un texte écrit en insistant sur sa portabilité et son adaptabilité.