MASTER EN SCIENCES INFORMATIQUES

Conception d'un analyseur syntaxico-sémantique basé sur les grammaires d'unification dans le cadre du dialogue oral homme-machine

Alexandre, Frédéric

Award date:
1991

Awarding institution:
Universite de Namur

Institut d'Informatique
Facultés Universitaires Notre-Dame de la Paix
21, rue Grandgagnage
5000 NAMUR

Conception d'un analyseur syntaxico-sémantique basé sur les grammaires d'unification dans le cadre du dialogue oral homme-machine

par Frédéric ALEXANDRE

Promoteur : Jacques Berleur s.j.

Mémoire de fin d'études présenté en vue de l'obtention du diplôme de Licencié et Maître en Informatique

Année académique 1990 - 1991

Remerciements

Ce travail de fin d'études n'aurait certainement pas été possible sans l'aide, les conseils et les encouragements de nombreuses personnes que je tiens à remercier.

Jacques Berleur, Recteur des Facultés Universitaires Notre-Dame de la Paix à Namur, a su suscité mon intérêt pour le traitement du dialogue oral homme-machine. Il est également le promoteur de ce mémoire qu'il a supervisé malgré ses nombreuses autres tâches.

Jean-Marie Pierrel, Professeur a l'Université de Nancy I, m'a acceuilli au sein de l'équipe dialogue et m'a également permis de suivre son cours d'"Informatique et Linguistique" qu'il dispense aux étudiants de DEA à Nancy.

Laurent Romary et Bertrand Gaiffe m'ont encadré tout au long de mon stage à Nancy et même encore après. De même, les membres de l'équipe Dialogue du CRIN m'ont permis de m'intégrer dans ce milieu passionnant qu'est la recherche.

Enfin, j'adresse tous mes remerciements à toutes les personnes qui ont collaboré à ce travail de près ou de loin et bien entendu à mes parents, sans qui, la moindre ligne de ce rapport n'aurait pu être écrite.

TABLE DES MATIERES

INTRODUCTION

Depuis l'apparition des machines, l'homme a toujours voulu les diriger par la voix. Les robots qui parlent et à qui l'on peut parler envahissent la science-fiction. L'impossibilité d'utiliser la parole ou le langage naturel est aussi une des raisons pour lesquelles l'outil informatique peut paraître rébarbatif à certaines personnes.

Les informaticiens qui, au début utilisaient un langage qui leur était propre ont modifié l'interface pour que l'utilisateur ne se sente pas trop dépaysé. Nous avons donc vu apparaître des menus déroulants qui permettent une meilleure convivialité. Néanmoins, le meilleur système serait celui où l'utilisateur pourrait utiliser le langage naturel qu'il soit écrit ou oral. Malheureusement, il n'est pas encore possible à l'heure actuelle de réaliser de tels systèmes. Est-ce même possible ou même utile? S'il est évident qu'un ordinateur ne peut "comprendre" le langage oral dans son intégralité, nous pouvons raisonnablement penser qu'il est possible de concevoir des systèmes qui seront opérationnels dans certaines conditions et pour des travaux bien déterminés.

Dans le chapitre 1, nous allons examiner ce qui se fait actuellement dans le domaine du dialogue oral homme-machine et nous détaillerons le système utilisé au Centre de Recherche en Informatique de Nancy (CRIN).

Le chapitre 2 sera consacré à l'étude de différents analyseurs syntaxiques et à leurs lacunes. Cela nous amènera à définir un nouveau mode de représentation et de traitement qui sera basé sur le mécanisme de l'unification et qui sera traité au chapitre 3.

Dans ce chapitre 3, nous nous intéresserons tout d'abord à la théorie traitant de l'unification pour l'appliquer ensuite, dans une approche pratique au traitement du dialogue oral. Nous examinerons ensuite deux théories linguistiques qui utilisent les grammaires basées sur l'unification et qui sont très en vogue actuellement. Nous terminerons ce chapitre par l'examen d'un système de traduction automatique de textes qui utilise ces grammaires et qui a été implémenté dans le cadre du projet Eurotra.

Le chapitre 4 sera consacré à l'étude de l'utilisation des grammaires d'unification aux autres niveaux du processus de traitement du dialogue.

Enfin, nous terminerons par des conclusions et perspectives sur ce mécanisme qu'est l'unification et son utilisation.

CHAPITRE 1. LES INFORMATIONS PROPRES AU DIALOGUE HOMME-MACHINE

1.1. Utilisations possibles et réalisations de systèmes de dialogue oral homme-machine

La communication orale homme-machine peut, dit-on, s'avérer très utile dans un certain nombre de domaines. Lea (Lea 80) fait état des cas où l'utilisateur a besoin de ses mains pour faire autre chose (ex. pour le triage d' objets ou pour le pilotage d'un avion), ou lorsqu'il s'agit de suppléer la présence de l'homme (ex. la machine à dicter, la demande téléphonique de renseignements administratifs), ou de faciliter le travail (ex. la saisie de données, la traduction automatique : Cf. le système CAT 2 au chapitre 3) ou encore d'aider la vie quotidienne (ex. aide aux handicapés, interrogation d'une base de données oralement).

Actuellement, il existe déjà de nombreux systèmes opérationnels (Cf. Haton 91) :

- rédaction de comptes-rendus opératoires pour les chirurgiens : le système Satic de Cognitec et du CRIN/INRIA, ou le produit commercialisé par Kurtzweil aux USA;

- commande d'ordinateur : le système Voice Key de Roar Technology pour l'interface avec un PC, ou le Voice Navigator de Articulate Systems pour MacIntosh;

- commande d'une console sonar du CRIN/INRIA et Thomson DASM avec l'aide de la DRET ...

1.2. Buts et complexité du dialogue oral homme-machine

Le but ultime des systèmes de dialogues homme-machine serait-il de rendre possible une conversation entre un homme et une machine qui serait semblable à un dialogue entre deux êtres humains? Sans vouloir répondre théoriquement à cette question, on constate qu'aujourd'hui - et depuis longtemps déjà - les domaines d'utilisation ont été réduits. Le but actuel est donc de concevoir des systèmes pour des utilisations bien déterminées. Le CRIN, pour sa part essaie notamment de concevoir un système où la machine doit être capable de satisfaire les demandes d'informations concernant les documents administratifs telle qu'on peut les trouver dans les pages roses des annuaires téléphoniques en France (projet DIAL : Cf. Roussanaly 88). Le projet Multiworks (Cf. Gaiffe 90) par ailleurs, tente de créer un système multimodal dont l'un des modes serait le langage parlé. Il s'agit donc de permettre à l'utilisateur de dialoguer avec un ordinateur au moyen du clavier, de la souris et de la parole. L'emploi de la voix dans ce contexte, permet d'utiliser beaucoup plus rapidement la machine qu'en tapant les commandes au clavier, et d'éviter certaines erreurs de compréhension. Un autre avantage est de pouvoir utiliser simultanément la parole et la souris qui indiquera alors l'endroit où la commande orale doit être exécutée.

Dans ces deux projets, le champ d'application a été restreint pour limiter le vocabulaire ainsi que certaines constructions de phrases. En effet, toutes les structures de phrase ne se retrouvent pas forcément dans tous les dialogues. De même, certaines s'y retrouvent beaucoup plus souvent. Pour les demandes d'informations par exemple, on trouvera beaucoup plus de questions que dans une conversation courante et ainsi que des phrases commençant par *est-ce que*. A l'inverse, les phrases exclamatives seront plus rares. Toutes ces indications seront données par l'analyse de corpus. D'autre part, la restriction du champ d'application permet également de lever

certaines ambiguïtés grâce à une meilleure connaissance du contexte.

Pour faciliter également la reconnaissance de la parole, certains systèmes limitent le vocabulaire à un certain nombre de mots. Dans le même ordre d'idées, il est également possible de limiter le nombre de personnes pouvant utiliser le système (Cf. infra) ou encore de ne permettre que la prononciation de mots isolés ou bien séparés. Dans ces derniers cas, nous nous éloignons alors de la véritable communication orale naturelle.

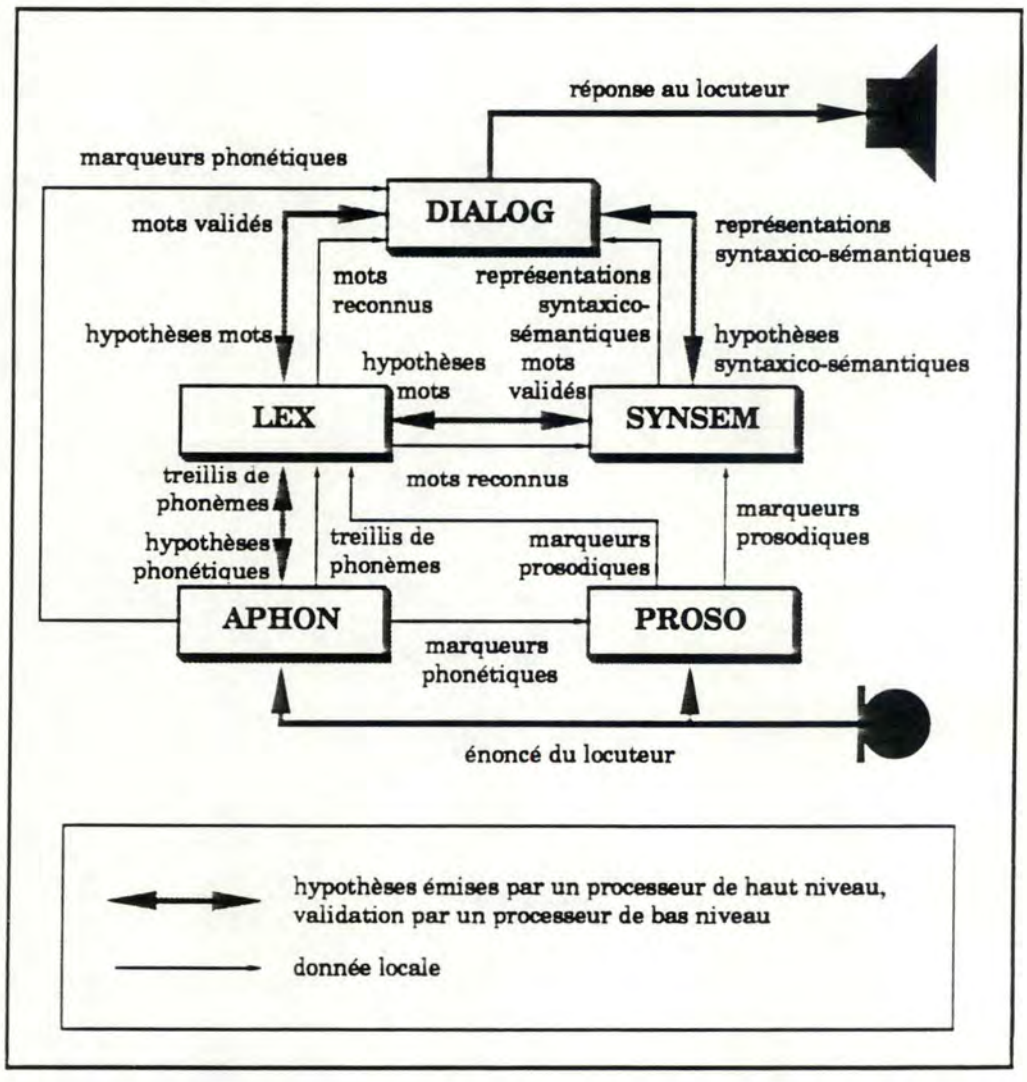
1.3. Principaux niveaux de décomposition d'un système de dialogue oral homme-machine

Nous prendrons comme système de référence le système DIAL qui est actuellement en cours d'élaboration au CRIN, mais il rejoint la décomposition pratiquée par bien d'autres systèmes (Cf. Lea 80). Ce système doit permettre de simuler un interlocuteur humain qui fournirait les informations contenues dans les pages roses des annuaires téléphoniques français et qui concernent les documents administratifs. Voici un exemple de dialogue qui pourrait être traité, S étant le système et L le locuteur (Cf. Romary 89) :

S : *Centre de renseignements administratifs, bonjour.*
L : Bonjour, à qui dois-je m'adresser pour une carte d'identité ?
S : *A un poste de police ou à la mairie.*
L : Quelles sont les formalités ?
S : *Etes-vous mineur ?*
L : Non, non.

S : *Vous devez présenter votre livret de famille ou une fiche d'état civil et fournir un timbre fiscal.*
L : Combien coûte un timbre fiscal ?
S : *115 F.*
U : Merci.
S : *Désirez-vous autre chose ?*
U : Non. Merci. Au revoir.
S : *Au revoir*

Nous allons donc examiner l'architecture du système DIAL en partant du niveau le plus bas jusqu'au niveau le plus élevé.



Architecture du système DIAL (Mousel 90)

1.3.1. Décodage acoustico-phonétique

Ce niveau transforme le signal vocal émis par le locuteur en un treillis de phonèmes. Cette phase est difficile à maîtriser car il s'agit de décoder les mots de différents locuteurs. La difficulté vient du fait que les personnes ne prononcent pas toujours les mots de la même façon (accents) et qu'une même personne, suivant son état de santé par exemple, aura une prononciation différente. Pour faciliter le décodage, certains systèmes peuvent demander à l'interlocuteur de prononcer une phrase bien déterminée sur laquelle il pourra se baser par la suite. Au CRIN, le module APHON utilise un système expert, APHODEX (Fohr 87) qui est multilocuteur et qui ne nécessite aucun apprentissage. Celui-ci prend en compte les segments qui se trouvent à côté de celui qui est analysé. Ce processus permet de déterminer plus facilement le phonème grâce à une connaissance générale de ces segments. En fait, il tient compte du contexte grâce aux règles contenues dans la base de connaissance du système expert.

1.3.2. La prosodie

La prosodie est l'étude de la "mélodie" de la phrase. Cette étude peut amener deux types de résultats. D'une part, elle peut déterminer le type de la phrase, à savoir si c'est une affirmative, une interrogative et cela en se basant sur l'intonation générale. Ce résultat sera surtout utilisé par le module d'analyse syntaxique qui pourra déterminer plus facilement la structure de la phrase. D'autre part, l'étude de la prosodie permet, à l'intérieur même de la phrase, de délimiter les mots ou les groupes de mots et de permettre ainsi d'éviter parfois les ambiguïtés. Par exemple, si l'on examine la phrase 'la belle ferme le voile', elle peut être comprise de deux façons différentes suivant l'endroit où l'on place les marqueurs prosodiques (/) : 'la belle ferme / le voile' ou 'la belle / ferme le voile'. Seule l'étude de la

prosodie peut lever facilement cette ambiguïté sans faire appel au contexte. Ce module PROSO n'est pas encore vraiment développé dans le système DIAL.

1.3.3. Le lexique

Le module *lexique* reçoit du module de décodage acoustico-phonétique un treillis de phonèmes et doit reconstruire les mots de la phrase. Les résultats de l'analyse lexicale seront très utiles dans notre approche car ils constituent la source d'information sur le vocabulaire de la phrase. En effet, grâce à une recherche dans les dictionnaires, ce module fournit aussi des renseignements concernant les mots de la phrase (Cf. infra la fonction *analyse lexicale*). Il est important, par exemple pour le niveau syntaxique de connaître les différentes fonctions grammaticales que peut avoir un mot. De même, le niveau sémantique aura probablement besoin d'informations sur l'utilisation d'un mot dans tel ou tel contexte (Cf. infra la théorie des grammaires de cas notamment).

Actuellement, le lexique prend de plus en plus d'importance. Pour s'en rendre compte, il suffit de se rapporter aux travaux de Gaston Gross au LADL (Gross 88) qui a notamment fait une étude sur les mots composés du français et où l'on remarque que l'étude sémantique des mots pris isolément ne peut pas toujours se faire. La composition de deux noms peut apporter un sens différent de celui de chacun d'eux. Ainsi, une expression figée telle que "le grand patron" ne désigne pas une personne qui est grande, mais bien un directeur. "Le nom composé évoque dans l'esprit, non les images distinctes répondant à chacun des mots correspondants, mais une image unique (hôtel-de-ville, pomme de terre, arc de triomphe)" (Grevisse 69). La simple liste de mots ne suffit donc plus si l'on veut faire une bonne analyse tant syntaxique que sémantique.

Ce module peut également venir en aide au module de décodage acoustico-phonétique en lui soumettant des hypothèses de travail pour tel ou tel groupe de phonèmes. Cela constitue donc un travail de prédiction qui permettra une amélioration de la reconnaissance des mots et donc diminuera les hypothèses émises par ce niveau.

Cette fonction d'analyse lexicale est assurée par le module LEX dans le système DIAL, mais n'est pas encore totalement terminée.

1.3.4. La syntaxe et la sémantique

La syntaxe peut être définie comme l'ensemble des règles de formation des phrases. La sémantique définit les règles liées au sens des mots, à leur signification. Dans certains systèmes, ces deux modules sont séparés (Cf. infra le système CAT2). Le CRIN a préféré les laisser ensemble dans la mesure où il est très difficile de faire la distinction entre les deux niveaux. Doit-on, par exemple, considérer que le genre d'un nom fait partie de la syntaxe ou de la sémantique? Tout dépend de l'importance que l'on accorde à chacun de ces niveaux. Dans l'approche pratique qui sera développée au chapitre 3, aucune distinction n'est faite entre les éléments dits syntaxiques et ceux dits sémantiques.

Actuellement, le module SYNSEM du CRIN est basé sur les R.N.P. (Réseaux à Noeuds Procéduraux : Cf. chapitre 2).

1.3.5. La gestion du dialogue

Ce module constitue le sommet de l'architecture. Il représente l'aspect pragmatique du système. C'est ce module qui doit gérer les informations qui viennent non seulement des

niveaux inférieurs mais aussi des informations sur le monde du discours. C'est ce niveau qui va gérer l'historique du dialogue et donc aussi mettre à jour les données et cela grâce aux nouvelles phrases reconnues. Ce module doit donc comprendre des procédures qui vont faire des "raisonnements" sur les données et pouvoir ainsi déterminer les requêtes du locuteur et essayer de les satisfaire. Pour ce faire, il est parfois nécessaire de poser des questions à l'interlocuteur pour avoir de meilleurs renseignements sur la tâche ou alors faire des suppositions qui, si elles ne sont pas invalidées seront acceptées.

C'est également ce module qui doit gérer tous les problèmes liés aux références anaphoriques interphrastiques (qui reprennent l'idée exprimée dans une phrase antérieure; ex. : les pronoms), aux déictiques (mots dont la référence fait partie de la situation; ex. : *celui-ci*), aux ellipses ou aux phrases incomplètes.

1.3.6. Circulation du flux d'informations

Il est bien évident que des flux d'informations circulent entre ces différents modules. Les modules des niveaux inférieurs fournissent le résultat de leur travail aux niveaux supérieurs comme nous l'avons vu pour le module de décodage acoustico-phonétique qui fournit au module lexical un treillis de phonèmes qu'il va recomposer en mots. Mais un module de haut niveau peut fournir des prédictions aux niveaux inférieurs. C'est le cas notamment du module syntaxico-sémantique qui peut fournir des renseignements sur le mot que le module lexical doit identifier (Cf. schéma page suivante). Dans ce système, chaque module a sa propre base de connaissances et sa propre représentation.

1.4. Conclusion

Nous avons vu dans ce premier chapitre l'utilité et les réalisations des systèmes de dialogues oraux homme-machine. Nous avons également donné un aperçu de la composition de tels systèmes et en particulier du système DIAL qui est en cours d'élaboration au CRIN. Dans les chapitres suivants, nous nous intéresserons plus particulièrement au problème de l'analyse syntaxico-sémantique et aux solutions qui peuvent y être apportées.

CHAPITRE 2. LES ANALYSEURS SYNTAXIQUES

2.1. Introduction

Les objectifs de l'analyse syntaxique sont principalement au nombre de trois :

- vérifier si la phrase est bien formée, c'est-à-dire si elle appartient à la langue utilisée. Toutefois, il faut être prudent vis-à-vis de cet objectif. En effet, si toutes les règles de grammaires n'ont pas été prises en compte lors de la conception de l'analyseur, certaines phrases ne sont pas directement analysables par le système qui va les rejeter alors qu'elles sont correctes;

- construire la structure syntaxique de la phrase qui sera alors utilisée par les modules des niveaux supérieurs (le module pragmatique par exemple). En général, il s'agit d'un arbre syntaxique dont la racine représente le type phrase et les feuilles les mots;

- l'analyse syntaxique sert également à guider la reconnaissance en prédisant par exemple le type du mot suivant. Dans le cas de la reconnaissance de la parole, ce point peu être très important dans la mesure où l'analyse syntaxique peut aider le module lexicale ou bien encore le module de décodage acoustico-phonétique.

2.2. Méthodes principales d'analyse syntaxique

2.2.1. Analyseurs en hauteur et en largeur

Un des objectifs des analyseurs est donc de construire la structure syntaxique de la phrase. Le processus d'analyse se résume souvent à la construction d'un arbre syntaxique. Trois solutions sont alors envisageables (Cf. Pierrel 87) :

- les analyseurs ascendants partent des feuilles, c'est-à-dire des mots terminaux du langage et remontent à la racine.

- les analyseurs descendants partent de la racine et vont vers les feuilles.

Ces deux solutions travaillent généralement de gauche à droite bien que rien à priori, ne les empêche de travailler de droite à gauche même si cela paraît moins logique et ne correspond sans doute pas à la manière dont l'homme interprète un texte;

- certains analyseurs combinent ces deux méthodes : les analyseurs mixtes sont donc tantôt ascendants et tantôt descendants. Contrairement aux deux premiers, ils travaillent souvent du milieu vers les côtés. Partant d'un mot qui constitue un îlot de confiance et qui a été parfaitement identifié lors du décodage acoustico-phonétique par exemple, on tente de déterminer les mots qui sont directement à gauche ou à droite. Cette méthode est particulièrement utile lors du traitement de la parole où tous les mots ne sont pas toujours parfaitement reconnus.

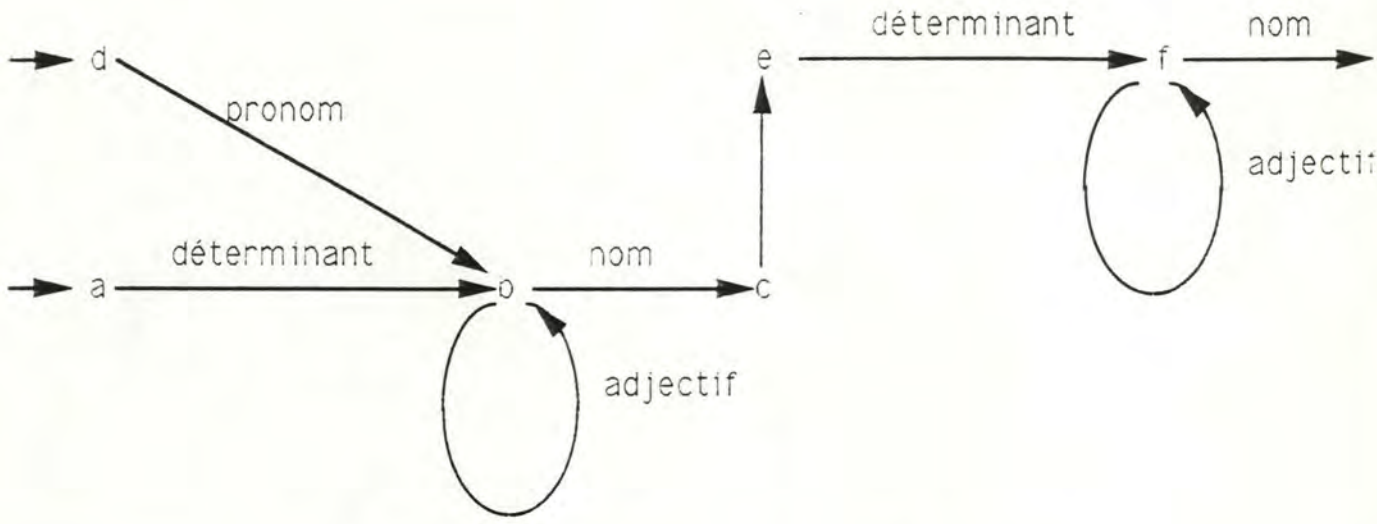
Outre ces analyseurs classiques, examinons deux types d'analyseurs fondés sur des réseaux syntaxiques.

2.2.2. A.T.N.

Les A.T.N. ou Augmented Transition Networks ont été définis par Woods en 1970 dans son article "Transition Network Grammar for Natural Language Analysis". (Sabah 88 vol. 2) ou (Winograd 83) en donnent une étude complète.

Cette théorie repose sur les réseaux de transitions. Ce modèle est constitué d'un réseau de noeuds et d'arcs orientés. Les noeuds représentent les états dans une machine à états finis et les arcs les transitions. Chaque arc est étiqueté par un symbole dont l'input provoque la transition de l'état du début de l'arc vers l'état de la fin de l'arc. Les arcs représentent des catégories lexicales (nom adjectif, déterminant,...) ou des mots. Le réseaux possède des arcs initiaux qui sont les points d'entrée et des arcs finaux qui sont les points de sortie.

Malheureusement, ce système ne permet pas de suspendre le processus d'analyse en cours pour l'utiliser à nouveau pour analyser un constituant imbriqué. Tous les composants sont au même niveau. La représentation d'une phrase se fait donc avec des catégories lexicales et non en fonction de classes syntaxiques telles les groupes nominaux, les groupes verbaux, etc. qui permettent d'avoir une représentation sous la forme d'un arbre syntaxique. Ces arbres sont d'une part beaucoup plus lisibles et permettent la généricité, c'est-à-dire que l'on va pouvoir travailler au niveau de classes plutôt qu'au niveau de catégories et donc a un niveau d'abstraction plus élevée.



exemple de réseau de transition non récursif

Une première idée a donc été d'ajouter ce mécanisme de généricité, ce qui a donné les "Recursive Transition Networks" (R.T.N.) ou "réseaux de transitions récursifs". Dans ce système, les arcs peuvent représenter des noms, des catégories lexicales mais également des catégories syntaxiques du style groupe nominal. Lorsque l'analyseur arrive sur un arc de ce type, il doit alors vérifier que les mots qui suivent dans la phrase correspondent aux éléments de cette catégorie. Il faut donc appliquer le mécanisme d'analyse à cet élément, d'où le nom de récursif. En fait, une catégorie syntaxique correspond à un sous-réseau qu'il faut explorer. Un nouveau type d'arc a été ajouté : l'arc JUMP qui permet d'éviter de passer par certaines catégories qui ne sont pas obligatoires. Cela ne change pas la puissance du réseau, mais simplifie leur écriture.

Les réseaux de transition récursifs constituent des grammaires hors contexte. Mais, le français, tout comme l'anglais, demande plus pour être analysé correctement. Par exemple, les problèmes d'accord nécessitent une connaissance sur certains autres mots de la phrase pour être vérifiés. Il faut donc avoir une connaissance du "contexte". C'est pourquoi Woods ajoute à chaque arc du réseau une série de conditions qui doivent être remplies pour y accéder et une série

d'actions qui devront être exécutées si l'on utilise cet arc. C'est la raison pour laquelle ces nouveaux réseaux ont été appelés "réseaux de transition augmentés" (Augmented Transition Networks ou A.T.N.). Les actions vont servir à construire des descriptions des parties analysées. Ces descriptions seront contenues dans des registres qui contiennent aussi des paramètres qui seront testés lors de la vérification de conditions.

Nous reprenons ici les définitions telles que Woods les a données dans son article de 1970. Nous donnerons parfois les noms utilisés dans d'autres articles notamment chez Winograd et Sabah.

Les A.T.N. peuvent contenir 4 sortes d'arcs différents : le *category arc* ('CAT Arc'), dont l'étiquette est une catégorie lexicale et qui doit être celle du mot en entrée; *PUSH arc* (ou *seek arc* chez Winograd) spécifie un appel récursif à un réseau, son étiquette est celle du réseau ou d'un état de celui-ci; le *TST arc* est un arc qui permet un test arbitraire pour déterminer si un arc doit suivre; le *pop arc* (ou *send arc* chez Winograd) est un arc factice qui indique sous quelles conditions l'état peut être considéré comme état final et il effectue les actions qui découlent du fait que l'on ait réussi l'analyse.

Les deux actions terminales possibles, TO et JUMP, indiquent si le pointeur peut être avancé ou non. En fait, le TO indique l'état à envisager avec le mot suivant tandis que le JUMP indique l'état à envisager avec le même mot.

Les trois *actions* changent la valeur du registre indiqué par celle indiquée dans la *form* (fonction). *SETR* indique que cela doit être fait au niveau courant, *SENDR* au prochain niveau inférieur d'imbrication et *LIFTR* au niveau supérieur. C'est donc grâce à cette *action* que l'on peut remonter de l'information. Les *forms* sont en fait des fonctions : *GETR* est une fonction dont la valeur est le contenu du registre indiqué; * renvoie le mot courant; *GETF* est une fonction qui détermine la valeur d'un attribut pour le mot courant; *BUILDQ* est une fonction qui prend une liste qui représente un fragment d'arbre avec des noeuds marqués et renvoie comme valeur le résultat de la substitution de ces noeuds marqués par le contenu des registres indiqués.

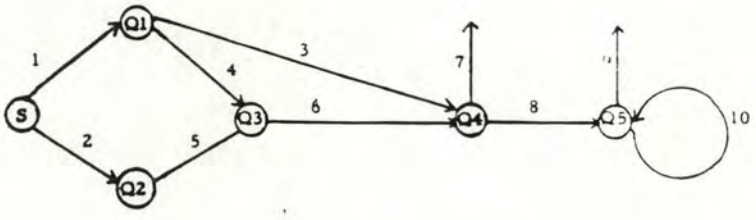
Spécifications du langage pour représenter un A.T.N. (Woods 70) : [1]

```
<transition network> -> (<arc set><arc set>*)
<arc> -> (CAT <category name><test><action>*<term act>)/
         (PUSH <state><test><action>*<termact>)/
         (TST <arbitrary label><test><action>*<term act>)/
         (POP <form><test>)
<action> -> (SETR <register> <form>)/
            (sendr <register><form>)/
            (LIFTR <register><form>)
<term act> -> (TO <state>)/
              (JUMP <state>)
<form> -> (GETR <register>)/
          */
          (GETF <feature>)/
          (BUILDQ <fragment><register>*)/
          (LIST <form>*)/
          (APPEND <form><form>)/
          (QUOTE <arbitrary structure>)
```

1 : * signifie répétition possible tandis que / indique l'alternative.

Pour représenter un A.T.N., il ne suffit pas d'avoir la représentation du réseau, il faut aussi avoir un tableau où sont reprises les actions attachées aux arcs du réseau.

Les actions et conditions sont conçues pour un analyseur pariculier qui travaille de gauche à droite et de bas en haut. Cela enlève la flexibilité lors de la conception de l'analyseur, mais cela permet aussi de mieux gérer la grammaire en connaissant la stratégie utilisée.



état de départ	arc	état d'arrivée	Test	Actions
S	1	Q1	NP	SETR SUBJ * SETR TYPE (QUOTE DCL)
	2	Q2	AUX	SETR AUX * SETR TYPE (QUOTE Q)
Q1	3	Q4	V	SETR AUX NIL SETR V *
	4	Q3	AUX	SETR AUX *
Q2	5	Q3	NP	SETR SUBJ *
Q3	6	Q4	V	SETR V *
Q4	7	END	POP	BUILDQ (S+++ (VP +)) TYPE SUBJ AUX V
	8	Q5	NP	SETR VP (BUILDQ (VP (V+) *) V)
Q5	9	END	POP	BUILDQ (S++++) TYPE SUBJ AUX VP
	10	Q5	PP	SETR VP (APPEND (GETR VP) (LIST *)))

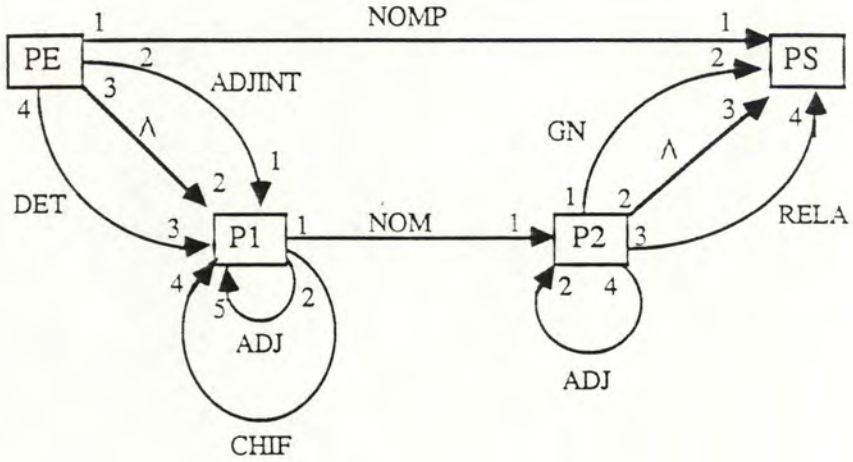
Exemple d'A. T. N. tiré de WOODS

2.2.3. R.N.P

Les réseaux syntaxico-sémantiques à noeuds procéduraux ou R.N.P. ont été développés au CRIN par J.M. Pierrel pour le système Myrtille II (Cf. Pierrel 81) et sont repris dans le système DIAL (Cf. Mousel 90 ou Deville 89).

Tout comme les A.T.N., les R.N.P. décrivent la syntaxe d'un langage à l'aide de réseaux. Contrairement aux A.T.N., les procédures sont attachées aux noeuds plutôt qu'aux arcs. Ces procédures servent à déterminer l'ordre le plus probable des éléments de la phrase. Elles indiquent donc avec une certaine probabilité, les arcs suivants en fonction de l'arc précédent, de la prosodie, d'aspects phonétiques et syntaxico-sémantiques (ex. : la probabilité d'apparition de tel type de construction de phrase). Il y a donc une prise en compte du contexte. Ce mécanisme permet également de fournir des informations au niveau lexical en indiquant le type de mot suivant. Il y a donc un aspect de prédiction qui n'est pas à négliger dans un système de gestion du dialogue oral. La construction de la structure syntaxico-sémantique est assurée par des procédures extérieures. Ces procédures externes sont conçues indépendamment du réseau et donc de la grammaire utilisée. En fait, les R.N.P. séparent la représentation des connaissances et leur traitement. "Tandis que les A.T.N. sont composés d'une grammaire et d'un analyseur dans un seul réseau, les R.N.P. sont une grammaire qui est exploitée par un analyseur séparé. L'analyseur indique quel chemin ou quelle stratégie suivre, de gauche à droite, de droite à gauche ou du milieu vers les côtés, ascendant ou descendant. Lorsque la grammaire doit être changée pour un autre type de sous-langage, l'analyseur reste le même." (Deville 89 p. 117)

A titre d'exemple, voici le sous-réseau du groupe nominal tel qu'on pourrait le trouver dans le R.N.P. de DIAL (Pierrel 87).



PE : Procédure d'entrée
PS : Procédure de sortie
P1 : Procédure interne
P2 : Procédure interne
Λ : Branche vide
ADJINT : Adjectif interrogatif (accès au lexique)
ADJ : Adjectif (accès au lexique)
CHIF : Chiffre (référence à un sous-réseau)
DET : Déterminant (accès au lexique)
GN : Groupe nominal (référence à un sous-réseau)
NOMP : Nom propre (accès au lexique)
NOM : Nom (référence à un sous-réseau)
RELA : Proposition relative (référence à un sous-réseau)

Sous-réseau groupe nominal.

2.2.4. Les modèles markoviens

Les modèles markoviens (Hidden Markov Model, HMM) sont basés sur les probabilités. En effet, en regardant de plus près les phrases du français par exemple, on se rend compte que certaines constructions de phrases reviennent beaucoup plus souvent que d'autres. Ainsi, il y a beaucoup de phrases composées d'un sujet, d'un verbe et d'un complément. Il peut en être de même pour les mots : ainsi certains mots sont très peu utilisés dans certains domaines. De là, découle l'idée de donner des taux de probabilité à certaines constructions et de vérifier les constructions les plus probables.

Cette théorie découle donc des modèles de Markov c'est-à-dire des processus stochastiques. La probabilité d'apparition d'un état E_t connaissant les états E_i pour i inférieur ou égal à t-1 est égale à la probabilité d'apparition de E_t connaissant les états allant de t-p à t-1, p définissant l'ordre du processus. L'apparition de l'état E_t ne dépend donc que des p états précédents et pas de tous. En général, on prend p égal à deux ou trois et donc on prend une série de 3 ou 4 mots.

Deux types de modèles peuvent découler de cette théorie : le modèle n-grammes où les probabilités sont calculées sur les mots ou alors le modèle n-classes qui prend en compte les classes syntaxiques. Le problème principal dans ce genre de technique est de déterminer les probabilités de transitions. En effet, il n'est pas possible, dans un premier temps de déterminer automatiquement les classes syntaxiques des mots dans une phrase si ce n'est en utilisant un autre analyseur. Il faut donc procéder manuellement. Or, pour avoir une bonne connaissance de notre langue, il est nécessaire d'évaluer plusieurs millions de mots. Lorsque le modèle commence à être bon, on peut l'utiliser pour affiner les probabilités, mais il faut toujours effectuer une vérification manuelle pour lever certaines ambiguïtés. Ainsi, certains mots peuvent appartenir à plusieurs classes syntaxiques. Par exemple, le mot *porte* peut être considéré tantôt comme un verbe et tantôt comme un nom.

Les résultats de ces modèles ne sont pas toujours très utilisables par les autres modules de traitement car ils n'effectuent qu'une identification des catégories syntaxiques et non une véritable analyse syntaxique. Ils peuvent néanmoins s'avérer très intéressant pour faire des prédictions, mais nécessitent un long travail de recherches dans les corpus pour bien déterminer les probabilités.

Ces modèles sont utilisés au CRIN pour l'analyse syntaxique dans le projet de la machine à dicter. Ils sont également souvent employés dans les systèmes de décodage acoustico-phonétique (Cf. Kita 89 ou encore Marino 90) où ils apparaissent comme une méthode des plus prometteuses.

2.3. Conclusion

Nous venons d'examiner différentes sortes d'analyseurs qui ne sont pas toujours suffisants pour le traitement du langage naturel.

Les analyseurs classiques, tels que définis en 2.2.1, n'apportent pas suffisamment d'informations, par exemple, pour traiter facilement la vérification des accords. Pour qu'ils soient plus efficaces, il faudrait leur adjoindre un mécanisme qui fasse notamment ces vérifications et qui traite plus d'informations que la simple détermination de la classe syntaxique des mots.

Les A.T.N. sont déjà plus efficaces dans la mesure où, grâce aux registres et aux procédures attachées aux arcs, ils prennent en compte le contexte immédiat. Certaines vérifications sont donc possibles. Néanmoins, le fait de lier la représentation des connaissances et leur traitement oblige le concepteur à les envisager simultanément et toute modification à l'un entraîne une modification à l'autre. Cela n'est donc pas très pratique pour réaliser un système expérimental.

Quant aux R.N.P., ils évitent ce problème en séparant la représentation des connaissances et leur traitement. Néanmoins, il faut concevoir pour chaque noeud une nouvelle procédure. De plus, il n'existe pas de point d'ancrage qui permette de traiter les dépendances éloignées.

Les modèles markoviens ont l'avantage d'être basés complètement sur le vécu mais sont diffilement réalisables. De plus ils ne donnent qu'une simple représentation syntaxique et ne peuvent être utilisés tels quels. En effet, dans un système de traitement du dialogue, l'information sémantique est également nécessaire et utile et doit alors être traitée séparément.

Ces modèles sont essentiellemnt dirigés par la syntaxe et ne donnent que peu d'informations sémantiques bien que les A.T.N. et les R.N.P. tentent de les intégrer.

Dans le chapitre suivant, nous alons étudier un système de représentation qui, par le fait qu'il donne une plus grande importance au lexique, apporte plus d'information sémantique. Ce système doit permettre de séparer la représentation des connaissances de leur traitement. Cela permet d'avoir une approche incrémentale et de changer plus facilement d'architecture. De même, la représentation des connaissances doit permettre une vérification aisée de certaines règles telles que celles des accords en genre et en nombre ou encore le traitement des dépendances éloignées.

CHAPITRE 3. LES GRAMMAIRES BASEES SUR L'UNIFICATION

Au chapitre 2, nous avons donné les principales qualités qu'un analyseur devait avoir. Dans ce chapitre, nous allons étudier une méthode faisant appel principalement au lexique comme source de connaissances. Elle est d'ailleurs classée dans les grammaires lexicales alors que les A.T.N. sont plutôt une méthode basée sur la syntaxe.

Dans ce chapitre, nous allons étudier une méthode pour représenter l'information linguistique et un mécanisme associé à cette méthode qui permet d'analyser simultanément la syntaxe et la sémantique dans le cadre du dialogue oral homme-machine. Nous définissons tout d'abord le cadre théorique pour ensuite l'étendre au cas spécifique de la reconnaissance de la parole. Dans un troisième temps, nous reprenons les aspects plus linguistiques développés pour d'autres systèmes. Enfin nous terminerons par un exemple pratique, à savoir la traduction automatique. Cela permettra d'insérer la théorie dans la pratique.

3.1. Approche théorique

Les grammaires d'unification sont des grammaires qui font une grande place au lexique dont elles exploitent pleinement les caractéristiques syntaxiques et sémantiques. Nous trouverons donc dans ce lexique les éléments nécessaires à l'analyse, à savoir les catégories grammaticales (nom, verbe, déterminant, ...), les données concernant les accords (genre, nombre, personne, ...), ainsi que des éléments plus "sémantiques" (verbe transitif, cas nominaux, ...).

Nous verrons dans la suite que les grammaires d'unification peuvent représenter les structures profondes à

partir des structures de surface. Elles constituent un outil de description et de traitement de l'information qui est de plus en plus utilisé actuellement.

L'article de Stuart Shieber : "An Introduction to Unification-based Approches to Grammar" de 1985, peut constituer un bon point de départ. Néanmoins, c'est Martin Kay qui a établi les bases de cette théorie dans son article "Parsing in functional unification grammar" en 1982.

3.1.1. Définitions

Une structure de traits est un ensemble de couples < attribut, valeur >. Une valeur peut être un symbole atomique, mais elle peut également être une structure de traits enchâssée. L'ordre des traits n'a aucune importance mais un trait ne peut être défini qu'une seule fois.

Ce que nous appelons structure de traits est appelé chez Kay une Description Fonctionnelle (FD) et les attributs des descripteurs. En effet, nous pouvons définir les structures de traits comme une fonction d'un ensemble de traits vers un ensemble de valeurs.

D'après Shieber, l'unification de deux structures de traits est la combinaison qui consiste à prendre l'union des paires < attribut, valeur > et dans le cas où les deux ensembles assignent des valeurs à un même trait, à combiner ces valeurs récursivement. Son but est de combiner deux structures de traits en vue d'obtenir l'information contenue dans chacune d'elles et rien que celle-là. C'est cette définition que nous avons utilisée dans notre travail. Deux structures sont incompatibles si elles ont un attribut en commun et si leurs valeurs sont différentes. C'est à dire que s'il s'agit de symboles atomiques, ils sont inégaux ou s'il

s'agit de structures, qu'elles ne sont pas unifiables. Deux solutions peuvent alors être envisagées : la première dit simplement qu'il y a échec et la deuxième consiste à créer une structure spéciale qui sera alors surspécifiée (trop d'informations) et sera donc inconsistante et sans valeur linguistique.

La notation pour représenter une structure de traits est la suivante :

$$\begin{bmatrix} \text{nombre singulier} \\ \text{personne troisième} \end{bmatrix}$$

Au trait *nombre* est associée la valeur *singulier* et au trait *personne*, la valeur *troisième*. La représentation est similaire pour les attributs à valeur complexe, il suffit de l'intégrer directement.

$$\begin{bmatrix} \text{cat gn} \\ \text{accord} \quad \begin{bmatrix} \text{nombre singulier} \\ \text{personne troisième} \end{bmatrix} \end{bmatrix}$$

On définit aussi la notion de chemin par la séquence finie d'attributs qui aboutit à une valeur, celle du chemin. Par exemple, la valeur du chemin (accord - nombre) est *singulier*. Les chemins permettent donc la sélection d'une partie d'une structure de traits. Exemple d'unification :

$$\begin{bmatrix} \text{cat SN} \\ \text{accord} \quad \begin{bmatrix} \text{nombre singulier} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{accord} \quad \begin{bmatrix} \text{genre masculin} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \text{cat SN} \\ \text{accord} \quad \begin{bmatrix} \text{nombre singulier} \\ \text{genre masculin} \end{bmatrix} \end{bmatrix}$$

Il est à noter qu'une structure de traits peut être représentée par un graphe acyclique orienté (directed acyclic graphs : dags). L'unification consiste alors à superposer les deux graphes.

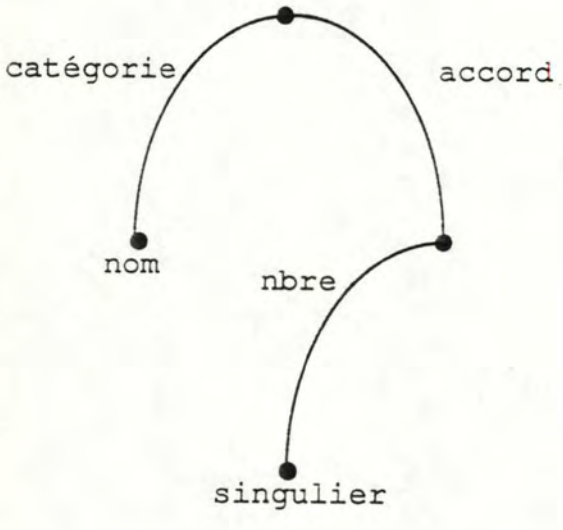


schéma 1

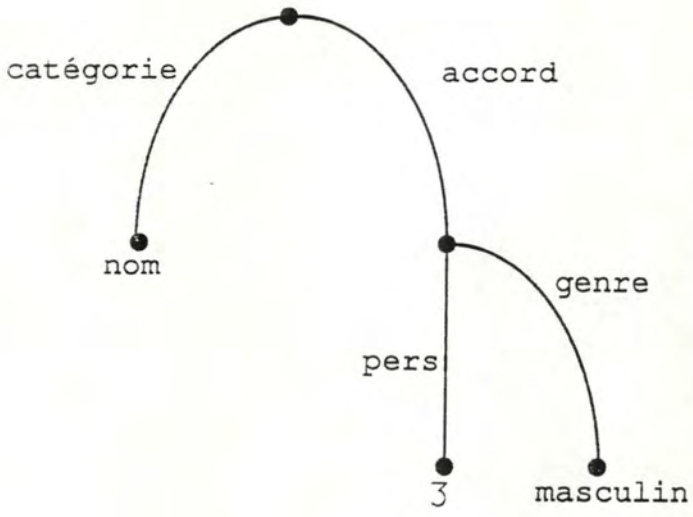


schéma 2

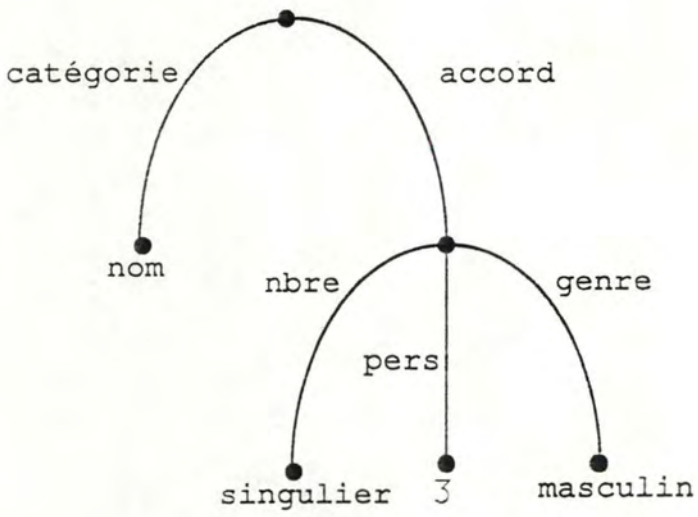


schéma 3 = unification des schémas 1 et 2

3.1.2. Propriétés

Propriétés générales : (exemples page suivante)

1) l'unification ajoute de l'information : ceci est le but principal.

2) l'unification est idempotente : cela permet d'éviter certaines redondances inutiles et donc de surcharger les structures de traits.

3) les variables sont les éléments identité de l'unification.

4) l'unification se fait indépendamment de l'ordre des traits.

5) l'unification est commutative : A U B = B U A

6) l'unification est associative :

(A U B) U C = A U (B U C)

Propriété particulière : le lexique et la grammaire sont représentés de la même façon au moyen de structures de traits. Nous allons voir que cela peut s'étendre aussi à l'implémentation des règles et à leur vérification.

l'unification ajoute de l'information

$$\begin{bmatrix} \text{cat} & \text{SN} \end{bmatrix} \sqcup \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{nombre} & \text{singulier} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \text{cat} & \text{SN} \\ \text{accord} & \begin{bmatrix} \text{nombre} & \text{singulier} \end{bmatrix} \end{bmatrix}$$

l'unification est indempotente

$$\begin{bmatrix} \text{cat} & \text{SN} \end{bmatrix} \sqcup \begin{bmatrix} \text{cat} & \text{SN} \\ \text{accord} & \begin{bmatrix} \text{nombre} : \text{singulier} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \text{cat} & \text{SN} \\ \text{accord} & \begin{bmatrix} \text{nombre} & \text{singulier} \end{bmatrix} \end{bmatrix}$$

les variables sont l'élément neutre de l'unification

$$\begin{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{cat} & \text{SN} \end{bmatrix} = \begin{bmatrix} \text{cat} & \text{SN} \end{bmatrix}$$

3.1.3. Les structures partagées

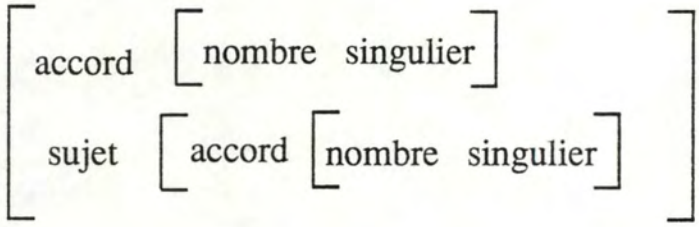
Le mécanisme des structures partagées, anodin au premier abord, va donner toute sa puissance au système.

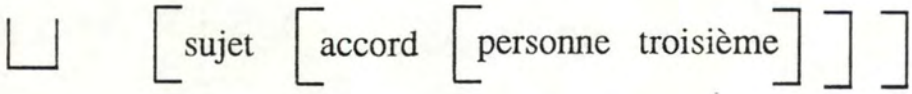
Principe : une même valeur ou une même sous-structure est partagée par plusieurs traits et est identifiée par une étiquette. Cette étiquette se compose d'un signe distinctif, à savoir, dans notre implémentation, d'une @ suivit d'un numéro identifiant et d'une flèche indiquant que la valeur qui suit est bien celle qui se rapporte à l'étiquette (ex. : [accord @ 1 <- [nombre singulier]]). Dans la suite de la structure de trait, seule le @ et le numéro d'étiquette subsiste car il n'est plus nécessaire d'indiquer sa valeur vu qu'elle est déjà connue.

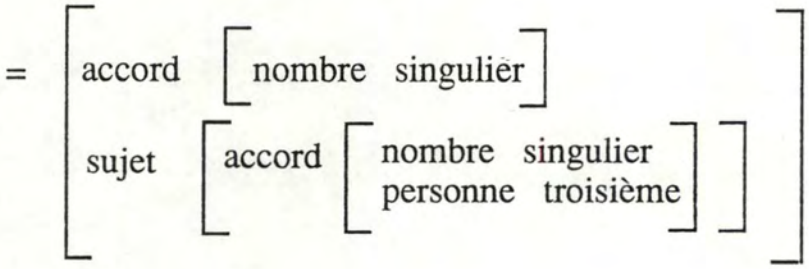
Le partage de structures implique que lorsque la valeur de ce trait est modifiée, elle est modifiée partout où l'étiquette se trouve dans la structure principale, ce qui n'est évidemment pas le cas sans ce mécanisme (Cf. exemples pages suivantes). Ce mécanisme va nous permettre, d'une part, d'intégrer directement les règles de grammaire et, d'autre part, de pouvoir répartir les informations. Ainsi, si une information est complétée à un niveau inférieur lors d'une unification, elle peut être rapidement remontée ou être utilisée lors d'une autre unification.

Le partage de structures introduit la notion d'égalité de valeur, ce qui est plus puissant que l'égalité d'instance ou de type. L'égalité d'instance ne "partage" pas, en effet, l'adjonction d'information qui peut être apportée. De plus, s'il est utilisé efficacement, ce mécanisme permet de gagner de la place mémoire, vu que l'on ne recopie pas la valeur mais seulement le pointeur de cette valeur (exemple pages suivantes).

Structures non partagées







Structures partagées

$$\left[\begin{array}{l} \text{accord} \ @ \ 1 \text{ <- } \left[\text{nombre} \ \ \text{singulier}\right] \\ \text{sujet} \ \ \left[\text{accord} \ @ \ 1\right] \end{array}\right]$$

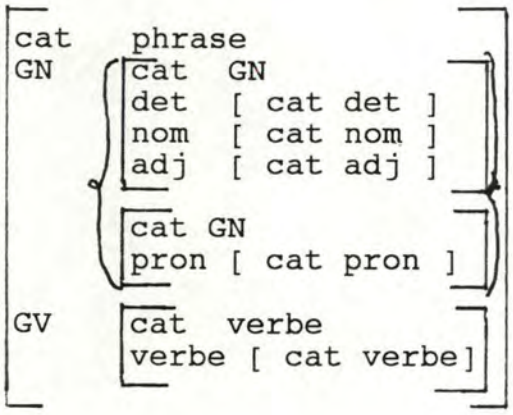
$$\sqcup \qquad \left[\text{sujet} \ \left[\text{accord} \ \left[\text{personne} \ \ \text{troisieme}\right]\right]\right]$$

$$= \left[\begin{array}{l} \text{accord} \ @ \ 1 \text{ <- } \left[\begin{array}{l}\text{nombre} \ \ \text{singulier} \\ \text{personne} \ \ \text{troisième}\end{array}\right] \\ \text{sujet} \ \ \left[\text{accord} \ @ \ 1\right] \end{array}\right]$$

Lorsqu'on utilise des graphes (dags) pour représenter des structures, ils deviennent vite illisibles si on utilise des structures partagées.

3.1.4. Extensions possibles

Il est également possible de représenter dans ce formalisme l'alternative et la négation. L'alternative peut être utile dans la mesure où elle permet de réduire le nombre de structures de traits en en définissant plusieurs dans une.



Si l'alternative peut se concevoir facilement, il n'en n'est pas de même pour la négation. En effet, que signifie par exemple, nier la troisième personne pour un verbe? Est-ce la deuxième ou la première? La négation ne peut être correctement définie que pour un attribut à deux valeurs possibles et donc ne présente pas d'intérêt réel. Le seul intérêt serait celui de représenter une contrainte et donc, obliger l'unification à ne se faire que sur une structure qui n'aurait pas la propriété niée ce qui n'est pas vraiment dans la philosophie de l'unification.

3.2. Approche pratique

3.2.1. Domaine d'utilisation

Nous voudrions maintenant montrer la nécessité d'adjonctions à une structure de traits pour qu'elle puisse être pleinement exploitée lors de l'analyse d'une phrase. Nous avons voulu que ce système soit suffisamment souple pour permettre, d'une part, de traiter des "phrases" dans un dialogue oral. Cela sous-entend que certains mots ne sont pas toujours correctement entendus ou sont même parfois absents. D'autre part, notre analyse ne se base pas sur un analyseur particulier mais a voulu au contraire, permettre tout type d'analyse à partir de la représentation considérée (analyseur ascendant ou descendant, droite-gauche ou gauche-droite, mixte, partant d'un type de mot particulier, ...).

Il est à noter que nous n'avons pas implémenté l'alternative, jugeant que, même s'il est parfois plus facile d'utiliser ce mécanisme, nous pouvions toujours l'éviter en dédoublant la structure de traits. Si cela parait trop rébarbatif, il est toujours possible de concevoir facilement un système utilisant un précompilateur qui transformera chaque alternative en deux nouvelles structures.

3.2.2. Extension des structures pour l'analyse syntaxique

Dans cette partie, nous définissons les extensions nécessaires à l'exploitation du mécanisme de l'unification pour être utilisé dans un environnement réel d'analyse. Nous avons le mécanisme principal; adjoignons-lui les outils nécessaires à son exploitation dans le cadre du dialogue oral homme-machine. Pour ce faire, nous allons concevoir une structure globale dans laquelle nous trouverons différents champs. Cette approche à été choisie, car elle permet de

séparer parfaitement les informations qui ont un rapport avec l'implémentation même de l'analyseur et celles qui sont de type linguistique. La séparation des informations s'accompagne d'une séparation des traitements et donc d'une plus grande modularité. Ce point est important dans la mesure où nous désirons concevoir les outils pour différents types d'analyseur et pas pour un analyseur bien particulier et parfaitement défini.

- code
- type
- prec
- prec_i
- inc
- inc_i
- struc
- env
- op_eff
- satisfait
- syntaxe

1) le champ syntaxe :

Ce champ est le champ le plus important du point de vue linguistique. C'est lui qui contient toutes les informations tant syntaxique que sémantique. Ce champ se nomme syntaxe, car au départ, il ne devait contenir que des informations syntaxiques. Mais comme nous l'avons vu au chapitre 1, il est tout a fait inutile, voir même impossible de séparer les deux aspects. C'est dans ce champ et seulement dans celui-ci que s'effectue l'unification.

2) le champ code :

Il nous est apparu nécessaire d'inclure un champ permettant d'identifier chaque structure. Cet identifiant est un nombre qui est généré lors de la création de la structure. Cela peut être utile, notamment dans le cas où des recherches dans un espace d'état sont effectuées ou pour déterminer plus facilement l'égalité de deux structures, notamment dans le cas où l'on veut vérifier qu'une opération n'a pas encore été effectuée avec deux structures déterminées. Cela évite de recommencer la même opération plusieurs fois (Cf. champ *op_eff*).

3) le champ type :

Ce champ sert à identifier le champ syntaxe des structures. Cet identifiant doit être donné lors de la création de toute règle de grammaire ou tout élément du lexique. Ce champ est nécessaire pour retrouver une structure d'une classe particulière et pour avoir une probabilité plus élevée qu'elle s'unifie ensuite avec une autre.

4) le champ prec :

Ce champ contient la liste des structures qui suivent l'élément courant. Ces structures peuvent être indifféremment la structure d'un mot ou la structure d'un syntagme. Cette liste est initialisée lors de l'analyse lexicale (Cf. infra).

5) le champ prec_i :

Ce champ est la réciproque du champ précédent. Il contient, en effet, les structures qui précédent l'élément courant.

6) le champ inc :

Ce champ contient la liste des structures où est incluse la structure courante. Lors de l'analyse morphologique, ce champ est évidemment vide. A la fin d'une analyse, ce champ contient toutes les structures qui sont intervenues directement.

7) le champ inc_i :

Inverse du précédent.

8) le champ struc :

Ce champ détermine les éléments grammaticaux composant le champ syntaxe de la structure considérée. En fait, il est constitué d'étiquettes qui font références à ses composants. Les éléments constituant un groupe nominal sont par exemple, un article, un nom et un adjectif. Lors de l'analyse, l'étiquette qui a été unifiée (Cf. fonction *assemble*) est remplacée par le couple formé des deux structures qui ont été utilisées pour créer la structure considérée. Cela permet d'avoir une trace du processus d'analyse et donc de retrouver les éléments constitutifs. Ce champ peut être également utile pour déterminer l'ordre des éléments d'une structure, mais ce n'est pas obligatoire. Il est bien évident que ce champ est vide pour les mots, vu qu'ils n'ont pas d'élément constitutif.

9) le champ env :

Il détermine les types de structures qui peuvent inclure celle que l'on analyse. C'est donc un peu l'inverse du champ précédent. Cela peut être très utile lorsque l'on fait une recherche ascendante.

10) le champ op_eff :

Il contient la liste des opérations déjà effectuées sur cette structure. Cela permet de ne pas effectuer plusieurs fois la même opération sur les même objets.

11) le champ satisfait :

Ce champ est mis à *oui* si tous les éléments du champ *struc* ont été instanciés. Pour les mots, c'est la fonction *analyse lexicale* qui réalise cette instanciation.

Comme nous voulons travailler avec la parole, il se peut que l'on ne trouve pas d'instanciation complète pour le champ *struc*. Cela peut être dû à un décodage acoustico-phonétique qui n'a pas su reconnaître tous les mots de la phrase. Il suffit alors de concevoir un analyseur qui permet de laisser des étiquettes non instanciées. Dans notre approche, le champ *satisfait* serait mis à non. Mais, on pourrait imaginer une représentation dans lequel le champ *satisfait* prendrait des valeurs comprises en 0 et 1 et qui correspondrait à un taux de remplissage du champ *struc*. Une stratégie d'analyse serait alors de privilégier les structures qui auraient ce taux égal à 1 ou le plus proche possible.

3.2.3. Fonctions principales pouvant intervenir dans un analyseur

Nous reprenons ici, les spécifications de trois fonctions que nous avons implémentées (Cf. annexes) et qui nous semblaient essentielles dans les analyseurs se basant sur les grammaires d'unification. Ces fonctions constituent la base

nécessaires à tout analyseur utilisant l'unification. Des fonctions auxiliaires ont également été implémentées et sont décrites dans les annexes. Il y a tout d'abord la fonction effectuant l'analyse lexicale qui nous permet de récupérer les données des mots de la phrase dans le dictionnaire. Ensuite, la fonction *fusion* effectue l'unification de deux structures de même catégorie et finalement la fonction *assemble* qui unifie deux structures de catégories différentes.

Remarque préliminaire : dans les fonctions que nous allons examiner, nous avons utilisé des fonctions de Laurent Romary qui permettent de faire une gestion temporelle des éléments de la phrase à analyser. Ces fonctions vérifient la précédence et l'inclusion des différentes structures de traits (Cf. Romary 89).

1) Analyse lexicale

Cette fonction reçoit en entrée une liste comprenant les mots de la phrase à analyser. A partir de cette liste, elle fait une recherche dans le dictionnaire pour retrouver toutes les instanciations de chaque mot. Elle retourne comme résultat une liste de structures où sont initialisés les champs *codes*, *prec-i* et *prec*. Les mots repris dans le dictionnaire plusieurs fois, mais avec des caractéristiques différentes, sont évidement repris chaque fois. Finalement, la liste renvoyée est un treillis de structures instanciées. Cela vient du fait que nous initialisons les champs *prec* et *prec_i* dans cette fonction. Les champs *inc* et *inc_i* sont, bien entendu, laissés à vide. Il est bien évident aussi que, seuls les mots répertoriés dans le dictionnaire sont pris en compte lors de cette analyse et donc, que les mots inconnus ou mal orthographiés sont ignorés. Cette fonction est le point de départ de tout analyse vu qu'elle construit la représentation interne de la phrase ainsi que ses aspects temporels.

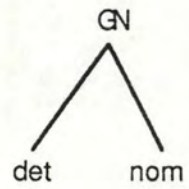
2) Assemble

Cette fonction reçoit trois éléments : une première structure dite supérieure, une deuxième, dite inférieure et enfin une étiquette. Cette étiquette correspond à l'endroit, dans le champ syntaxe de la structure supérieure où se fera l'unification avec la totalité du champ syntaxe de la structure inférieure. Cette fonction greffe une petite structure sur la grande au niveau de l'étiquette. Elle réunit donc deux structures de catégories différentes.

Dans l'exemple de la page suivante, nous assemblons le déterminant *"le"* (n° de code = 24) avec un groupe nominal (n° de code = 12) composé d'un déterminant et d'un nom. La greffe se faisant au niveau de l'étiquette 41. Nous pouvons remarquer que la structure résultat, à savoir la structure ayant le numéro de code 76, a le même type que la 12. L'étiquette 41 du champ *struc* a été remplacée par le couple (12 . 24) ce qui est bien le numéro des structures composantes. Le champ *satisfait* est toujours à NIL dans la mesure où le champ *struc* n'est pas encore totalement instancié. Nous pouvons observer également que l'unification s'est faite au niveau du déterminant et que le mécanisme du partage de structures a bien fonctionné dans la mesure où le genre et le nombre du groupe nominal correspondent bien au genre et au nombre du déterminant. Le deuxième exemple s'effectue de manière similaire avec un nom commun.

Première structure générique :

```
code : 12
type : GN1
struc : (@ 41 @ 42)
env : (PHR CCL GV)
satisfait : NIL
syntaxe :
 (
 CAT GN
 GENRE @ 43  NIL
 NOMBRE @ 44  NIL
 DETER @ 41 <- (
        CAT DET
        GENRE @ 43  NIL
        NOMBRE @ 44  NIL
              )
 NOM_PRINC @ 42 <- (
           CAT NOM
           GENRE @ 43  NIL
           NOMBRE @ 44  NIL
                 )
 )
```



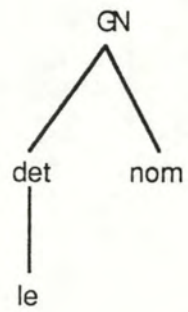
Premier "mot" : le

```
code : 24
type : det1
struc : NIL
env : (GN)
satisfait : OUI
syntaxe :
 (
 CAT DET
 GENRE MASCULIN
 NOMBRE SINGULIER
 LEX LE
 )
```

Etiquette considéréé pour le premier assemblage : 41

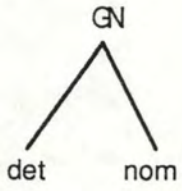
Résultat du premier assemblage :

```
code : 13
type : GN1
struc : ((12 . 24) @ 42)
env : (PHR CCL GV)
satisfait : NIL
syntaxe :
 (
 CAT GN
 GENRE @ 43  <- MASCULIN
 NOMBRE @ 44  <- SINGULIER
 DETER @ 41 <- (
        CAT DET
        GENRE @ 43
        NOMBRE @ 44
        LEX LE
        )
 NOM_PRINC @ 42 <- (
           CAT NOM
           GENRE @ 43
           NOMBRE @ 44
           )
)
```



Deuxième structure générique :

```
code : 14
type : GN1
struc : (@ 56 @ 57)
env : (PHR CCL GV)
satisfait : NIL
syntaxe :
 (
 CAT GN
 GENRE @ 58  NIL
 NOMBRE @ 59  NIL
 DETER @ 56 <- (
        CAT DET
        GENRE @ 58
        NOMBRE @ 59
        )
 NOM_PRINC @ 57 <- (
           CAT NOM
           GENRE @ 58
           NOMBRE @ 59
           )
 )
```



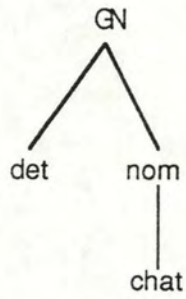
Deuxième "mot" : chat

```
code : 27
type : nom1
struc : NIL
env : (GN)
satisfait : OUI
syntaxe :
 (
 CAT NOM
 GENRE MASCULIN
 NOMBRE SINGULIER
 LEX CHAT
 )
```

Etiquette considérée pour le deuxième assemblage : 57

Résultat du deuxème assemblage :

```
code : 15
type : GN1
struc : (@ 56 (14 . 27))
env : (PHR CCL GV)
satisfait : NIL
syntaxe :
 (
 CAT GN
 GENRE @ 58 <-  MASCULIN
 NOMBRE @ 59  <- SINGULIER
 DETER @ 56 <- (
        CAT DET
        GENRE @ 58
        NOMBRE @ 59
        )
 NOM_PRINC @ 57 <- (
           CAT NOM
           GENRE @ 58
           NOMBRE @ 59
           LEX CHAT
           )
)
```



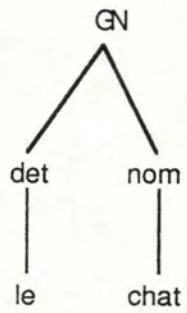
3) Fusion

Cette fonction reçoit deux structures et va faire l'unification de leur champ *syntaxe* dans leur totalité. Il faut, pour que cela réussisse, que ces deux structures aient au moins la même catégorie et il est souhaitable qu'elles aient aussi le même type. Dans le cas où la dernière condition ne serait pas vérifiée, il y aurait alors création d'une nouveau type de structure, ce qui n'est pas toujours souhaitable. En effet, cette nouvelle structure pourrait n'être qu'une copie d'une autre. Nous obtiendrions donc, à la fin de l'analyse, deux résultats qui auraient la même représentation alors qu'il n'y aurait aucune ambiguïté.

Si nous prenons comme exemple le résultat des "assemblages" précédents (Cf. page suivante), que nous les "fusionnons", nous obtenons la structure n° 16 où nous pouvons observer que le champ *struc* est totalement instancié, que le champ *satisfait* est donc mis à *oui* et que le champ *syntaxe* est parfaitement instancié.

Résultat de la fusion entre les deux résultats précédents :

```
code : 16
type : GN1
struc : ((12 . 24) (14 . 27))
env : (PHR CCL GV)
satisfait : OUI
syntaxe :
 (
 CAT GN
 GENRE @ 43  <- MASCULIN
 NOMBRE @ 44 <- SINGULIER
 DETER @ 41 <- (
        CAT DET
        GENRE @ 43
        NOMBRE @ 44
        LEX LE
        )
 NOM_PRINC @ 42 <- (
           CAT NOM
           GENRE @ 43
           NOMBRE @ 44
           LEX CHAT
           )
)
```



3.3. Formalismes linguistiques utilisant les grammaires d'unification

Nous allons introduire deux formalismes linguistiques qui vont nous permettre de découvrir certains problèmes résolus facilement par l'homme mais pas par un ordinateur. Nous avons choisi ces formalismes dans la mesure où ils sont souvent repris dans la littérature et qu'il utilisent les grammaires d'unification. Il s'agit des systèmes qui représentent l'information tant syntaxique que sémantique, mais qui ne constituent en aucune façon un analyseur, ce n'est pas leur but.

3.3.1. G.P.S.G.

La Grammaire syntagmatique généralisée ou Generalized Phrase Structure Grammar (G.P.S.G.) a été élaborée à partir de la fin des années 70 par Gérald Gazdar et a surtout été mise au point dans les années 80 (Gazdar 85). Elle fait partie des formalismes grammaticaux qui utilisent les grammaires d'unification et qui sont les plus utilisées actuellement. Elle est classée dans les grammaires non contextuelles mais parvient malgré tout à rendre compte des dépendances non locales.

Bien que le lexique fut un point important pour les créateurs de cette théorie, c'est l'aspect syntaxique qui a été le plus développé. Il existe des règles sémantiques qui sont associées aux règles syntaxiques et qui construisent la représentation logique de la phrase. La théorie sémantique utilisée se rapproche de la théorie sémantique formelle de Montague (Montague 74). Les règles sémantiques permettent notamment de lever des ambiguïtés relatives à une même construction syntaxique (ex. *Jean est habile à convaincre* et *Jean est facile à convaincre*), le traitement des pronoms et de

la quantification à l'aide de variables logiques, la représentation de la même structure sémantique alors que la syntaxe est différente,...

L'aspect lexical se limite, lui aussi, à l'interface entre la syntaxe et la sémantique. Une entrée lexicale est représentée de la façon suivante : représentation phonétique, l'ensemble des traits syntaxiques, l'information morphologique et la forme sémantique. Exemple :

</marʃe/, {<N,->,<V,+>,<SOUSCAT,4>}, régulier du 1er groupe, **marcher**'>

La représentation phonétique est simplement la transcription du mot dans l'alphabet phonétique (ici : marche), l'information morphologique donne des renseignements sur la formation du mot, la forme sémantique donne le mot duquel on a tiré le mot concerné. Les traits syntaxiques sont sous la forme de <nom de trait, valeur de trait>, c'est-à-dire des traits dans notre modèle ou encore des catégories en G.P.S.G.. Le trait <N,-> nous indique que nous n'avons pas à faire à un groupe nominal mais à un groupe verbal (<V,+>). Le trait <SOUSCAT,4> indique que le mot fait partie de la sous-catégorie 4 qui donne en fait le type du verbe : transitif, intransitif ou bitransitif. Le fait d'être dans la sous-catégorie 4 signifie que le verbe marcher est intransitif.

Il existe deux sortes de règles syntaxiques : les règles de dominance immédiate (DI) et les règles d'ordre linéaire (OL) ou précédence linéaire (PL). Les règles de dominance immédiate expriment la relation de dominance qui lie un syntagme et ses constituants immédiats sans fixer leur ordre. En fait, ce sont des règles de réécriture. On aura donc, par exemple SV --> V, SN qui signifie qu'un syntagme verbal se compose d'un verbe et d'un syntagme nominal. Ce sont les règles d'ordre linéaire qui explicitent les ordres acceptables

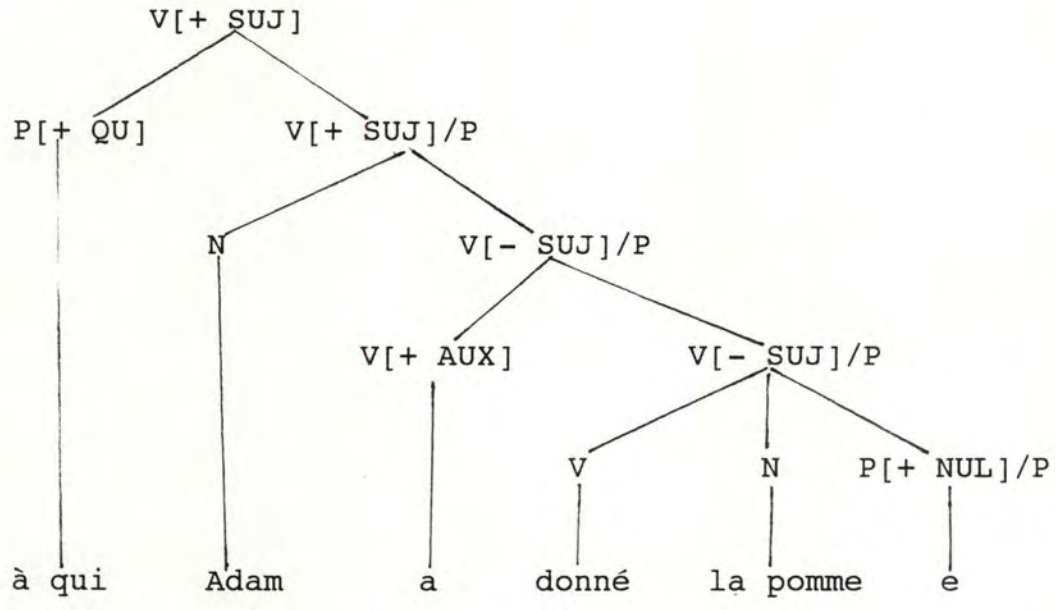
entre les différentes catégories grammaticales.

Dans les DI, il existe une distinction entre les DI lexicales et les DI non-lexicales. Les règles lexicales sont celles qui introduisent une tête lexicale, c'est-à-dire que l'élément qui reflète le statut syntaxique (par exemple le verbe pour un syntagme verbal) qui appartient à la classe lexicale.

G.P.S.G. a introduit la notion de *métarègles* : ces règles ne peuvent s'appliquer que sur des DI lexicales et leur but est de produire de nouvelles règles. Elles expriment des généralisations d'un niveau supérieur à celui des règles et n'apportent rien de plus à la puissance du formalisme, mais permettent de le simplifier à certains moments.

Détaillons à présent les traits possibles en G.P.S.G.. Les traits à valeur atomique sont les traits classiques que l'on retrouve dans la plupart des grammaires : N, V, PLU, MASC. Nous avons vu précédemment la signification de SOUSCAT, mais il existe aussi le trait BARRE qui indique le niveau dans la hiérarchie syntaxique. <PFORME, à> oblige un syntagme prépositionnel à commencer par la préposition *à*.

Examinons à présent quatre traits à valeurs catégorielles qui permettent de traiter les dépendances non locales : RE sert à encoder la relation entre un élément réfléchi ou réciproque et son antécédent; ACCORD spécifie la relation entre un élément et l'élément avec lequel il s'accorde; QU et SLASH (/) caractérisent la relation entre une position vide et un élément antéposé. Exemple : pour le traitement de la phrase *A qui Adam a-t-il donné la pomme?*



Le [+ SUJ] signifie qu'il existe un sujet; V[+ SUJ]/P indique que le syntagme V ne comprend pas le syntagme prépositionnel et P[+ NUL] exprime le fait que P est vide.

Comme nous pouvons le constater, G.P.S.G. est donc une théorie que tente de prendre en compte la plupart des aspects de la langue.

3.3.2. H.P.S.G.

Head-driven Phrase Structure Grammar (Grammaire syntagmatique guidée par la tête) a été développée par Carl Pollard et Ivan Sag dans leur ouvrage "Information-Based Syntax and Semantics" (Pollard 87). Cette théorie était fortement inspirée de G.P.S.G. car ses auteurs ont travaillé dans l'équipe de Gazdar. Depuis 1987, ses auteurs ont modifié ce formalisme régulièrement pour y introduire le traitement de plus de faits linguistiques. Leur but est de concevoir une théorie qui intègre la syntaxe et la sémantique du langage naturel.

Dans (Pollard 91), un objet linguistique est constitué de trois types d'information : phonologique (*phon*), syntaxico-sémantique (*synsem*) et quantitatif (*qstore*). La combinaison de ces trois éléments forme un "signe". Actuellement, l'aspect phonologique est peu développé, il s'agit simplement d'une chaîne de caractères ou d'une liste de phonèmes. Dans le premier volume, l'aspect syntaxico-sémantique était considéré dans ses deux composantes séparément. Le trait *qstore* (*quantifier store*) n'était pas représenté directement mais était plus ou moins inclus dans le trait syntaxique.

Le trait *synsem* se divise en deux structures : *local* et *nonlocal* (anciennement *bind*). L'attribut *local* concerne toutes les informations relatives aux différents aspects propres au signe tandis que l'attribut *nonlocal* concerne les aspects qui lui sont extérieurs. C'est notamment le cas pour les clauses interrogatives ou relatives. Ce trait *nonlocal* se divise en trois attributs possibles : *slash* pour les trous ou traces, *rel* pour les relatives et *que* pour les pronoms interrogatifs. L'attribut *local* se divise également en trois structures : *category*, *content* et *context*.

La catégorie représente plus que la simple catégorie grammaticale, elle contient d'autres arguments grammaticaux. Elle est composée de deux attributs : *head* et *subcat*. *Head* prend deux sortes de valeur : *substantive* (nom, verbe, adjectif et préposition) et *fonctionnelle* (déterminant). *Subcat* est une liste des signes qui doivent être ajoutés pour former une phrase ou un syntagme complet, c'est-à-dire la liste des autres signes pour que le signe en question soit "saturé". Exemple : pour le verbe *to give* conjugué à la troisième personne de l'indicatif présent, nous aurons la représentation suivante :

$$\begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD verb [fin]} \\ \text{SUBCAT} < \text{NP [nom]}_{\text{[3rd, sing]}}, \text{ NP [acc]}, \text{ NP [acc]} > \end{bmatrix} \end{bmatrix}$$

L'aspect sémantique est représenté dans le trait *content* et *context*. Le trait *context* contient des informations qui relèvent de suppositions. Par exemple, le pronom *she*, en anglais ne peut se référer qu'à une femme, alors que le pronom *elle* du français peut se référer à une chose. Le trait *content* est composé de traits reprenant le prédicat et les arguments sémantiques tels qu'on peut en trouver dans les grammaires casuelles.

H.P.S.G. utilise également la notion d'étiquette pour faire de la vérification sémantique ou syntaxique mais également pour traiter la notion de *dépendance à longue distance* comme c'est le cas pour les phrases interrogatives ou pour les relatives. Contrairement à notre notation, ils indiquent les étiquettes en utilisant des carrés dans lequel se trouve le numéro et l'inscrive après la structure plutôt qu'avant.

Si nous reprenons ces trois éléments, nous pouvons compléter notre exemple du verbe *to give* :

$$\begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD verb [fin]} \\ \text{SUBCAT} < \text{NP [nom]}_{[1]}\ [3rd, sing],\ \text{NP [acc]}_{[2]},\ \text{NP [acc]}_{[3]} > \end{bmatrix} \\ \text{CONTENT} & \begin{bmatrix} \text{REL} & \text{give} \\ \text{AGENT} & [1] \\ \text{GOAL} & [2] \\ \text{THEME} & [3] \end{bmatrix} \end{bmatrix}$$

Cette théorie est, comme son nom l'indique "dirigée par la tête", c'est-à-dire qu'elle considère certains éléments comme suffisamment importants soit au niveau de la syntaxe (souvent le verbe ou le nom), soit au niveau de la sémantique,

pour qu'ils puissent représenter l'ensemble de l'aspect syntaxique ou sémantique de la phrase. La "tête" est le constituant principal de la phrase et c'est donc sur cet élément qu'il faut porter son attention.

3.3.3. Conclusion

Ces deux théories pourraient très bien s'insérer dans notre modèle défini en 3.2.. En effet, elles n'utilisent pas de nouveaux outils de traitement. H.P.S.G est sans doute mieux adaptés pour le type d'analyseur que nous avons envisagé dans la mesure où il ne demande pas de traitement spécial pour découvrir la "tête" comme c'est le cas en H.P.S.G.

3.4. Un système utilisant les grammaires d'unification : CAT2

Examinons à présent un exemple réel de système utilisant un formalisme basé sur les grammaires d'unification. CAT2 est un système développé à Saarbrücken dans le cadre du programme européen Eurotra (Cf. Mesli 91, Sharp 88 et Sharp 90). Ce système est destiné à faire de la traduction automatique pour les neuf langues officielles de la Communauté Economique Européenne (français, italien, espagnol, portugais, anglais, allemand, néerlandais, danois et grec).

Actuellement, il est implémenté pour traduire des textes entre le français, l'anglais et l'allemand et considère le texte comme une suite de phrase qu'il traduit une à une plutôt qu'un texte global.

Le mode de traduction utilisé est le système à transfert, c'est-à-dire qu'il existe un module de transfert qui transforme une représentation de la phrase source en une représentation de la phrase cible sans passer par une représentation intermédiaire qui serait partagée par les deux langages. Cela oblige les concepteurs à écrire 72 modules de transfert pour traiter les neuf langues européennes.

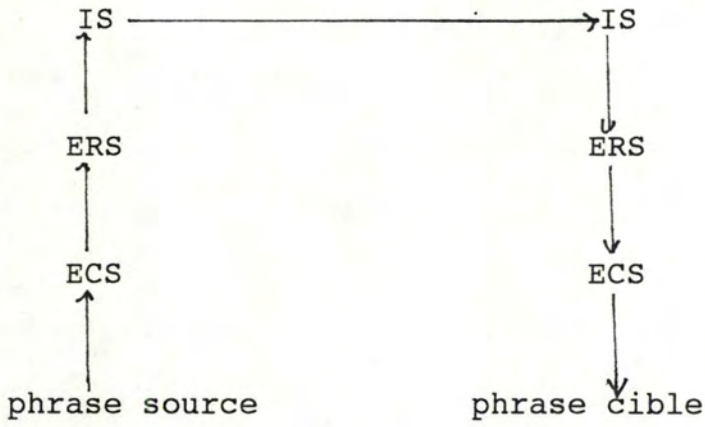
La traduction d'une phrase se passe de la façon suivante: la phrase source est transformée par le module d'analyse en un arbre (ou plusieurs en cas d'ambiguïté) la représentant dans la langue source. Cet arbre est transformé par le module de transfert en un arbre de la langue cible. Le module de génération le travaille alors pour construire une phrase de la langue cible.

Le module de génération est le symétrique du module d'analyse qui fonctionne de la façon suivante : (Mesli 91)

- "le niveau 1 ou Niveau "ECS" (*EUROTRA Configurational Structure*) représente l'analyse *syntaxique*. Les mots de la phrase reçoivent la (ou leur) catégorie de discours (nom, verbe, déterminant, adjectif ...) et sont regroupés en constituants (groupe nominal, verbal, prépositionnel ...).

- Le niveau 2 ou niveau "ERS" (*EUROTRA Relational Structure*) représente l'analyse *relationnelle*. Les constituants ECS sont réorganisés selon leur fonction syntaxique : verbe, sujet, complément, modifieur.

- Le niveau 3 ou niveau "IS" (*Interface Structure*) représente l'analyse *sémantique*. Les fonctions ERS sont réorganisées ici selon leur rôle sémantique : prédicat (appelé "gouverneur"), argument ou modifieur."



Ce système a été construit en C-Prolog de Sicstus, car ce langage permet de construire facilement des maquettes et est plus facile a utilisé par des linguistes que le Lisp par exemple.

Chapitre 4. Autres utilisations des grammaires d'unification

Dans ce chapitre, nous allons étudier les autres possibilités apportées par les grammaires d'unification. Dans un premier temps, nous allons voir les possibilités offertes à un niveau inférieur à l'analyse syntaxique, c'est-à-dire au niveau du décodage acoustico-phonétique principalement. Dans un deuxième temps, nous analyserons les niveaux supérieurs en examinant deux théories qui ont dominé leur époque, à savoir les grammaires casuelles pour les années 70 et la DRT pour les années 80. Nous conclurons enfin en examinant les avantages et inconvénients d'une représentation intégrée.

4.1. A un niveau inférieur à l'analyse syntaxique

La notion de trait peut être utilisée au niveau de la phonétique. En fait, c'est de ce niveau qu'elle provient. Pour caractériser les consonnes et les voyelles. On peut ainsi distinguer les voyelles labialisées avec un + ouvert, les consonnes sourdes (p, t, k) avec un - voisée, les sonores (b, d, g) avec un + voisée et ainsi de suite. Ces caractéristiques permettent ensuite de représenter les sons à l'intérieur de la bouche.

4.2. A un niveau supérieur à l'analyse syntaxique

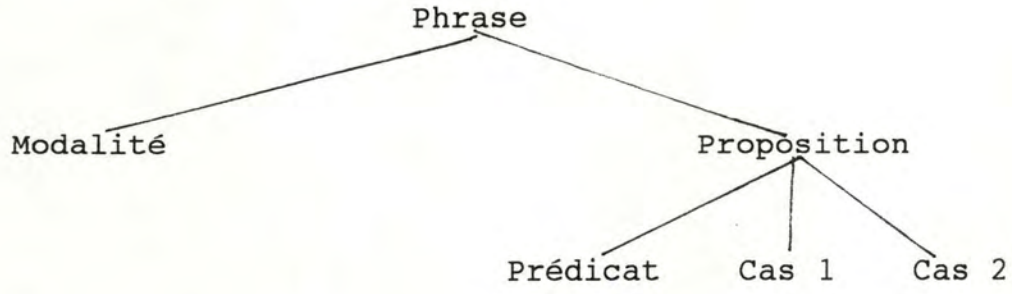
Dans notre approche, nous nous sommes surtout intéressés à l'aspect syntaxique. Nous allons voir avec les grammaires de cas, qu'il est possible d'ajoindre des traits pour rendre notre formalisme beaucoup plus sémantique. Dans un deuxième temps, nous verrons qu'avec la DRT, il est possible de concevoir une autre représentation à partir des structures de traits. Cette autre représentation permet de gérer un discours dans son ensemble et non plus une phrase isolée.

4.2.1. Les grammaires de cas

Les grammaires de cas ou casuelles datent de la fin des années 60. Elles ont en effet été élaborées par Fillmore en 1968 dans son célèbre article "The case for the case" bien qu'elles aient été suggérées dans l'article de 1966 "A proposal concerning English propositions". Nous allons l'étudier dans un premier temps pour voir ensuite les possibilités de son intégration dans un analyseur utilisant les grammaires d'unification. Cet analyseur deviendrait alors syntaxico-sémantique.

Fillmore a constaté que les relations grammaticales ne définissaient pas toujours des fonctions sémantiques et que plusieurs phrases ayant une structure de surface différente peuvent avoir la même signification. C'est notamment le cas pour les phrases à la voie active et à la voie passive. C'est la raison pour laquelle il va s'attacher à la structure profonde plutôt qu'à la structure de surface, aux aspects sémantiques plutôt qu'aux aspects syntaxiques.

Une phrase est représentée par une modalité et une proposition. La modalité concerne tout ce qui a rapport au temps grammatical (présent, imparfait, futur), au mode et à l'aspect (la forme de la phrase : affirmative ou négative; le type : interrogatif ou exclamatif; la voie : active ou passive, ...). La proposition est une structure indépendante de la structure de surface et est constituée du prédicat et des différents cas. Il est à noter que la fonction de prédicat ne se limite pas seulement au verbe (bien que cela soit souvent le cas) mais peut également être remplie par un adjectif ou un substantif.



La détermination des différents cas est ce qui pose le plus de problèmes à cette théorie. Nous reprenons ici ceux indiqués par J. M. Pierrel dans son cours de DEA :

- AGENT : l'instigateur animé d'une action

- CONTRE-AGENT : la force contre laquelle l'action est exécutée

- OBJET : l'entité qui bouge, change ou dont la position ou l'existence est en question

- RESULTAT : l'entité créée

- SOURCE : lieu de départ de quelque chose qui bouge

- BUT : lieu d'arrivée de quelque chose qui bouge

- PATIENT : l'entité qui reçoit accepte ou subit

- DATIF : l'animé affecté par l'action

- FACTITIF : l'objet résultant de l'action

- LIEU : le lieu.

Exemple tiré de (Sabah 88) : *Jean a conduit son bateau, malgré le courant, du large vers la plage*

Jean = Agent; *bateau* = objet; *courant* = Contre-agent;

large = Source; *plage* = But.

A partir d'un tel résultat, le module de dialogue peut alors mettre à jour sa base de connaissances et faire des inférences assez facilement (Cf. mémoire de Pascal Druart).

Comme nous l'avons signalé précédemment, le principal problème de cette théorie est le choix des cas. En effet, chaque utilisateur de cette théorie trouve des cas différents et un même auteur peut déterminer une série de cas qui seront valables pour une application bien déterminée, mais ne le seront plus pour une autre.

De même, dans la théorie de Fillmore, un cas ne peut avoir qu'une seule réalisation possible pour une phrase déterminée, alors que ce n'est pas toujours le cas chez certains auteurs.

Les analyseurs basés sur les grammaires casuelles devraient effectuer une analyse sémantique sans faire appel à la syntaxe. Mais, pratiquement, cela très pas très pratique, c'est la raison pour laquelle, dans le système DIAL du CRIN, ils utilisent les R.N.P. pour déterminer les classes syntaxiques. Ensuite, ils recherchent le verbe de la phrase et essaient de déterminer le prédicat correspondant et sa structure. Pour cela, on détermine si le prédicat est dynamique ou/et contrôlé et donc, s'il s'agit d'un acte, d'un processus ou d'un état. En fonction de ces résultats, on active des primitives qui déterminent les cas grâce à la recherche par mots clés (Cf. Deville 89).

Un des avantages de ce type de grammaire est de pouvoir traiter des phrases non normées qui ne nécessitent donc pas

une étude exhaustive des constructions syntaxiques possibles. Par contre, un des désavantages est de ne pas pouvoir traiter des phrases où il n'y a pas de verbe ou encore celles qui sont enchâssées. En effet, seul le verbe principal est pris en compte alors que l'information peut se trouver dans une relative.

Il est facile d'intégrer les grammaires de cas dans les grammaires d'unification. En effet, il suffit de prévoir des traits qui vont, d'une part effectuer des vérifications pour déterminer le type casuel qui correspond à la possession ou non de certaines valeurs. D'autre part, grâce au mécanisme des structures partagées, il est facile de remonter l'information et de la regrouper à un niveau supérieur. C'est un peu se qui se fait dans le trait *content* de H.G.S.G.

De plus amples détails sur les grammaires de cas sont donnés dans la Thèse de Doctorat de G. Deville ou encore dans les travaux de P. Godin sur "Les aspects syntaxiques et sémantiques de la grammaire casuelle appliquée au français" (Godin 75).

4.2.2. La DRT

La DRT pour Discourse Representation Theory a été développée au début des années 80 par Hans Kamp. Son but était de concevoir un système capable de représenter la syntaxe et la sémantique d'une phrase et de pouvoir traiter notamment les problèmes de références. En fait, il traite le discours dans son entièreté et prend donc en compte les phrases précédemment prononcées. En effet, si l'on veut analyser un discours cohérent, il est nécessaire de le prendre dans son ensemble plutôt que d'isoler chaque phrase. Cela permet de mieux situer

le contexte dans lequel est situé le discours et donc de lever certaines ambiguïtés. Sa théorie a été appliquée à la langue anglaise.

La syntaxe de base est inspirée de celle de GPSG (Cf. 3.3.1). Il reprend notamment les catégories syntaxiques classiques suivantes : *S* (phrase), *NP* (syntagme nominal), *VP* (syntagme verbal), *PN* (nom propre), *PRO* (pronom), *DET* (déterminant), *N* (nom). Il utilise les traits *Num* (nombre) avec comme valeur *sing* (singulier) ou *plur* (pluriel), *Case* avec les valeurs *+nom* (nominatif) ou *-nom* (non-nominatif) suivant que l'on a à faire à un syntagme nominal sujet ou non (cela intervient surtout pour les pronoms : *he, she, they* seront nominatifs alors que *him, her, them* ne le seront pas, *it* pouvant prendre les deux valeurs), *Gen* (genre) aura trois valeurs possibles : *male* (masculin), *fem* (féminin) et *-hum* (non humain). Il fait également la distinction entre les verbes transitifs ou intransitifs au moyen du trait *Trans* qui peut prendre comme valeur + ou -. Il y a aussi les verbes ditransitifs (qui ont deux objets) et les verbes qui demandent une préposition (ex. : *to rely*) mais il n'en tient pas compte car son but n'est pas de construire une grammaire qui recouvrirait toutes les phrases possibles mais de rendre compte des phénomènes les plus importants du langage. Il introduit aussi le trait *Fin* pour désigner les verbes qui sont (valeur -) ou qui ne sont pas (valeur +) à l'infinitif.

Enfin, il traite les clauses relatives grâce au trait *gap*. Dans les phrases avec un verbe transitif, il y a un objet direct qui suit le verbe. Or, dans les clauses relatives, le rôle de l'objet direct est joué par un pronom relatif et la place derrière le verbe est inoccupée; il y a un trou. Cela introduit deux nouvelles catégories syntaxiques : celle des clauses relatives RC et celle des pronoms relatifs RPRO.

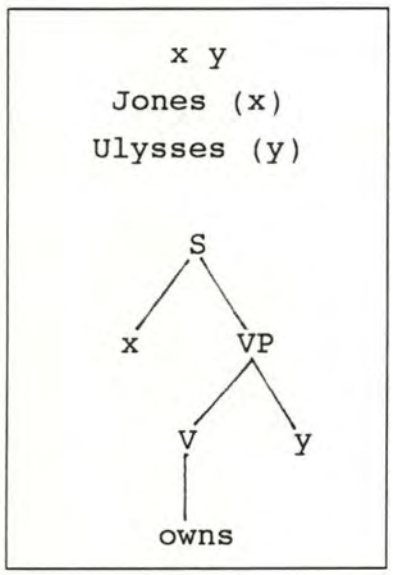
Kamp introduit aussi la notion de valeur par défaut,

c'est-à-dire qu'un trait aura une certaine valeur si elle n'est pas définie explicitement. Pour pouvoir faire de la vérification de règles, il utilise des variables qu'il nomme avec des lettres grecques. Le fait de retrouver deux variables entre des accolades (ex : {α,ß}) comme valeur d'un trait signifie qu'elles doivent avoir la même valeur et que le trait aura cette valeur.

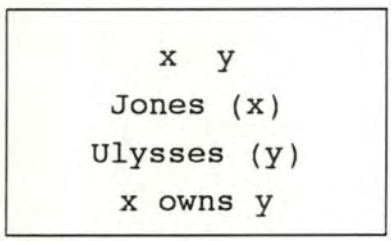
Après avoir vu la syntaxe, examinons la théorie proprement dite. Comme nous l'avons indiqué, le but de la DRT est de traiter le discours dans son ensemble. Pour cela, il analyse chaque phrase l'une à la suite de l'autre pour transformer la "Structure de Représentation du Discours" (Discourse Representation Structures ou DRS). "Pour comprendre quelle information est ajoutée par la phrase suivante d'un discours à ce qu'il a déjà appris dans les phrases la précédent, l'interpréteur doit rattacher cette phrase à l'information antérieure. Donc, l'interprétation d'une nouvelle phrase doit se faire sur deux types de structures, la structure syntaxique de la phrase elle-même et la structure représentant le contexte des phrases précédentes."(Kamp 90). Pour illustrer ce phénomène, examinons le problème de l'anaphore pronominale. Prenons la phrase :

"*Jones owns Ulysses. It fascinates him.*"

On ne peut comprendre la deuxième qu'en ayant la première et en sachant que Ulysses est un livre. Pour résoudre ce problème, construisons tout d'abord la représentation de la première phrase en spécifiant, grâce au lexique, que Jones est un nom propre masculin, que Ulysses est un nom propre non animé et aussi le fait que Jones possède le livre Ulysses. Cela sera représenté en DRS de la façon suivante : tout d'abord, la représentation de l'univers :



et un ensemble de conditions DRS :



En fait, ces diagrammes sont obtenus après avoir analysé chaque mot. Chacun de ceux-ci déclenchent des règles de construction suivant leur catégorie lexicale. Pour un nom propre par exemple, on aura une règle pour la construction de noms propres (*Construction Rule for Proper Names* ou CR.PN).

A partir de là, on construit une représentation similaire de "*It fascinates him*" grâce à des règles de construction pour pronoms (CR.PRO) qui indiqueront que *It* est non humain et que *him* est du genre masculin. Ensuite, l'interpréteur fait des inférences sur l'univers existant pour retrouver des mots qui ont ces caractéristiques et peut alors ajouter le fait que *Ulysses fascine Jones*.

Une DRS est vraie s'il existe des individus appartenant au cadre de la théorie des modèles du discours qui satisfont les conditions. Une condition DRS réductible est une DRS qui contient une configuration qui peut déclencher des règles de construction. L'algorithme se résume donc en deux récursions : une sur les phrases (les phrases doivent être analysées les unes après les autres) et une sur les conditions DRS sur lesquelles doivent être appliquées les règles de construction tant qu'elles sont réductibles.

Pour bien débuter l'analyse d'un discours, il faudrait que le contexte soit mis sous forme DRS. Or, la DRS ne traite que les évènements qu'au travers de leur représentation syntaxique. Elle ne traite donc que les phrases d'un discours et ne peut prendre en compte le véritable contexte. Elle utilise des informations co-textuelles plutôt que des informations contextuelles.

4.2.3. Conclusion

Aprés l'examen de ces deux théories, nous pouvons conclure que les grammaires d'unification peuvent être utilisées efficacement au niveau sémantique et peuvent conduire à une représentation d'un discours. Cette représentation du discours permet notamment le traitement des anaphores inter-phrastiques alors que la simple analyse syntaxique d'une phrase ne permet que le traitement des anaphores intra-phrastiques.

4.3. Vers une représentation intégrée

La représentation intégrée consiste à utiliser le même formalisme aux différentes étapes de l'analyse d'une phrase. Cela permet d'échanger le même type d'information entre les différents modules et donc d'éviter de devoir créer une interface entre chaque module. De plus, avec cette méthode, il est possible de faire transiter plus facilement de l'information entre deux modules alors que les deux modules n'ont pas encore terminé leur processus d'analyse. On peut donc interrompre un module pour lui demander de l'information. Un système parallèle entre les différents niveaux peut donc être implémenter aisément.

CHAPITRE 5. CONCLUSIONS ET PERSPECTIVES

Après avoir examinané le fonctionnement des grammaires d'unification, nous pouvons conclure qu'elle ont la puissance suffisante pour représenter les informations linguistiques nécessaire au traitement du langage naturel. Notre approche théorique et pratique nous a montré qu'il est tout à fait possible d'utiliser ce formalisme dans un analyseur syntaxico-sémantique.

La qualité principale des grammaires d'unification est leur capacité à être utilisées dans la plupart des niveaux d'un systèmes de dialogue oral homme-machine. La simplicité de ce formalisme lui permet de s'adapter facilement aux problèmes rencontrés et convient donc parfaitement pour un système global de traitement du discours.

Pour compléter notre approche, il faudrait tout d'abord concevoir une grammaire française qui réponde aux spécificités du système DIAL ou Multiworks. L'étude d'une stratégie d'analyse syntaxico-sémantique optimale s'avère également nécessaire. De plus, il faudrait créer un véritable module lexical qui recevrait les données d'un module de décodage acoustico-phonétique du style d'Aphodex, pour pouvoir testé le tout dans un environnement réel.

Néanmoins, l'approche des grammaires d'unification semble être tout à fait adéquate pour le traitement du dialogue homme-machine.

ANNEXES

L'implémentation des fonctions vues en 3.2.3., constituent la base d'un analyseur basé sur les grammaires d'unification. Elles ont été écrites en Common Lisp et sont décrites dans les pages suivantes.

Elles sont présentées avec toutes une séries d'autres fonctions auxiliaires qui sont utilisées par ces fonctions de base ou pour la conception des autres fonctions d'un analyseur.

La fonction *habillage* représente l'implémentation de la fonction d'analyse lexicale. Cette fonction "habille" la phrase pour qu'elle puisse être utilisable par les autres fonctions.

Aug 7 14:00 1991 init.lisp Page 1

```
(in-package "USER")

(defstruct (struc_de_trait (:print-function print_struc))
        type      ; type de la structure de trait
        code      ; identifiant de structure de trait
        struc     ; liste des types d'elements qui peuvent etre contenu
                  ; dans cette structure de trait
        env       ; liste des types de structures de trait pouvant contenir la presente
        prec      ; liste des structures qui suivent la presente
        prec_i    ; liste des structures qui precedent la presente
        inc       ; liste des structures ou est incluse la presente
        inc_i     ; liste des structures qui sont inclusent dans la presente
        op_eff    ; liste des operations deja effectuees sur cette structure, cette liste est
                  ; composee de (l'autre structure, type de l'operation, echec ou reussite, resultat)
        satisfait ; champ determinant si le champ struc est completement instancie
        syntaxe   ; liste de cle-valeur determinant la "syntaxe" de la presente
)
(defvar $compteur 0)

(defvar $profondeur 2)
;;; variable etablissant la condition d'arret lors de l'analyse

(defvar $cond_arret ())

;;; variable de l'espace des contraintes temporelles

(defvar $prec_inc () )

;;; compteur representant le type des structures

(defvar $count_type 0)

;;; compteur de structures

(defvar $count_struc 0)

;;; variable de presentation du champ syntaxe

(defvar $syntaxe_col 'N)

;;; variable definissant la profondeur a laquelle on doit aller lors de la
;;; presentation du champ syntaxe en "colonnes".

(defvar $prof_synt 2)

;;; nbre d'entrees ds ana_chp_struc

(defvar so 0)

;;; numero d'etiquette

(defvar $etiq 0)

;;; espace d'etat

(defvar $ee ())

;;; variable contenant le nom du fichier ou se trouve le lexique et la grammaire

(defvar $fich_lex_gram "lexgramter.lisp")

;;; variable contenant la grammaire et le lexique

(defvar $gram_lex ())

;;; variable contenant les operations (fusion ou assemblage) effectuees
;;; entre structures
;;; type des elements de cette liste : (struc1 struc2 type_d'op res)

;(defvar $op_eff ())

;;; variables pour l'affichage d'une structure
;;; '+ ==> afficher

(defvar $champ_type '+)
(defvar $champ_code '+)
(defvar $champ_struc '+)
(defvar $champ_env '+)
(defvar $champ_satisfait '+)
(defvar $champ_synt '+)
(defvar $champ_op_eff '-)

(load "assemble")
(load "faux")
```

```
(load "traite")
(load "habillage")
(load "unification")
(load "~romary/=maquette/tables.lisp") ; cf ~romary/=maquette
(load "~romary/=maquette/tempsinfer.lisp") ; cf ~romary/=maquette
(load "~romary/=maquette/utils_infer") ; cf ~romary/=maquette
(load "var")

(load "fusion")
```

Jan 9 17:09 1991 habillage.lisp Page 1

```
(in-package "USER")

;;;                          HABILLAGE (phrase)
;;;
;;; but : cette fonction renvoit la phrase entree sous la forme d'une liste de structures de trait
;;;       qui sont les equivallent lexiquaux de chaque element de la phrase. Un mot pouvant avoir
;;;       plusieurs entrees lexicales, celles-ci sont a chaque fois reprise. De plus les liens
;;;       temporels entre les mots sont calcules.
;;;
;;; arg : phrase : phrase mise sous la forme d'une liste de mots
;;;
;;; precond : phrase ne contient que des mots du lexique-grammaire
;;;
;;; poscond : - phrase inchangee
;;;
;;; resultat : une liste de structures de trait avec leurs relations de precedence
;;;
;;; rem : seuls les mots qui appartiennent au lexique-grammaire sont pris en compte, les mots
;;;       inconnus sont ignores

(defun habillage (phrase)
  (hab phrase () (lectgramlex $fich_lex_gram ))
  )

;;; but : transformer un fichier sous format "[" en un fichier lisible par Lisp
;;; rem : le nom du nouveau fichier est celui de l'ancien auquel on a change
;;;       l'extension en .aux

(defun transfich (fich)
  (with-open-file (caract fich :direction :input)
      (with-open-file (cartrans (concatenate 'string (subseq fich 0
                                      (search "." fich)) ".aux")
                                :direction :output)
          (do ((tempo_char (read-char caract nil) (read-char caract nil)))
              ((null tempo_char) ())
              (cond ((eq tempo_char #\[) (write-char #\( cartrans))
                    ((eq tempo_char #\]) (write-char #\) cartrans))
                    ((eq tempo_char #\=) ())
                    ((eq tempo_char #\#) (progn
                                           (write-char #\! cartrans)
                                           (write-char #\space cartrans)))
                    (t (write-char tempo_char cartrans))
              ))
          )
      )
  )

;;; but : lire un fichier contenant la grammaire et le lexique,  renvoyer une liste
;;;       les contenant sous forme de 'struc_de_trait
;;; rem : - seuls les champs 'code, 'struc, 'env, syntaxe sont garnis
;;;       - le fichier doit etre sous une forme lisible, c'est a dire qu'il a du etre transforme
;;;         prealablement par la fonction 'transfich (ci-dessus)

(defun lectgramlex (nom_fichier)
  (with-open-file (ligne nom_fichier :direction :input)
        (do ((ens_lignes () (cons temp ens_lignes))
             (une_ligne (read ligne nil) (read ligne nil)))
            ((not une_ligne) ens_lignes)
            (setq $count_struc (+ 1 $count_struc))
            (setq temp (make-struc_de_trait  :code $count_struc
                                             :struc (first une_ligne)
                                             :env (second une_ligne)
                                             :syntaxe (third une_ligne)))
        )
    )
)


;;;but : donner une liste de 'struc_de_trait dont les elements ont comme
;;;      valeur de 'lex  'mot dans le champ syntaxe.

(defun cherche_mot_lex (mot lexique)
  (when lexique
        (let* ((prem_mot_lex (first lexique))
               (val_lex_prem (second (member 'lex
                                             (struc_de_trait-syntaxe
                                              prem_mot_lex)))))
              (if (eq val_lex_prem mot)
                  (progn (setq premier (make-struc_de_trait))
                         (setq $count_struc (+ 1 $count_struc))
                         (setf (struc_de_trait-code premier) $count_struc)
                         (setf (struc_de_trait-satisfait premier) 'oui)
                         (setf (struc_de_trait-env premier)
                                  (struc_de_trait-env prem_mot_lex))
```

```
                    (setf (struc_de_trait-syntaxe premier)
                          (struc_de_trait-syntaxe prem_mot_lex))
                    (cons premier (cherche_mot_lex mot (rest lexique)))
                    )
                (cherche_mot_lex mot (rest lexique))
              )
            )
        )
  )


;;; but : donner une liste de structure de trait dont les elements ont comme
;;;       valeur de 'lex 'mot dans le champ syntaxe et qui sont les premiers
;;;       de la 'phrase

(defun cherche_prem_mot (mot phrase_hab)
  (when phrase_hab
        (let ((prem_mot_phr (first phrase_hab)))
          (if (equal (vallex prem_mot_phr) mot)
              (append (list prem_mot_phr) (cherche_prem_mot mot (rest phrase_hab)))
            )
          )
        )
  )


;;; but : ajout des pointeurs de liste_lex (liste des elements du lexique ayant meme
;;;       valeur de 'lex) avec l'element suivant de la phrase

(defun ajout_pteur_liste (liste_lex elem)
 (dolist (une_struc liste_lex liste_lex)
         (push elem (struc_de_trait-prec une_struc))
         (push une_struc (struc_de_trait-prec_i elem))
         (push (list 'prec une_struc elem) $prec_inc)
         )
  )

;;; but : ajouter les structures du trait qui on comme valeur lexicale 'mot

(defun ajout_liste_lex (liste_lex phrase_hab mot)
(if phrase_hab
    (if (and (eq (second (member 'lex (struc_de_trait-syntaxe (first phrase_hab))))
                 mot)
             mot)
      (progn (ajout_liste_lex liste_lex (rest phrase_hab) mot)
             (ajout_pteur_liste liste_lex (first phrase_hab))
             )
      )
  )
 )

; but : habiller une phrase en tenant compte d'un lexique donne

(defun hab (phrase phrase_hab lexique)
  (if  phrase
      (let* ((prem_mot (first phrase))
             (liste_prem (cherche_mot_lex prem_mot lexique))
             (hab_rest_phrase (hab (rest phrase) phrase_hab lexique)))
        (if hab_rest_phrase
            (let ((mot1 (second (member 'lex (struc_de_trait-syntaxe
                                              (first hab_rest_phrase))))))
                  (ajout_liste_lex liste_prem hab_rest_phrase mot1)
                  (append liste_prem hab_rest_phrase)
                  )
            liste_prem
            )
         )
    )
  )

;;; fonction de remplacement des etiquettes d'une liste d'etiquettes
;;; par leur valeur (structure partagee)

(defun tr_champ_struc (lstruc app)
  (if lstruc
        (if (eq (first lstruc) '@)
            (let ((etiq (cdr (assoc (second lstruc) app))))
               (if (atom etiq)
                   (cons etiq (tr_champ_struc (rest (rest lstruc)) app))
                   (append (list etiq) (tr_champ_struc (rest (rest lstruc)) app))
                   )
              )
            (append (first lstruc) (tr_champ_struc (rest lstruc) app))
          )
```

```
        )
  )
```

Aug 7 12:06 1991 assemble.lisp Page 1

```
(in-package "USER")

;;;                          ASSEMBLE (sup inf etiquette)
;;;
;;; but : reunir une structure de trait  une structure d'un niveau superieur (sup)
;;;       avec une de niveau directement inferieur (inf) grace a une etiquette
;;; rem : 'reunir = faire une nvelle structure
;;;
;;; arg : - sup : structure de trait "superieure" (+ general)
;;;       - inf : structure de trait "inferieure" qui sera inseree dans sup
;;;       - etiquette : une etiquette qui fait partie du champ 'struc de sup
;;;
;;; precond : - inf et sup doivent etre du type 'struc_de_trait et sont "traites"
;;;             (cf. fonction traitement = suppression des '<= )
;;;           - etiquette appartient a (struc_de_trait-struc sup) et a (struc_de_trait-syntaxe sup)
;;;
;;; postcond : sup, inf, etiquette sont inchanges
;;;
;;; resultat : - un drapeau indiquant si l'operation s'est bien passe ($OK ) ou non
;;;              (cause de l'echec)
;;;            - la nouvelle structure creee par l'assemblage de sup et inf en cas de reussite

(defun assemble (sup inf etiquette)
  (let* ((nvelle (crea_struc))
         (nvelle_rel (crea_liste_nvelles_rel nvelle sup inf etiquette)))
    (multiple-value-bind (arcs drap)
        (calcul_transitions_t nvelle_rel :pre_etudies $prec_inc)
        (if (eq drap '$OK)
            (let* ((a_sup  (copie_liste (struc_de_trait-syntaxe sup)))
                   (a_inf  (copie_liste (struc_de_trait-syntaxe inf)))
                   (res_unif (unif_s_p a_sup a_inf etiquette)))
              (if (eq res_unif 'echec)
                  (values 'echec_a_l_unification ())
                (progn
                  (setf (struc_de_trait-type nvelle) (struc_de_trait-type sup))
                  (m_a_j_prec sup inf etiquette nvelle)
                  (m_a_j_prec_i sup inf etiquette nvelle)
                  (m_a_j_inc_inc_i sup inf nvelle)
                  (setf (struc_de_trait-env nvelle) (struc_de_trait-env sup))
                  (setf (struc_de_trait-struc nvelle) (nv_struc sup inf etiquette))
                  (setf (struc_de_trait-syntaxe nvelle) res_unif)
                  (m_a_j_satisfait nvelle)
                  (values '$OK nvelle)
                  )
                )
              )
          (values '$INCOMPATIBLE ())
          )
        )
)
)


;;; but : modifier le champ 'struc en remplacant '@ etiquette par le couple
;;;       grde petite

(defun nv_struc (grde petite etiquette)
  (let* ((stru (struc_de_trait-struc grde))
         (fin (nthcdr (+ 1 (search (list etiquette) stru)) stru))
         (debut (reverse (nthcdr (+ 2 (search (list etiquette) (reverse stru))) (reverse stru))))
         )
    (append debut (cons (cons  grde petite) fin))
    )
  )


;;; but : effectue le traitement du champ syntaxe d'une structure de trait (affectation
;;;       des etiquettes)
;;; rem : il n'y a pas recopie de la structure de trait mais simplement modification
;;;       du champ syntaxe

(defun tr_struc_ent (struc_ent)
  (let ((synt_struc_ent (struc_de_trait-syntaxe struc_ent)))
    (setf (struc_de_trait-syntaxe struc_ent) ())
    (setf (struc_de_trait-syntaxe struc_ent) (traitement synt_struc_ent))
    (values struc_ent)
    )
  )


;;; but : creer une liste de nouvelles relations prec inc

(defun crea_liste_nvelles_rel (nvelle sup inf etiquette)
 (let* ((nv_listel (ajout_prec sup inf etiquette nvelle ))
```

```
        (nv_liste2 (ajout_prec_i sup inf etiquette nvelle nv_liste1))
        (nv_liste3 (ajout_inc sup inf nvelle nv_liste2))
        (nv_liste4 (ajout_inc_i sup inf nvelle nv_liste3))
       )
   )
)


;;; but : effectuer la mise a jour du champ prec de la nouvelle structure
;;;       on reprend celui de l'ancienne (sup) et on ajoute celui de
;;;       l'inferieure si l'etiquette est la dernier du champ struc.
;;;       De plus, il faut mettre a jour le champ prec_i de toutes les
;;;       structures du champs prec de nvelle

(defun m_a_j_prec (sup inf etiq nvelle)
  (if (dernieretiq etiq sup)
      (setf (struc_de_trait-prec nvelle) (append (struc_de_trait-prec inf)
                                                 (struc_de_trait-prec sup))
            )
      (setf (struc_de_trait-prec nvelle) (struc_de_trait-prec sup))
      )
  (m_a_j_autre_prec_i nvelle)
  )


;;; but : mettre a jour le champ prec_i des structures du champ prec de nvelle

(defun m_a_j_autre_prec_i (nvelle)
  (let ((chp_prec (struc_de_trait-prec nvelle)))
    (dolist (une_struc chp_prec chp_prec)
            (setf (struc_de_trait-prec_i une_struc) (append (list nvelle)
                                                            (struc_de_trait-prec_i une_struc)))
            (push (list 'prec nvelle une_struc) $prec_inc)
            )
    )
  )


;;; but : effectuer la mise a jour du champ prec_i de la nouvelle structure
;;;       on reprend celui de l'ancienne (sup) et on ajoute celui de
;;;       l'inferieure si l'etiquette est la premier du champ struc

(defun m_a_j_prec_i (sup inf etiq nvelle)
  (let ((champ_struc_sup (struc_de_trait-struc sup)))
    (if (and (eq '@ (first champ_struc_sup)) (eq etiq (second champ_struc_sup)))
        (setf (struc_de_trait-prec_i nvelle) (append (struc_de_trait-prec_i sup)
                                                     (struc_de_trait-prec_i inf)))
        (setf (struc_de_trait-prec_i nvelle) (struc_de_trait-prec_i sup))
        )
    (m_a_j_autre_prec nvelle)
    )
  )

;;; but : mettre a jour le champ prec des structures du champ prec_i de nvelle

(defun m_a_j_autre_prec (nvelle)
  (let ((chp_prec_i (struc_de_trait-prec_i nvelle)))
    (dolist (une_struc chp_prec_i chp_prec_i)
            (setf (struc_de_trait-prec une_struc) (append (list nvelle)
                                                          (struc_de_trait-prec une_struc)))
            (push (list 'prec une_struc nvelle) $prec_inc)
            )
    )
)


;;; but : effectuer la mise a jour du champ inc de la nouvelle structure et
;;;       de celui de la structure inferieure

(defun m_a_j_inc_inc_i (sup inf nvelle)
  (setf (struc_de_trait-inc inf) (append (list nvelle) (struc_de_trait-inc inf)))
  (setf (struc_de_trait-inc_i nvelle) (append (list inf) (struc_de_trait-inc sup)))
  (m_a_j_autre_inc nvelle)
  (m_a_j_autre_inc_i nvelle)
  )


;;; but : mettre a jour le champ inc des structures du champ inc_i de nvelle

(defun m_a_j_autre_inc (nvelle)
  (let ((chp_inc_i (struc_de_trait-inc_i nvelle)))
    (dolist (une_struc chp_inc_i chp_inc_i)
            (setf (struc_de_trait-inc une_struc) (append (list nvelle)
                                                         (struc_de_trait-inc une_struc)))
```

```
            (push (list 'in une_struc nvelle) $prec_inc)
            )
    )
  )

;;; but : mettre a jour le champ inc_i des structures du champ inc de nvelle

(defun m_a_j_autre_inc_i (nvelle)
  (let ((chp_inc (struc_de_trait-inc nvelle)))
    (dolist (une_struc chp_inc chp_inc)
            (setf (struc_de_trait-inc_i une_struc) (append (list nvelle)
                                                           (struc_de_trait-inc_i une_struc)))
            (push (list 'in nvelle une_struc) $prec_inc)
            )
    )
  )

;;; but determine si l'etiquette se trouve en fin du champ struc

(defun dernieretiq (etiq sup)
  (let ((chp_struc (struc_de_trait-struc sup)))
    (eq (car (last chp_struc)) etiq)
    )
  )



;;; but : construire une liste contenant les suivants de la future nouvelle structure
;;;       dans le cas d'un assemblage

(defun ajout_prec (sup inf etiquette nvelle)
  (if (dernieretiq etiquette sup)
      (let ((l_prec (append (struc_de_trait-prec sup) (struc_de_trait-prec inf))))
        (ajout_autre_prec_i l_prec nvelle)
        )
      (let ((l_prec (struc_de_trait-prec sup)))
        (ajout_autre_prec_i l_prec nvelle)
        )
      )
  )


;;; but :ajouter de nvelles relations de "suivante"

(defun ajout_autre_prec_i (l_prec nvelle)
  (let ((nv_liste ()))
    (dolist (une_struc l_prec nv_liste)
            (push (list 'prec nvelle une_struc) nv_liste)
          )
    )
  )

;;; but : ajouter a une liste les relations de precedence du a l'arrivee de nvelle
;;;       dans le cas d'un "assemblage"

(defun ajout_prec_i (sup inf etiquette nvelle liste1)
  (let ((champ_struc_sup (struc_de_trait-struc sup)))
    (if (and (eq '@ (first champ_struc_sup)) (eq etiquette (second champ_struc_sup)))
      (let ((l_prec (append (struc_de_trait-prec_i inf) (struc_de_trait-prec_i sup))))
        (ajout_autre_prec l_prec liste1 nvelle )
        )
      (let ((l_prec (struc_de_trait-prec sup)))
        (ajout_autre_prec l_prec liste1 nvelle)
        )
      )
    )
  )


;;; but :ajouter de nvelles relations de precedence

(defun ajout_autre_prec (l_prec liste1 nvelle)
  (dolist (une_struc l_prec liste1)
          (push (list 'prec  une_struc nvelle) liste1)
          )
  )

;;; but ajouter de nouvelles relations d'inclusion dans le cas d'un "assemblage"

(defun ajout_inc_i (sup inf nvelle liste2)
  (let ((liste3 (append (list 'in inf nvelle) liste2))
        (chp_inc_i_sup (struc_de_trait-inc_i sup))
        )
```

```
    (dolist (une_struc chp_inc_i_sup liste3)
            (push (list 'in une_struc nvelle) liste3)
            )
    )
  )

;;; but : ajouter de nouvelles relations d'inclusion lors de l'assemblage

(defun ajout_inc (sup inf nvelle liste3)
  (let ((chp_inc_sup (struc_de_trait-inc sup)))
    (dolist (une_struc chp_inc_sup liste3)
            (push (list 'in nvelle une_struc) liste3)
            )
    )
  )

;;; but : mettre a jour le champ satisfait s'il y a lieu, c-a-d si le champ struc
;;;       ne comporte plus d'etiquette

(defun m_a_j_satisfait (nvelle)
  (if (not (member '@ (struc_de_trait-struc nvelle)))
      (setf (struc_de_trait-satisfait nvelle) 'oui)
    )
  )
```

```
(in-package "USER")

;;;                                FUSION (struc1 struc2)
;;;
;;; but : fonction qui renvoit 'OK si la fusion s'est bien passe ou la raison de l'echec dans le cas
;;;       contraire, et une structure qui est le resultat de la fusion de deux structures de trait,
;;;       en fait elle effectue l'unification entre les deux champs syntaxe dans leur totalite.
;;; rem : on ne peut fusionner que des structures qui ont ete instanciees au moins une fois
;;;
;;; arg : - struc1 : structure de trait
;;;       - struc2 : structure de trait
;;;
;;; precond : struc1 et struc2 doivent etre du type 'struc_de_trait et sont "traites"
;;;           (cf. fonction traitement = suppression des '<= )
;;;
;;; postcond : struc1 et struc2 inchanges
;;;
;;; resultat : - un drapeau indiquant si l'operation s'est bien passe ($OK ) ou non
;;;              (cause de l'echec)
;;;            - la nouvelle structure creee par la fusion des deux structures, c'est a dire une
;;;              structure de trait dont le champ syntaxe est egal a l'unification du champ syntaxe
;;;              de chacune des deux structures de trait

(defun fusion (struc1 struc2)
  (let* ((nvelle (crea_struc))
         (synt1 (struc_de_trait-syntaxe struc1))
         (synt2 (struc_de_trait-syntaxe struc2))
         (chp_struc1 (struc_de_trait-struc struc1))
         (chp_struc2 (struc_de_trait-struc struc2))
         (res_verif_type (verif_anc struc1 struc2))
         )
    (if (not res_verif_type)
        (values 'type_de_structure_different ())
        (let ((res_verif_temp (verif_temp nvelle struc1 struc2)))
          (if (not(eq res_verif_temp '$OK))
              (values 'echec_a_la_verification_temporelle ())
              (multiple-value-bind (app res)
                  (unif_gen () (copie_liste synt1) (copie_liste synt2))
                (if (eq app 'echec)
                    (values 'echec_a_l_unification ())
                    (values '$OK (m_a_j_nvelle struc1 struc2 nvelle res))
                    )
                )
              )
          )
        )
    )
  )


;;; but : verifier que deux structures sont de meme type generique
;;;       cette fonction renvoit 'OK en cas de reussite, nil sinon

(defun verif_type_gen (struc1 struc2)
  (if (equal (struc_de_trait-type struc1) (struc_de_trait-type struc2))
      'OK
    )
  )


;;; but : construire une liste contenant les suivants de la future nouvelle structure
;;;       dans le cas d'une fusion

(defun ajout_prec_fusion (struc1 struc2 nvelle)
  (let ((l_prec (append (struc_de_trait-prec struc1) (struc_de_trait-prec struc2))))
    (ajout_autre_prec_i l_prec nvelle)
    )
  )

;;; but : ajouter a une liste les relations de precedence du a l'arrivee de nvelle
;;;       dans le cas d'une fusion

(defun ajout_prec_i_fusion (struc1 struc2 nvelle)
  (let ((l_prec_i (append (struc_de_trait-prec_i struc1) (struc_de_trait-prec_i struc2))))
    (ajout_autre_prec l_prec_i () nvelle)
    )
  )

;;; but ajouter de nouvelles relations d'inclusion dans le cas d'une fusion

(defun ajout_inc_i_fusion (struc1 struc2 nvelle)
  (let ((l_nvelle ()))
       (chp_inc_i_gl (append (struc_de_trait-inc_i struc1)(struc_de_trait-inc_i struc2)))
       )
```

```
    (dolist (une_struc chp_inc_i_gl l_nvelle)
            (push (list 'in une_struc nvelle) l_nvelle)
           )
    )
  )

;;; but ajouter de nouvelles relations d'inclusion dans le cas d'une fusion

(defun ajout_inc_fusion (struc1 struc2 nvelle)
  (let ((l_nvelle ())
        (chp_inc_gl (append (struc_de_trait-inc struc1)(struc_de_trait-inc struc2)))
        )
    (dolist (une_struc chp_inc_gl l_nvelle)
            (push (list 'in  nvelle une_struc) l_nvelle)
           )
    )
  )

;;; but : effectuer les verifications de coherence temporelle dans le cas d'une fusion

(defun verif_temp (nvelle struc1 struc2)
  (let* ((l1 (ajout_prec_fusion struc1 struc2 nvelle))
         (l2 (append l1 (ajout_prec_i_fusion struc1 struc2 nvelle)))
         (l3 (append l2 (ajout_inc_i_fusion struc1 struc2 nvelle)))
         (l4 (append l3 (ajout_inc_fusion struc1 struc2 nvelle)))
         )
    (multiple-value-bind (arcs drap)
        (calcul_transitions_t l4 :pre_etudies $prec_inc)
        drap
        )
    )
  )

;;; but : mettre a jour les champs de la nouvelle structure pour la fusion lorsque
;;;       la coherence temporelle est respectee et que l'unification s'est bien passee
;;; rem : struc1 a priorite pour le champ env et pour le champ struc

(defun m_a_j_nvelle (struc1 struc2 nvelle res_unif)
  (setf (struc_de_trait-type nvelle) (struc_de_trait-type struc1))
  (m_a_j_prec_fusion struc1 struc2 nvelle)
  (m_a_j_prec_i_fusion struc1 struc2 nvelle)
  (m_a_j_inc_fusion struc1 struc2 nvelle)
  (m_a_j_inc_i_fusion struc1 struc2 nvelle)
  (setf (struc_de_trait-syntaxe nvelle) res_unif)
  (setf (struc_de_trait-env nvelle) (struc_de_trait-env struc1))
  (m_a_j_struc_fusion  struc1 struc2 nvelle)
  (m_a_j_satisfait nvelle)
  nvelle
  )


;;; but : mettre a jour le champ prec ainsi que la variable $prec_inc

(defun m_a_j_prec_fusion (struc1 struc2 nvelle)
  (let* ((l_prec (append (struc_de_trait-prec struc1) (struc_de_trait-prec struc2))))
    (setf (struc_de_trait-prec nvelle) l_prec)
    (m_a_j_autre_prec_i nvelle)
    )
  )

;;; but : mettre a jour le champ prec_i ainsi que la variable $prec_inc

(defun m_a_j_prec_i_fusion (struc1 struc2 nvelle)
  (let ((l_prec_i (append (struc_de_trait-prec_i struc1) (struc_de_trait-prec_i struc2))))
    (setf (struc_de_trait-prec_i nvelle ) l_prec_i)
    (m_a_j_autre_prec nvelle)
    )
  )

;;; but : mettre a jour le champ inc_i ainsi que la variable $prec_inc

(defun m_a_j_inc_i_fusion (struc1 struc2 nvelle)
  (let ((chp_inc_i_gl (append (struc_de_trait-inc_i struc1)(struc_de_trait-inc_i struc2)))
        )
    (setf (struc_de_trait-inc_i nvelle) chp_inc_i_gl)
    (dolist (une_struc chp_inc_i_gl $prec_inc)
            (setf (struc_de_trait-inc_i une_struc) (append (list nvelle)
                                                           (struc_de_trait-inc_i une_struc)))
            (push (list 'in une_struc nvelle) $prec_inc)
           )
    )
  )

;;; but : mettre a jour le champ inc ainsi que la variable $prec_inc
```

```
(defun m_a_j_inc_fusion (struc1 struc2 nvelle)
  (let ((chp_inc_gl (append (struc_de_trait-inc struc1)(struc_de_trait-inc struc2)))
        )
    (setf (struc_de_trait-inc nvelle) chp_inc_gl)
    (dolist (une_struc chp_inc_gl $prec_inc)
            (push (list 'in nvelle une_struc) $prec_inc)
           )
    )
  )


;;; but : mettre a jour le champ struc de la nouvelle structure = superposer les deux champs
;;;       struc de chacune des deux structures generatrices
;;; rem : s'il existe deux champs communs aux 2 sous-structures, on impose celui de la premiere

(defun m_a_j_struc_fusion (struc1 struc2 nvelle)
  (let ((chp_struc1 (struc_de_trait-struc struc1))
        (chp_struc2 (struc_de_trait-struc struc2))
        )
    (setf (struc_de_trait-struc nvelle) (cal_struc_fusion chp_struc1 chp_struc2))
    )
  )

;;; but : calculer la valeur du champ struc dans le cas d'une fusion

(defun cal_struc_fusion (chp_struc1 chp_struc2)
  (when chp_struc1
        (cond ((and (atom (first chp_struc1)) (atom (first chp_struc2)))
               (cons '@ (cons (second chp_struc1) (cal_struc_fusion (nthcdr 2 chp_struc1)
                                                                    (nthcdr 2 chp_struc2))))
               )
              ((and (not (atom (first chp_struc1))) (not (atom (first chp_struc2))))
               (cons (first chp_struc1) (cal_struc_fusion (rest chp_struc1) (rest chp_struc2)))
               )
              ((and (atom (first chp_struc1)) (not (atom (first chp_struc2))))
               (cons (first chp_struc2) (cal_struc_fusion (nthcdr 2 chp_struc1) (rest chp_struc2)))
               )
              (t
               (cons (first chp_struc1) (cal_struc_fusion (rest chp_struc1) (nthcdr 2 chp_struc2)))
               )
              )
        )
  )
```

```
;;; ensemble de fonctions auxillaires servant a retrouver les champs
;;; ou parties de champs d'une structure de trait (ST) ou servant a
;;; l'affichage de certains champs
;;; plus une fonction 'transfo_lex_gram qui 'traite tous les elements
;;; du lexique et de la grammaire
;;; plus toute une serie d'autres fonctions

(in-package "USER")

;;; but : renvoyer la valeur du champ "lex" d'une structure de trait

(defun vallex (struc)
  (second (member 'lex (struc_de_trait-syntaxe struc)))
  )

;;; but : renvoyer la valeur du champ "cat" d'une structure de trait

(defun valcat (struc)
  (second (member 'cat (struc_de_trait-syntaxe struc)))
  )

;;; but : renvoyer la valeur du champ environnement d'une ST

(defun liste_env (struc)
  (struc_de_trait-env struc)
  )

;;; but : renvoyer la valeur du champ structure d'une ST

(defun liste_struc (structure)
  (struc_de_trait-struc structure)
  )

;;; but : renvoyer la valeur du champ satisfait d'une ST

(defun satisfait (structure)
  (struc_de_trait-satisfait structure)
  )

;;; but : renvoyer la liste des pointeurs avant d'une ST

(defun list_pointeurs_avant (structure)
   (struc_de_trait-prec structure)
   )
;;; but : renvoyer la liste des pointeurs arriere d'une ST

(defun list_pointeurs_arriere (structure)
  (struc_de_trait-prec_i structure)
  )

;;; but : renvoyer le champ syntaxe d'une ST

(defun synt (struc)
  (struc_de_trait-syntaxe struc)
  )


;;; fonction d'impression d'une structure de trait

(defun print_struc (trait sortie profondeur)
  (if (eq $champ_type '+)
      (format sortie "~% type : ~S" (struc_de_trait-type trait)))
  (if (eq $champ_code '+)
      (format sortie "~% code : ~S" (struc_de_trait-code trait)))
  (if (eq $champ_struc '+)
      (impres_champ_struc trait sortie))
  (if (eq $champ_env '+)
      (format sortie "~% env : ~S" (struc_de_trait-env trait)))
  (if (eq $champ_op_eff '+)
      (format sortie "~% op_eff : ~S" (struc_de_trait-op_eff trait)))
  (if (eq $champ_satisfait '+)
      (format sortie "~% satisfait : ~S" (struc_de_trait-satisfait trait)))
  (if (eq $champ_synt '+)
      (aff_synt (struc_de_trait-syntaxe trait) sortie))
  (format sortie "~%")
  )

;;; fonction d'impression du champs struc d'une structure de trait

(defun impres_champ_struc (trait sortie)
  (let ((champ_struc (struc_de_trait-struc trait)))
    (format sortie "~% struc : ~S" (for_struc champ_struc))
    )
  )
```

```
;;; mise en forme du champ struc par l'analyse atome, paire pointee.
;;; Le champ struc est uniquement compose d'atomes et de paires pointees

(defun for_struc (champ_struc)
  (if (not champ_struc)
      ()
      (let ((prem_struc (first champ_struc)))
        (if (atom prem_struc)
            (cons prem_struc (for_struc (rest champ_struc)))
            (cons (an_pair  prem_struc) (for_struc (rest champ_struc)))
        )
      )
  )

;;; mise en forme d'une paire pointee composee de deux structure de trait
;;; seul le champ code est pris en compte

(defun an_pair (l_struc)
  (let ((prem_struc (car  l_struc))
        (deux_struc (cdr  l_struc)))
       (cons (struc_de_trait-code prem_struc) (struc_de_trait-code deux_struc))
       )
  )

;;; fonction assurant l'affichage du champ syntaxe d'une structure de trait
;;; rem : l'affichage se fait avec une presentation en colonnes lorsque la
;;;       variable $syntaxe_col = O ou o
;;;       par defaut, cette variable est mise a N


(defun aff_synt (chp_syntaxe sortie)
  (format sortie "~% syntaxe : ")
  (if (or (eq $syntaxe_col 'O) (eq $syntaxe_col 'o))
      (progn (format sortie "~%    ( ~%")
             (aff_synt_col chp_syntaxe 2 1 sortie))
      (format sortie "~S" chp_syntaxe)
      )
  )


;;; fonction affichant le champ syntaxe sous forme de colonnes
;;; avec le niveau de profondeur desire ($prof_synt)
;;; rem : ne fonctionne que pour des ST "traitees"

(defun aff_synt_col (chp_syntaxe tab niveau sortie)
  (if (> niveau $prof_synt)
      (format sortie "~A ... ~% ~A ) ~%" (tabu tab) (tabu tab))
    (if (not chp_syntaxe)
        (let ((tabl (tabu tab)))
          (format sortie "~A ) ~%" tabl)
          )
        (let ((prem (first chp_syntaxe))
              (deux (second chp_syntaxe))
              (trois (third chp_syntaxe))
              (quatre (fourth chp_syntaxe))
              (tabl (tabu tab))
              )
          (cond ((and (atom deux) (not (eq deux '@)))
                 (progn (format sortie "~A ~S ~S ~%" tabl prem deux)
                        (aff_synt_col (nthcdr 2 chp_syntaxe) tab niveau sortie)
                        )
                 )
                ((not (atom deux))
                 (let ((nv_tab (+ tab (length (format nil "~a" prem)))))
                   (format sortie "~A ~S  (" tabl prem)
                   (aff_synt_col deux nv_tab (+ 1 niveau) sortie)
                   (aff_synt_col (nthcdr 2 chp_syntaxe) tab niveau sortie)
                   )
                 )
                ((and (eq deux '@) (not (atom quatre)))
                 (let ((nv_tab (+ tab (length (format nil "~a" prem))
                                  4 (length (format nil "~a" trois)))))
                   (format sortie "~A ~S @ ~S ( ~ ~%" tabl prem trois)
                   (aff_synt_col quatre nv_tab (+ 1 niveau) sortie)
                   (aff_synt_col (nthcdr 4 chp_syntaxe) tab niveau sortie)
                   )
                 )
                ((and (eq deux '@) (atom quatre))
                 (progn (format sortie "~A ~S @ ~S  ~S ~%" tabl prem trois quatre)
                        (aff_synt_col (nthcdr 4 chp_syntaxe) tab niveau sortie)
                        )
                 )
```

```
              ( t
                (format sortie "erreur")
                )
              )
        )
      )
    )
  )

;;; but : construire un string de 'tab blancs

(defun tabu (tab)
  (make-string tab :initial-element #\ )
  )

;;; but : traiter (cf. fonction tr_struc_ent) toutes les structures d'une liste

(defun transfo_lex_gram (l)
  (let ((lt ()))
    (dolist (une_struc l lt)
            (push (tr_struc_ent une_struc) lt)
           )
    )
  )


;;; but : renvoyer la premiere etiquette d'une liste d'etiquettes-paires pointees.
;;;       cf. champ struc

(defun prem_etiq (l)
  (if (not l)
      ()
      (if (eq (first l) '@)
          (second l)
          (prem_etiq (cdr l))
          )
      )
  )

;;; but : remvoyer la derniere paire pointee d'une liste d'etiquettes-paires pointees
;;;       cf. champ struc
;;; rem : la liste doit etre composee, d'abord de paires pointee, puis ensuite d'etiquettes

(defun dern_paire_point (l)
  (if (not l)
      ()
    (if (atom (first l))
        'erreur
        (if (not (second l))
            (first l)
            (if (atom (second l))
                (first l)
              (dern_paire_point (rest l))
              )
            )
        )
    )
  )

;;; but : renvoyer la structure suivante du deuxieme de la derniere paire pointee
;;;       du champ struc d'une structure donnee

(defun suiv_paire_point (structure)
  (let ((chp_struc (struc_de_trait-struc structure)))
    (suivant (rest (dern_paire_point chp_struc)))
    )
  )

;;; but : renvoyer la categorie d'une 'liste se trouvant juste apres une etiquette determinee

(defun cat_etiq (etiq l)
  (if (not l)
      ()
      (if (atom (first l))
          (if (and (eq (first l) '@) (equal (second l) etiq))
              (second (member 'cat (third l)))
              (cat_etiq etiq (cdr l))
              )
          (let ((res (cat_etiq etiq (first l))))
            (if res
                res
                (cat_etiq etiq (cdr l))
                )
            )
```

```
        )
      )
  )

;;; but : determiner l'element suivant a l'interieur d'une ST

(defun suivant (structure)
  (if (not structure)
      ()
    (if (eq (satisfait structure) 'oui)
        (struc_de_trait-prec structure)
        (suivant (trouv_suiv (struc_de_trait-struc structure)))
        )
    )
  )


;;; but : trouver le dernier couple de champ struc et renvoyer le suivant du
;;;       deuxieme element de ce couple

(defun trouv_suiv (chp_struc)
  (if (not chp_struc)
      ()
    (if (eq (second chp_struc) '@)
        (cdr (first chp_struc))
        (trouv_suiv (rest chp_struc))
        )
    )
  )

;;; but : determiner l'element precedent a l'interieur d'une ST

(defun precedent (structure)
  (if (not structure)
      ()
      (if (eq (satisfait structure) 'oui)
          (struc_de_trait-prec_i structure)
        (precedent (trouv_prec (struc_de_trait-struc structure)))
        )
      )
  )

;;; but : trouver le premier couple de champ struc et renvoyer le precedent du
;;;       deuxieme element de ce couple

(defun trouv_prec (chp_struc)
  (if (not chp_struc)
      ()
      (if (atom (first chp_struc))
          (trouv_prec (rest chp_struc))
          (cdr (first chp_struc))
          )
      )
  )

;;; but : donner le suivant du dernier couple du champ structure

(defun suiv_der_couple (chp_struc)
  (if (not chp_struc)
      ()
      (if (or (and (atom (first chp_struc)) (not (atom (second chp_struc))))
              (not (atom (first chp_struc))))
          (if (not (atom (first chp_struc)))
              (trouv_suiv chp_struc)
              (trouv_suiv (rest chp_struc)))
          (suiv_der_couple (rest chp_struc))
          )
      )
  )



;;; but : donner toutes les structures generiques qui ont une certaine categorie
;;;       a partir du lexique_grammaire ou autre liste de strucures
;;;       avec recopie

(defun struc_de_cat (categ lex_gram)
  (when lex_gram
        (let* ((prem_mot_lex (first lex_gram))
               (val_cat_prem (second (member 'cat
                                             (struc_de_trait-syntaxe
                                                   prem_mot_lex)))))
              (if (eq val_cat_prem categ)
                  (cons (copie_structure prem_mot_lex) (struc_de_cat categ (rest lex_gram)))
```

```
                (struc_de_cat categ (rest lex_gram))
              )
            )
        )
  )
;;; but : donner toutes les structures generiques qui ont une certaine categorie
;;;       a partir du lexique_grammaire
;;;       sans recopie

(defun struc_de_cat_sans_recopie (categ lex_gram)
  (when lex_gram
      (let* ((prem_mot_lex (first lex_gram))
             (val_cat_prem (second (member 'cat
                                           (struc_de_trait-syntaxe
                                                 prem_mot_lex)))))
            (if (eq val_cat_prem categ)
                (cons prem_mot_lex (struc_de_cat_sans_recopie categ (rest lex_gram)))
                (struc_de_cat_sans_recopie categ (rest lex_gram))
              )
            )
        )
  )

;;; but : donner toutes les structures generiques qui ont une certaine categorie
;;;       a partir du lexique_grammaire sauf si ce sont des 'mots terminaux'


(defun struc_de_cat_sans_lex (categ lex_gram)
  (when lex_gram
        (let* ((prem_mot_lex (first lex_gram))
               (val_cat_prem (second (member 'cat
                                             (struc_de_trait-syntaxe
                                                   prem_mot_lex)))))
              (if (eq val_cat_prem categ)
                  (if (member 'lex (struc_de_trait-syntaxe prem_mot_lex))
                      ()
                      (cons (copie_structure prem_mot_lex) (struc_de_cat_sans_lex categ (rest lex_gram)))
                      )
                  (struc_de_cat_sans_lex categ (rest lex_gram))
                )
              )
        )
  )

;;; but : copier une structure de trait

(defun copie_structure (structure)
  (let ((nvelle (make-struc_de_trait))
        )
    (setf (struc_de_trait-type nvelle) (struc_de_trait-type structure))
    (setf (struc_de_trait-code nvelle) (struc_de_trait-code structure))
    (setf (struc_de_trait-struc nvelle) (struc_de_trait-struc structure))
    (setf (struc_de_trait-env nvelle) (struc_de_trait-env structure))
    (setf (struc_de_trait-prec nvelle) (struc_de_trait-prec structure))
    (setf (struc_de_trait-prec_i nvelle) (struc_de_trait-prec_i structure))
    (setf (struc_de_trait-inc nvelle) (struc_de_trait-inc structure))
    (setf (struc_de_trait-inc_i nvelle) (struc_de_trait-inc_i structure))
    (setf (struc_de_trait-op_eff nvelle) (struc_de_trait-op_eff structure))
    (setf (struc_de_trait-satisfait nvelle) (struc_de_trait-satisfait structure))
    (multiple-value-bind (nv_struc tab_etiq1)
        (change_etiq (struc_de_trait-struc structure) ())
        (setf (struc_de_trait-struc nvelle) nv_struc)
        (multiple-value-bind (nv_synt tab_etiq2)
            (change_etiq (struc_de_trait-syntaxe structure) tab_etiq1)
            (setf (struc_de_trait-syntaxe nvelle) nv_synt)
            )
        )
    (values nvelle)
    )
  )


;;; but : mettre de nouveaux numero aux etiquettes

(defun change_etiq (l tab_etiq)
  (if (not l)
      (values () tab_etiq)
      (if (atom (first l))
          ;;; atome => pas besoin de faire de changements
          (if (eq (first l) '@)
              ;; etiquette a changer
              (let* ((anc_etiq (second l))
                     (res_assoc (assoc anc_etiq tab_etiq))
                     )
```

```
                (if res_assoc
                    (multiple-value-bind (res nv_tab_etiq)
                       (change_etiq (nthcdr 2 l) tab_etiq)
                       (values (cons '@ (cons (cdr res_assoc) res)) nv_tab_etiq)
                       )
                    (let ((num_etiq (+ 1 $etiq)))
                      (setq $etiq (+ 1 $etiq))
                      (push (cons anc_etiq $etiq) tab_etiq)
                      (multiple-value-bind (res nv_tab_etiq)
                         (change_etiq (nthcdr 2 l) tab_etiq)
                         (values (cons '@ (cons num_etiq res)) nv_tab_etiq)
                         )
                      )
                    )
                )
               ;; atome mais pas une etiquette
               (multiple-value-bind (res nv_tab_etiq)
                   (change_etiq (rest l) tab_etiq)
                   (values (cons (first l) res) nv_tab_etiq)
                   )
               )
             ;;; le premier element de l est une liste => changer les etiquettes
             (multiple-value-bind (res1 nv_tab_etiq1)
                 (change_etiq (first l) tab_etiq)
                 (multiple-value-bind (res2 nv_tab_etiq2)
                      (change_etiq (rest l) nv_tab_etiq1)
                      (values (cons res1 res2) nv_tab_etiq2)
                      )
                 )
             )
         )
    )

;;; but : faire une copie d'une liste composee d'atomes et de listes

(defun copie_liste (l)
  (when l
        (let ((copie_rest (copie_liste (rest l))))
          (if (atom (first l))
              (cons (first l) copie_rest)
              (cons (copie_liste (first l)) copie_rest)
            )
          )
        )
  )

;;; but : faire une copie du champ struc

(defun copie_chp_struc (chp_struc)
  (when chp_struc
        (cons (first chp_struc) (copie_chp_struc (rest chp_struc)))
        )
  )

;;; but : retrouver dans une liste de structures de trait une structure
;;;       ayant un numero de zone donne

(defun retrouv_zone (num_zone liste_struc)
  (when liste_struc
        (if (equal (struc_de_trait-code (first liste_struc)) num_zone)
            (first liste_struc)
            (retrouv_zone num_zone (rest liste_struc))
            )
        )
   )


;;; but : creer une structure avec un num de code

(defun crea_struc ()
  (let ((nvelle (make-struc_de_trait)))
    (setq $count_struc (+ 1 $count_struc))
    (setf (struc_de_trait-code nvelle) $count_struc)
    nvelle
    )
  )


;;; but : selectionner dans une liste de structures, celles qui ont le champ satisfait a 'oui

(defun select_satisfait (liste_struc)
  (when liste_struc
        (if (satisfait (first liste_struc))
            (cons (first liste_struc) (select_satisfait (rest liste_struc)))
```

```
            (select_satisfait (rest liste_struc))
            )
        )
  )

;;; but : determiner les elements d'une liste de structures qui n'ont pas de champ 'env

(defun select_non_env (liste_struc)
  (when liste_struc
        (if (not(liste_env (first liste_struc)))
            (cons (first liste_struc) (select_non_env (rest liste_struc)))
            (select_non_env (rest liste_struc))
            )
        )
  )

;;; but : selectionner les elements qui sont une phrase, c'est-a-dire ceux qui n'ont pas
;;;       de champ 'env et qui sont satisfait

(defun select_phrase (l_struc)
  (select_non_env (select_satisfait l_struc))
  )

;;; fonction de traitement des structures partagees d'une structure de trait

(defun traite_s_p (struc)
  (multiple-value-bind (app synt)
      (traite () (list (synt struc)))
    (setf (struc_de_trait-syntaxe struc) (first synt))
    (setf (struc_de_trait-struc struc) (tr_champ_struc (liste_struc struc) app))
   )
  )


;;; but : determiner la liste d'etiquettes qui conviendront pour faire l'assemblage
;;;       entre la petite et la grde structure

(defun trouv_etiq (grde petite)
  (let ((chp_struc_grde (struc_de_trait-struc grde))
        (cat_petite (valcat petite)))
    (cherch_etiq chp_struc_grde cat_petite grde)
    )
  )

;;; but : determiner la liste d'etiquettes qui ont comme "categorie" cate, qui sont
;;;       dans la liste l_etiq (chp struc) et qui appartienne a la strucure grde

(defun cherch_etiq (l_etiq cate grde)
  (when l_etiq
        (if (or (eq (first l_etiq) '@) (not (atom (first l_etiq))))
            (cherch_etiq (rest l_etiq) cate grde)
            (let ((cat_pot (cat_etiq (first l_etiq) (struc_de_trait-syntaxe grde))))
              (if (equal cat_pot cate)
                  (cons (first l_etiq) (cherch_etiq (rest l_etiq) cate grde))
                  (cherch_etiq (rest l_etiq) cate grde)
                  )
              )
            )
        )
  )

;;; but : trouver dans une liste (ex : chp struc) la derniere etiquette avant un objet de type liste

(defun etiq_avant_liste (l)
  (when l
        (if (atom (first l))
            (if (second l)
                (if (atom (second l))
                    (etiq_avant_liste (rest l))
                    (first l)
                    )
                (first l)
                )
            )
        )
  )

;;; but : donner les derniers elements d'une liste de strucures qui ont meme valeur lexicale

(defun der_meme_lex (l)
  (when l
        (let ((mot (vallex (first (last l)))))
          (ch_mot_der l mot)
          )
```

```
        )
  )

;;; but : donner les derniers elements d'une liste de structures qui ont
;;;       comme valeur lexicale 'mot

(defun ch_mot_der (l mot)
  (when l
        (let ((der (first (last l))))
          (if (equal (vallex der) mot)
              (cons der (ch_mot_der (remove der l) mot))
            )
          )
        )
  )

;;; but : reinitialiser les variables globales

(defun v ()
  (load "var")
  )
```

Aug 7 12:07 1991 traite.lisp Page 1

```
(in-package "USER")

;;; but : cette fonction met a jpur un ensemble de noeud comprenant des etiquettes dont
;;;       la valeur a leur affectee se trouve juste apres l'etiquette ( <= ) et des
;;;       etiquettes dont la valeur se trouve "ailleurs" (avant ou apres dans la liste).
;;;       Ceci se realisant a partir d'une liste de couples etiquette - valeur de l'etiquette.

(defun traite (appariement noeud)
(if (not noeud)
    (values appariement noeud)
  (if (first noeud)
      (let* ((prem_noeud (car noeud))
             (la_clef (car prem_noeud))
             (La_valeur_plus (cdr prem_noeud)))

        (if (nouv_affect_ou_affect prem_noeud)
            ;traitement des affectations de variables

            (if (nouv_affect prem_noeud)
                ;ajout et affectation d'un nouvel appariement

                (if (nouv_affect_atomique prem_noeud)
                    ; si la nouvelle valeur a affecter est un atome
                    (let* ((nouv_val_atom (fifth prem_noeud))
                           (etiq          (third prem_noeud))
                           (appar2 (cons (cons etiq nouv_val_atom)
                                         appariement)))
                      (multiple-value-bind (app res)
                          (traite appar2 (cons (nthcdr 5 prem_noeud) (cdr noeud)))
                          (values app (cons (append (list la_clef '@ etiq nouv_val_atom)
                                                    (first res))
                                            (rest res))
                          ))
                    )

                    ; si la  nouvelle valeur a affecter est une liste
                    (let* ((bidon (cons (third prem_noeud) (cons () ())))
                           (appar2 (cons bidon appariement)))
                      (multiple-value-bind (app res)
                          (traite appar2 (cons (fifth prem_noeud)
                                               (cons (nthcdr 5 prem_noeud)
                                                     (rest noeud))))
                          (let ((bidon2 (assoc (third prem_noeud) app)))
                            (rplaca (cdr bidon2) (first (first res)))
                            (rplacd (cdr bidon2) (cdr (first res)))
                            (values app (cons (append (list la_clef '@ (third prem_noeud))
                                                      (cons (first res)
                                                            (second res)))
                                              (nthcdr 2 res))
                                    )
                            )
                          )
                      )
                    )

                ;demande d'affectation
                (multiple-value-bind (app res)
                       (traite appariement (cons (nthcdr 3 prem_noeud) (cdr noeud)))
                       (values app (cons
                                     (append (cons la_clef
                                                   (cons '@
                                                         (cons (third prem_noeud)
                                                               (list (cdr (assoc (third prem_noeud) app))))))
                                             (car res))
                                     (cdr res))
                       )
                )
            )
        ; traitement des cas normaux (non affectation)
        (if (valeur_atomique prem_noeud)
            ; pas de traitement sur la valeur
            ; mais on propage malgre tout en cumulant les valeurs
            (multiple-value-bind (app res)
                                 (traite appariement
                                         (cons (cdr la_valeur_plus)
                                               (cdr noeud)))
                                 (values app
                                         (cons
                                          (append (list la_clef
                                                        (car la_valeur_plus)
                                                        )
                                                  (car res))
                                          (cdr res)
```

```
                                        )
                                       )
                                  )
            ; si c'est une liste on doit l'etudier en plus
            (multiple-value-bind (app res)
                                 (traite appariement
                                        (cons (car la_valeur_plus)
                                              (cons (cdr la_valeur_plus)
                                                    (cdr noeud)
                                                    )
                                              )
                                        )
                                 (values app
                                        (cons
                                         (cons la_clef (cons(first res) (second res)))
                                         (cddr res)
                                         )
                                        )
                                 )
            )
          )
        )
    (multiple-value-bind (app res)
                         (traite appariement (rest noeud))
                         (values app (cons () res)))
    )
  )
)

;;; but : determine si l'on va avoir une etiquette pour le premier trait du noeud considere

(defun nouv_affect_ou_affect (prem_noeud)
  (eq (second prem_noeud) '@ )
)

;;; but : determine si l'on va avoir une affectation d'etiquette avec le premier trait du
;;;       noeud considere

(defun nouv_affect (prem_noeud)
  (eq (fourth prem_noeud ) '<- )
)

;;; but : determine si l'on va avoir une affectation "atomique" avec le premier trait du noeud
;;;       considere
(defun nouv_affect_atomique (prem_noeud)
  (atom (fifth prem_noeud))
)

(defun valeur_atomique (prem_noeud)
  (atom (second prem_noeud))
)

;;; but : fonction generique de traitement d'une liste contenant des etiquettes,
;;;       c'est-a-dire que cette fonction renvoit une liste avec des structures partagees
;;;       qui sont "instanciees" (la valeur de la structure se trouve apres chaque etiquette)
;;;       et cela a partir d'une liste contenant les "references" a certaines etiquettes.
;;;       Donc, dans la liste resultat, on retrouve plus de symboles d'affectation ( <= )
;;;       Cette procedure est donc utilisee pour passer du format externe au format interne
;;;       et donc pour pouvoir traiter les listes avec etiquettes plus facilement.

(defun traitement (l)
  (multiple-value-bind (app lt)
    (traite () (list l))
    (values (first lt))
    )
)
```

```
(in-package "USER")

;;;                          UNIFICATION_S_P (grde petite etiquette)
;;;
;;; but : fonction generique de l'unification de deux listes ayant des structures partagees
;;;       l'unification se fait a partir d'une certaine etiquette.
;;;
;;; arg : grde et petite sont des listes trait-valeur pouvant contenir des etiquettes
;;;       etiquette est une etiquette de grande et indique l'endroit de grde ou commence
;;;       l'unification avec petite
;;;
;;; precond : - grde et petite sont des listes trait-valeur
;;;           - etiquette appartient a grde
;;;           - grde et petite n'ont pas de nom d'etiquette commun
;;;
;;; postcond : /
;;;
;;; resultat : une liste contenant l'unification de gdre et de petite si tout s'est bien passe
;;;            sinon, la fonction renvoit 'echec


(defun unif_s_p (grde petite etiquette)
  (multiple-value-bind (app result)
      (tr_unif () (list grde) petite etiquette)
      (if (eq app 'echec)
          'echec
          (first result)
          )
      )
  )
;;;                       UNIF_GEN (appar grde petite)
;;;
;;; but : fonction d'unification de deux listes "au meme niveau" avec mise a jour des
;;;       appariements
;;;
;;; arg : appar : liste des appariements etiquette-valeur-d'etiquette
;;;       grde et petite : listes trait-valeur pouvant contenir des etiquettes
;;;
;;; precond : - grde et petite sont des listes trait-valeur
;;;           - grde et petite n'ont pas de nom d'etiquette commun
;;;           - appar est une liste contenant 0 ou plusieurs appariement etiquette-valeur
;;;
;;; postcond : /
;;;
;;; resultat : - liste des appariements anciens plus ceux qui ont ete crees lors de l'unification
;;;              ou 'echec si l'unification a echouee
;;;            - la liste resultant de l'unification de grde et de petite, NIL est cas d'


(defun unif_gen (appar grde petite)
  (if (not (and grde petite)) ; la liste contient-elle un nil
      (if (eq grde nil)
          (values appar petite)
          (values appar grde))      ; renvoit de la liste ne contenant pas le nil
      (if (reduce #'(lambda (x y) (or x y)) (mapcar #'atom(list grde petite)))
          (if (eq grde petite)
              (values appar grde)
              (values 'echec ())
            )
          (let* ((premier (car grde))
                 (deuxieme (second grde))
                 (reste (nthcdr 2 grde))
                 (autre (member premier petite)))
            (if autre
                (let ((prem_autre (first autre))
                      (deux_autre (second autre)))
                  (cond
                   ; premier cas : les deux cles sont 'etiquetees
                   ((and (eq deuxieme '@ ) (eq deux_autre '@))
                    (let ((etiq_grde (third grde))
                          (etiq_petite (third autre))
                          (val_grde (fourth grde))
                          (val_petite (fourth autre))
                          )
                      (multiple-value-bind (app result)
                          (unif_gen appar val_grde val_petite)
                          (if (eq app 'echec)
                              (values 'echec ())
                              (let* ((nvappar1 (ajout_appar app etiq_grde result))
                                     (nvappar2 (ajout_appar nvappar1 etiq_petite result))
                                     (nv_petite (enlever_4_prem petite (subseq autre 0 4))))
                                (multiple-value-bind (appar_rest result_rest)
                                    (unif_gen nvappar2 (nthcdr 4 grde ) nv_petite)
                                    (if (eq appar_rest 'echec)
```

```
                                (values 'echec ())
                                (values appar_rest (append (cons premier
                                                                (cons '@
                                                                      (cons etiq_grde
                                                                            (list result))))
                                                           result_rest))
                                )
                            )
                        )
                    )
                )
            )
        ; deuxieme cas : seul la cle de la grande a une etiquette
        ((and (eq deuxieme '@ ) (not (eq deux_autre '@)))
         (let ((etiq_grde (third grde))
               (val_grde (fourth grde))
               (val_petite (second autre))
               )
           (multiple-value-bind (app result)
               (unif_gen appar val_grde val_petite)
               (if (eq app 'echec)
                   (values 'echec ())
                   (let* ((nvappar1 (ajout_appar app etiq_grde result)))
                     (multiple-value-bind (appar_rest result_rest)
                         (unif_gen nvappar1 (nthcdr 4 grde ) (remove (first autre)
                                                                     (remove (second autre)
                                                                             petite)))

                         (if (eq appar_rest 'echec)
                             (values 'echec ())
                             (values appar_rest (append (cons premier
                                                                (cons '@
                                                                      (cons etiq_grde
                                                                            (list result))))
                                                        result_rest))
                             )
                         )
                     )
                   )
               )
           )
         )
        ; troisieme cas : grde n'est pas 'etiquetee mais la deuxieme bien
        ((and (not (eq deuxieme '@)) (eq deux_autre '@))
         (let ((etiq_petite (third autre))
               (val_petite (fourth autre))
               (val_grde (second grde))
               )
           (multiple-value-bind (app result)
               (unif_gen appar val_grde val_petite)
               (if (eq result 'echec)
                   (values 'echec ())
                   (let ((nv_appar (ajout_appar app etiq_petite result))
                         (nv_petite (enlever_4_prem petite (subseq autre 0 4)))
                         )
                     (multiple-value-bind (app_rest result_rest)
                         (unif_gen nv_appar reste nv_petite)
                         (if (eq app_rest 'echec)
                             (values 'echec ())
                             (values app_rest (append (cons premier
                                                            (cons '@
                                                                  (cons etiq_petite
                                                                        (list result))))
                                                      result_rest))
                             )
                         )
                     )
                   )
               )
           )
         )
        ; quatrieme cas : aucune n'est etiquetee
        ((and (not (eq deuxieme '@)) (not (eq deux_autre '@)))
         (multiple-value-bind (nv_app nv_res)
               (unif_gen appar deuxieme deux_autre)
               (if (eq nv_app 'echec)
                   (values 'echec ())
                   (multiple-value-bind (app result)
                       (unif_gen appar deuxieme deux_autre)
                       (if (eq app 'echec)
                           (values 'echec ())
                           (multiple-value-bind (app_rest result_rest)
                               (unif_gen app reste (remove prem_autre
                                                           (remove deux_autre petite)))
```

```
                                        (if (eq app_rest 'echec)
                                            (values 'echec ())
                                            (values app_rest (append (cons premier (list result))
                                                                     result_rest))
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            ;; on n'a pas rencontre d'element ayant, dans petite, la cle = a 'premier
            (if (eq deuxieme '@)
                ; premier est suivis d'une etiquette => on poursuit et on remet la
                ; bonne valeur d'etiquette si elle a change
                (multiple-value-bind (nv_app nv_res)
                        (unif_gen appar (nthcdr 4 grde) petite)
                        (let ((res_assoc (assoc (third grde) nv_app)))
                          (if res_assoc
                              (values nv_app
                                      (cons premier
                                            (cons '@
                                                  (cons (third grde)
                                                        (append (list (cdr res_assoc))
                                                                nv_res)))))
                              (values nv_app
                                      (cons  premier
                                            (cons '@
                                                  (cons (third grde)
                                                        (cons (fourth grde)
                                                              nv_res)))))
                              )
                          )
                        )
                ; premier n'est pas suivis d'une etiquette => on poursuit le parcours
                (multiple-value-bind (nv_app nv_res)
                   (unif_gen appar reste petite)
                   (if (eq nv_app 'echec)
                       (values 'echec ())
                       (values nv_app (cons premier (cons deuxieme nv_res))))
                   )
                )
            )
        )
    )
)


;;; fonction unifiant une partie de structure de trait partiellement "appariee"
;;; et une autre qui n'a pas encore ete parcourue et met le resultat comme
;;; nouvel appariement

(defun unif (appar grde petite etiq)
  (multiple-value-bind (app nvgrde)
     (unif_gen  appar grde petite )
     (if (eq app 'echec)
         'echec
         (multiple-value-bind (m_a_j_app m_a_j_nvgrde)
            (m_a_j_etiq app nvgrde)
            (ajout_appar m_a_j_app etiq m_a_j_nvgrde)
            )
         )
     )
  )


;;; but : faire l'unification de deux liste comprenant des structures partagees
;;;       en fait d'une partie ( ce qui suit directement 'etiq) d'une avec
;;;       une autre complete
;;; preconditions : les deux structures ont subis la fonction 'traitement

  (defun tr_unif (appariement noeud petite etiq_unif)
    (if (not noeud)
        (values appariement noeud)
        (if (not (first noeud))
            (multiple-value-bind (app result)
                (tr_unif appariement (rest noeud) petite etiq_unif)
                (values app (cons () result))
                )
            (let* ((premier_noeud (first noeud))
```

```
                (la_cle (first premier_noeud)))
            (if (eq (second premier_noeud) '@)
                ;; cas ou l'on a une etiquette
                (let ((etiquette (third premier_noeud))
                      (val_etiq (fourth premier_noeud))
                      )
                  (if (eq etiquette etiq_unif)
                      ;; on doit effectuer l'unification
                      (multiple-value-bind (nvapp)
                         (unif appariement val_etiq petite etiq_unif)
                         (if (eq nvapp 'echec)
                             (values 'echec ())
                             (multiple-value-bind (app result)
                                (tr_unif nvapp
                                         (cons (nthcdr 4 premier_noeud)
                                               (rest noeud))
                                         petite
                                         etiq_unif )
                                (if (eq 'echec app)
                                    (values 'echec ())
                                    (values app
                                            (cons (append (cons la_cle
                                                                (cons '@
                                                                      (cons etiquette
                                                                            (list (cdr (assoc etiquette app))))))
                                                          (first result))
                                                  (rest result))
                                            )
                                    )
                                )
                             )
                         )
                      ;; cas sans unification
                      (if (atom val_etiq)
                       ; pas necessaire d'analyser l'atome
                       (multiple-value-bind (app result)
                          (tr_unif appariement
                                   (cons (nthcdr 4 premier_noeud)
                                         (rest noeud))
                                   petite
                                   etiq_unif)
; !! si echec de l'unification
                          (if (not (eq app 'echec))
                              (let ((assoc_etiq (assoc etiquette app)))
                                (if assoc_etiq
                                    (values app
                                            (cons (append (cons la_cle
                                                                (cons '@
                                                                      (cons etiquette
                                                                            (list (cdr assoc_etiq)))))
                                                          (first result))
                                                  (rest result))
                                            )
                                    (values app
                                            (cons (append (cons la_cle
                                                                (cons '@
                                                                      (cons etiquette
                                                                            (list val_etiq))))
                                                          (first result))
                                                  (rest result))
                                            )
                                    )
                                )
                              (values 'echec ())
                              )
                          )
                       ; analyse de la valeur de l'etiquette qui
                       ; est une liste
                       (multiple-value-bind (app result)
                          (tr_unif appariement
                                   (cons (fourth premier_noeud)
                                         (cons (nthcdr 4 premier_noeud)
                                               (rest noeud)))
                                   petite
                                   etiq_unif)
;! attention a la valeur de app = resultat de l'unification
                          (if (eq app 'echec)
                              (values 'echec ())
                              (let ((assoc_etiq (assoc etiquette app)))
                                (if (not assoc_etiq)
                                    (let ((bidon (cons etiquette (cons () ()))))
                                      (rplaca (cdr bidon) (first (first result)))
                                      (rplacd (cdr bidon) (cdr (first result)))
```

```
                                        (push bidon app)
                                       )
                                    )
                                   (values app (cons (append (cons la_cle
                                                                   (cons '@
                                                                         (cons etiquette
                                                                               (list (cdr (assoc etiquette app)))))
                                                             (second result))
                                                     (nthcdr 2 result)))
                                   )
                                  )
                                 )
                                )
                               )
                          )
                         ;; cas sans etiquette
                         (if (atom (second premier_noeud))
                           ;; traitement du reste du premier noeud et des autres noeuds
                           (multiple-value-bind (app result)
                              (tr_unif appariement
                                       (cons (nthcdr 2 premier_noeud)
                                             (rest noeud))
                                       petite
                                       etiq_unif)
                              (values app (cons (append (list la_cle
                                                              (second premier_noeud))
                                                        (first result))
                                                (rest result)))
                           )
                           ;; analyse de la liste, du reste du premier noeud et des
                           ;; autres noeuds
                           (multiple-value-bind (app result)
                              (tr_unif appariement
                                       (cons (second premier_noeud)
                                             (cons (nthcdr 2 premier_noeud)
                                                   (rest noeud)))
                                       petite
                                       etiq_unif )
                              (values app (cons (cons la_cle (cons (first result)
                                                                   (second result)))
                                                (nthcdr 2 result)
                                                )
                                       )
                           )
                         )
                        )
                       )
                      )
                     )
                    )
                   )
                  )
                 )
                )
               )
              )
             )
            )
           )
          )
         )
        )
       )
      )
     )
    )

;;; but : determine si x est un indicateur d'etiquette

(defun ind_etiq (x)
  (if (eq x '@)
      t
      ()
      )
  )

;;; ajout un nouvel appariement a une liste deja existante.
;;; si l'etiquette etait deja presente dans la liste, elle est d'abord eliminee

(defun ajout_appar (appar etiq val)
  (if (atom val)
      (cons (cons etiq val) (remove (assoc etiq appar) appar))
    (let ((bidon (cons etiq (cons () ()))))
      (rplaca (cdr bidon) (first val))
      (rplacd (cdr bidon) (cdr val))
      (cons bidon (remove (assoc etiq appar) appar))
      )
    )
  )


;;; fonction de mise a jour des etiquettes d'une liste grace a une liste d'appariements connus
;;; ou non

(defun m_a_j_etiq (app l)
  (if (not l)
      (values app ())
```

```
      (if (eq (first l) '@)
          ;; cas ou l'on a une etiquette (la cle est tj un atome => pas besoin de l'analyser)
          (let* ((etiq (second l))
                 (res_assoc (assoc etiq app))
                 )
            (if res_assoc
                (multiple-value-bind (app_res res_l)
                   (m_a_j_etiq app (nthcdr 3 l))
                   (values app_res (append (cons '@ (cons etiq (list (cdr res_assoc))))
                                           res_l))
                   )
                (let ((val_etiq (third l)))
                  (if (atom val_etiq)
                      (multiple-value-bind (app_res maj_res)
                         (m_a_j_etiq (ajout_appar app etiq val_etiq) (nthcdr 3 l))
                         (values app_res (append (cons '@ (cons etiq (list val_etiq)))
                                                 maj_res))
                       )
                      (multiple-value-bind (app_etiq maj_val)
                          (m_a_j_etiq app (third l))
                          (multiple-value-bind (app_res maj_res)
                              (m_a_j_etiq (ajout_appar app_etiq etiq maj_val) (nthcdr 3 l))
                              (values app_res (append (cons '@ (cons etiq (list maj_val)))
                                                      maj_res))
                              )
                          )
                      )
                  )
                )
            )
          ;; cas ou l'on n'est pas en presente d'une etiquette => distinguer ou le
          ;; premier element est un atome ou pas
          (if (atom (first l))
              (multiple-value-bind (app_res res_l)
                  (m_a_j_etiq app (rest l))
                  (values app_res (cons (first l) res_l))
               )
              (multiple-value-bind (app_nv maj_val)
                  (m_a_j_etiq app (first l))
                  (multiple-value-bind (app_res maj_res)
                       (m_a_j_etiq app_nv (rest l))
                       (values app_res (append (list maj_val) maj_res))
                       )
                  )
              )
          )
      )
  )


;;; fonction qui enleve une sous-sequence de longueur = 4 d'une liste

(defun enlever_4_prem (l sous_seq)
  (let ((fin (nthcdr (+ 4 (search sous_seq l)) l))
        (debut (reverse (nthcdr (+ 4 (search (reverse sous_seq) (reverse l))) (reverse l))))
        )
    (append debut fin)
    )
  )
```

BIBLIOGRAPHIE

- Bresman J. et Kaplan R. "Lexical-functional grammar : a formal system for grammatical representation", in "The mental representation of grammatical relations", édité par J. Bresman, MIT Press, Cambridge, 1981.

- Chow Y. et Roukos S. "Speech understanding using a unification grammar", IEEE, 1989.

- Deville G. "Modelization of task-oriented utterance in a man-machine dialogue system", Thèse de Doctorat de l'Université d'Anvers, 1989.

- Earley J. "An efficient context-free parsing algorithm", CACM 13(2), 1970.

- Fillmore C. "A proposal concerning English propositions", in Francis P. Dineen, S. J., ed., Monograph Georgetown University. Series on Languages and Linguistics, n° 19, pp. 19-33, 1966.

- Fillmore C. "The case for the case", in "Universals in Linguistics theory", Brech et Harms (eds) Holt, Reinhart et Winston, New-York, 1968.

- Fillmore C. "The case for case reopened", in "Syntax and Semantics : grammatical relations", Cole. P and Sadock J. éds, vol 8, Academic Press,New-York, 1977.

- Forh D., Carbonel N. and Haton J.P. "APHODEX, an acoustic-phonetic decoding expert system", International Journal Of Pattern Recognition and Artificial Intelligence, C.H. Chen ed., vol. 1 n°2, p 207-222, 1987.

- Gaiffe B. and Romary L. "Managing multimodal information and references in the Multiworks project", rapport interne CRIN n° 90-R-168, décembre 1990.

- Gazdar G., Klein E., Pullum G. et Sag I. "Generalized Phrase structure grammar", Oxford : Basil Blackwell, 1985.

- Godin P. "Aspects syntaxiques et sémantiques de la grammaire casuelle appliquée au français", U.C. Louvain, 1975.

- Grevisse M. "Le bon usage", Duculot 1969.

- Gross G. "Un projet d'étude systématique des noms composés du français", Université de Paris 13 et LADL, 1988.

- Haton J.P., Pierrel J.M., Perennou G., Caelen J. et Gauvain J.L. "Reconnaissance automatique de la Parole", Dunod Informatique, 1991.

- Hemphill C. et Picone J. "Speech recognition in a unification framework", IEEE, 1989.

- Kamp H. et Reyle U. "From discourse to logic", Actes de la Second European Summer School in Language, Logic and Information, Katholieke Universiteit Leuven, 30 juin - 10 août 1990.

- Kamp H. "Procedural and cognitive aspects of propositional attitudes contexts", Actes de la Third European Summer School in Language, Logic and Information, Universität des Saarlandes, Saarbrücken, 12 - 23 août 1991.

- Kay M. "Algorithm schemata and data structures in syntactic processing", CSL-80-12, octobre 1980.

- Kay M. "Parsing in functional Unification Grammar", Natural Language Parsing, D.R. Dowty, L. Kartunnen, and A. Zwicky, eds., 251-278, Cambridge, England : Cambridge University Press, 1982.

- Kay M. "An overview of natural langage understanding", Actes du séminaire INRIA. "Principes de la communication homme-machine : parole, vision et langage naturel", 28 mai - 7 juin 1985.

- Kita K., Kawabata T, Saito H. "HMM continuous speech recognition using predictive LR parsing", IEEE, pp. 703-706, 1989.

- Lea W.A. "Trends in speech recognition", Lea W.A. ed., Englewood Cliffs, N.J. : Prentice-Hall Inc., 1980.

- Marino J.B., Bonafonte A., Moreno A., Lleida E., Nadeu C. and Monte E. "Recognition of numbers by using demisyllables and Hidden Markov Models", Signal Processing V : Theories and Applications, L. Torres, E. Masgrau and M.A. Lagunas eds., Elsevier Sciences Publishers B.V., 1990.

- Mesli N. "Analyse et traduction automatique de constructions à verbe support dans le formalisme CAT2", Eurotra-D working papers n° 19, IAI Saarbrücken, juin 1991.

- Montague R. "The proper treatment of quantification in ordinary English", in Thomasson Richmond H éd., Formal Philosophy, New Haven, Connecticut : Yale University Press, pp. 247 - 270, 1974.

- Mousel P. "Syntaxe et sémantique dans un système de dialogue oral homme-machine finalisé en langage naturel", Thèse de Doctorat de l'Université de Nancy I, 1989.

- Pierrel J.M. "Etude et mise en oeuvre de contraintes linguistiques en compréhension automatique du discours continu", Thése d'état de l'Université de Nancy I, 1981.

- Pierrel J.M. "Dialogue oral homme-machine", Hermès 1987.

- Pierrel J.M. "Informatique et linguistique", cours de DEA à l'Université de Nancy I, 1990-1991.

- Pollard C. et Sag I. "Information-based syntax and semantics 1 : Fundamentals", Center for the Study of Language and Information, Stanford, CA, 1987.

- Pollard C. "Agreement, binding and control : IBSS2", in "Topics in constraint-based syntactic theory", Actes de la Third European Summer School in Language, Logic and Information, Universität des Saarlandes, Saarbrücken, 12 - 23 août 1991.

- Ravera S. "Information-based linguistics an head-driven phrase structure", Lecture Notes in AI, n° 476, Springer Verslag, 1991.

- Reyle U. "DRT : Nominal anaphora and tenses", Actes de la Third European Summer School in Language, Logic and Information, Universität des Saarlandes, Saarbrücken, 12 - 23 août 1991.

- Romary L. "Vers la définition d'un modèle cognitif autour de la représentation du temps dans un système de dialogue homme-machine", Thèse de Doctorat de l'Université de Nancy I, 1989.

- Roussanaly A. "DIAL : la composante dialogue d'un système de communication orale homme-machine finalisée en langage naturel", Thèse de Doctorat de l'Université de Nancy I, 1988.

- Sabah G. "L'intelligence artificielle et le langage", vol. 1 et 2, Hermès, 1988.

- Sharp R. "Implementing a formalism for multi-lingual MT", IAI Saarbrücken, juin 1988.

- Sharp R. "Modelling GB in the CAT2 machine translation

System", IAI Saarbrücken, juin 1990.

- Shieber S. "An introduction to unification-based approches to grammar", CSLI Lectures Notes, Stanford, 1986.

- Winograd T. "Language as a cognitive process", vol 1 Syntax, Addison Wesley, 1983.

- Woods W.A."Transition network grammars for natural langage analysis", CACM 3(10), 1970.