Intellectica, 1992 1/2, 13-14, pp. 129-184

Francis RENAUD

Générateurs automatiques d'analyseurs linguistiques : un point de vue linguistique

L'intelligence artificielle et la linguistique se sont développées jusqu'à ces dernières années de manière relativement indépendante, chacune en élaborant des techniques adaptées à ses visées propres. Mais les interactions de plus en plus fécondes entre ces deux domaines nous semblent porteuses d'un renouveau conceptuel fondamental de leur champ d'étude.

Pour faciliter le rapprochement de ces sciences, nous avons essayé d'adapter dans ce travail des techniques de traitement automatique des langues aux besoins spécifiques de la linguistique.

L'apport théorique de l'intelligence artificielle à la sémantique des langues nous semble essentiel. Le modèle linguistique que nous présenterons doit beaucoup aux travaux sur la représentation des connaissances et la théorie de la démonstration automatique.

Mais sous un angle plus pratique l'informatique peut aussi fournir au linguiste des moyens puissants permettant de manipuler des masses énormes et complexes de connaissances. Il est d'ailleurs surprenant de constater qu'encore aujourd'hui la plupart des linguistes ne disposent d'aucun outil informatique pour les aider dans leur travail. Leur habitude de travailler sur des grammaires extrêmement frustes constituées de

quelques règles, tient certainement à la difficulté de construire "à la main" des systèmes cohérents de plusieurs centaines de règles. Encore que l'approche générativiste voulant à tout prix expliquer le fonctionnement des langues par quelques principes universaux **simples** ait aussi une part de responsabilité.

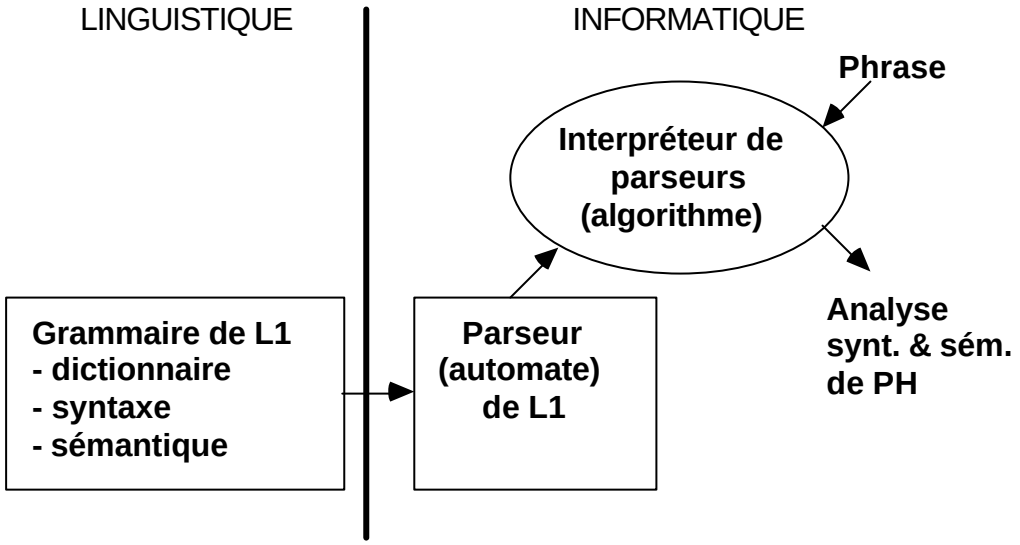
Les générateurs de parseurs[1] que nous proposerons par la suite visent à faciliter la construction de gros systèmes de règles cohérents rendant compte de fragments importants de la langue. Ils nous ont servi à construire une grammaire des expressions spatio-temporelles du chinois de plusieurs centaines de règles [Renaud 1988a]. Nous les utilisons actuellement à la construction d'une grammaire des expressions de temps du français de taille comparable.

Le parsage des langues naturelles fait appel à des compétences tant informatiques que linguistiques. Un bon projet de linguistique informatique doit combiner un interpréteur de parseur efficace tournant avec un automate basé sur une solide étude linguistique préalable. Le premier stade du travail grammatical aura intérêt à être totalement indépendant de toutes considérations informatiques. Ce n'est que dans un deuxième temps qu'on peut procéder à une *traduction* des connaissances linguistiques dans la syntaxe d'un interpréteur : réseau pour les ATN, automate pour Déredec [Plante 1988], listes d'attributs-valeurs etc...Mais cette deuxième étape est parfois délicate car elle suppose une bonne connaissance à la fois de la linguistique et des techniques de parsage (doit-on faire une analyse ascendante, descendante ou mixte, quels sont les éléments pointés quand le parseur est dans tel état etc...?).

---

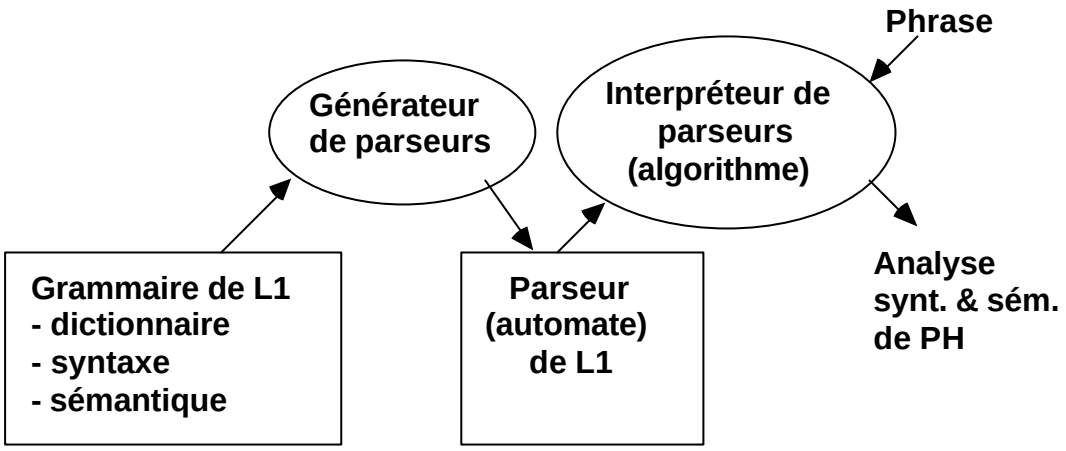
[1]Les termes d'analyse et d'analyseur étant par trop vagues en français et à défaut de mieux, nous parlerons désormais de *parsage* et de *parseurs*, suivant l'usage canadien francophone, pour désigner les techniques mathématiques et réserverons le terme d'analyse linguistique aux descriptions empiriques.

Avant de poursuivre, nous sentons le besoin de fixer un peu mieux une terminologie particulièrement flottante. Nous ne parlerons de **parseur** (ou d'**automate**) que pour désigner les connaissances linguistiques traduites dans la syntaxe de l'**interpréteur de parseur**.



Le passage du domaine linguistique au domaine informatique se fait pour l'instant "à la main" mais il nous semble possible de l'automatiser.

Il s'agit de concevoir un algorithme qui prenne les données linguistiques et les transforme en représentations directement utilisables par les interpréteurs de parseurs.



Deux genres de techniques sont concevables :

- soit le générateur travaille globalement sur l'ensemble de la grammaire et construit en une seule fois le parseur (appelé dans ce cas "table de parsage").
- soit le générateur travaille localement, règle après règle, et construit éventuellement en plusieurs fois le parseur.

Dans la première approche, le gain d'efficacité au moment du parsage se paye au prix de la complexité de la construction de la table de parsage. La construction d'un tel parseur n'est donc intéressante que si l'on compte faire de nombreuses analyses avec le même parseur.

Par contre, les parseurs du second type, peu efficaces mais faciles à mettre en oeuvre, sont particulièrement utiles au linguiste lors de la phase de mise au point d'une grammaire.

Chacune de ces approches est suffisamment générale pour permettre de construire aussi bien des parseurs descendants qu'ascendants.

Depuis longtemps de telles techniques existent en informatique. Mais elles ne pourront être utilisables en linguistique qu'*à condition de disposer d'un modèle linguistique bien formalisé*. Le fossé entre linguistique et informatique tient essentiellement au fait que le linguiste ne fait pas assez de théorie et que l'informaticien néglige les données linguistiques. Notre effort vise à combler ce fossé en définissant

précisément un modèle linguistique formel et en montrant qu'il est possible de lui adapter les grandes techniques de parsage classiques.

On peut cependant se demander s'il est possible de travailler à la confluence de ces deux sciences sans sacrifier la méthodologie propre de chacune. En effet, la linguistique, science empirique basée sur l'observation ne partage pas ses méthodes avec l'algorithmique, science formelle a priori. Il paraît donc méthodologiquement préférable de donner les connaissances linguistiques au programme sous forme "déclarative" plutôt que de les représenter directement à l'intérieur du programme. Néanmoins on voit régulièrement des chercheurs préférer la seconde démarche pour des raisons d'efficacité. Le traitement de grosses masses de connaissances est un des problèmes les plus ardus de l'IA. Comment pouvoir espérer en venir à bout ? Un effort théorique sérieux en linguistique nous paraît plus utile que des recherches de calculs toujours plus astucieux.

Nous cherchons donc à construire un modèle linguistique ayant une bonne assise empirique et doté de propriétés formelles suffisamment "solides" pour autoriser des *traductions* dans les langages formels de chaque parseur. *Les techniques informatiques doivent donc être les servantes de la linguistique et non l'inverse.* Nous ne dirons pas comme certains que "le parseur est la grammaire" mais plutôt que "le parseur est une traduction particulière (parmi d'autres) de la grammaire".

Enfin, il nous semble que le fait de préserver l'indépendance des connaissances linguistiques à l'intérieur des programmes facilite le travail linguistique de mise au point des grammaires. En effet, la construction d'une grammaire un tant soit peu complexe est un travail de longue haleine, se faisant pas à pas, par approximations successives et nécessitant parfois de revenir sur des décisions prises antérieurement. Un dépendance trop profonde des programmes à l'égard des données linguistiques obligerait à remettre trop souvent en cause les programmes.

A l'inverse, leur indépendance devrait permettre au linguiste de s'affranchir de la collaboration de programmeurs.

Nous proposons dans une première section un modèle linguistique combinant des règles de réécritures algébriques, des structures d'unification et des règles sémantiques dans un langage fonctionnel typé. Dans une deuxième section, nous présentons des parseurs ascendants et descendants pour ce genre de grammaire ainsi que leur générateurs.

Cet article a été écrit en 1988 et présenté à *Intellectica* en 1989.

Nous voudrions remercier M. Philip Miller qui en a fait une relecture très attentive. Nous avons pris de notre mieux en compte toutes ses suggestions.

1. Le modèle linguistique

Il est constitué par un ensemble de règles permettant d'effectuer une analyse syntaxique et sémantique intégrée. Les règles de réécritures des **grammaires algébriques**[2] sont particulièrement bien adaptées pour mener des analyses locales. Par contre, elles sont beaucoup plus difficiles à manier pour traiter les phénomènes de longues portées (tels que relatives, accord...). Aussi est-on obligé de les compléter par des **structures de traits**. Nous nous inspirerons de la théorie des traits de Gazdar [Gazdar et al. 1988] mais n'adopterons pas son langage de contrainte ni la voie prise dans GPSG [Gazdar et al. 1985] où une partie des traits est héritée et une autre instanciée. Depuis les travaux sur les grammaires d'unification [Kay 1985, Shieber 1986], il semble beaucoup plus simple de considérer tous les traits comme hérités.

A chaque règle syntaxique $X_0 \varnothing X_1\ X_2\ ...X_n$ est associée une règle sémantique permettant de calculer le sens de X_0 en fonction des sens de

[2]Dites en anglais "context-free". Nous adoptons le terme de "grammaire algébrique" à "l'Ecole Française" de théorie des langages formels animée par M. Schützenberger et M. Nivat. Cette école a selon Autebert, acquis une réputation mondiale [Autebert 1987].

X_1 X_2 ...X_n. Afin de pouvoir recourir aux techniques les mieux établies de la *déduction*, nous choisissons de représenter le sens des phrases en *logique des prédicats.* Toutes les manipulations sur les représentations du sens entre le niveau lexical et le niveau phrastique seront effectuées dans un **langage fonctionnel typé** (un lambda-calcul étendu par des fonctions récursives, cf. Renaud 1988a,b).

Voyons sur un exemple simple comment une règle se présente. Elle comporte quatre parties :

(R1) 1. V_1 ∅ vt N_2
2. vt:clasv = actif
3. V_1 vt
4. λ(a x) .N_2 (λ (y) vt (a, x, y))

(1) une règle de réécriture du groupe verbal (les catégories lexicales sont en minuscules)
(2) un test. La règle (R1) n'est applicable que si le verbe transitif vt possède un trait "clasv" de valeur "actif".
(3) signifie que la racine V_1 est affectée de tous les traits de la tête verbale vt.
(4) est une lambda-expression calculant la représentation du sens de V_1 en fonction des représentations du sens de vt et de N_2. Si par exemple, la représentation du sens du (vt "voit") est le prédicat VOIR et celle du (N_2 Marie) est λ(P) P(m) alors par substitution et bêta-réduction dans le λ-terme de (R1)4, on obtient[3]

λ (a x) ((λ (P) P(m)) (λ (y) VOIR(a, x ,y)))
=> λ (a x) VOIR(a, x, m)
avec "a" un index identifiant l'événement.

[3]Nous adoptons les conventions de notations suivantes : m est une constante de type individu ind ; x et y sont des variables de type individu ; P est une variable de type ind∅prop ; a est une variable de type événement.

Certaines règles comportant plusieurs occurrences de même catégorie (par exemple V1 ∅ vaux V1 a deux occurrences de V1), nous devons modifier la notation pour éviter toute confusion. Nous associerons conventionnellement à une règle de production $X_0 \varnothing X_1 X_2 ... X_n$ d'une grammaire algébrique GA = (CAT, LEX, P_0) avec CAT : ensemble des catégories, LEX : ensemble des catégories lexicales (les terminaux de GA), P_0 : ensemble des règles de production et

X_0 CAT \ LEX (catégories non-lexicales) et Xi CAT

la liste des structures de traits $*U=(*U_0,*U_1, ...*U_n)$

avec $*U_i$ structure de traits de X_i i=0,1..n

la liste des représentations du sens $*S = (*S_0,*S_1, ...*S_n)$

avec $*S_i$ représentations du sens de X_i i=0,1..n

L'exemple précédent s'écrira maintenant :

(R1) 1. V_1 ∅ vt N_2

2. $*U_1$: clasv = actif

3. $*U_0$ $*U_1$

4. $*S_0 <<= \lambda$ (a x) $*S_2$ (λ (y) $*S_1$(a, x, y))

Les équations sur les structures de traits sont de deux types

- soit elles donnent les conditions d'application d'une règle
- soit elles assurent la transmission de structures de traits d'une catégorie à une à une autre.

Voyons maintenant comment construire une dérivation dans ces grammaires. Considérons par exemple une dérivation ascendante.

La grammaire algébrique GA comporte comme *ensemble des terminaux* LEX l'ensemble des catégories lexicales (telles que vi, vt, n, art....). Nous lui associerons à un niveau supérieur, une **grammaire généralisée G** traitant non seulement des structures syntaxiques mais aussi des structures de traits et des représentations sémantiques.

A chaque acception d'une entrée lexicale est associé un triplet formé d'une catégorie syntaxique, d'une structure de traits et d'une représentation du sens. D'une manière générale, partout où dans la

grammaire GA on avait une catégorie, on aura dans la grammaire G un "triplet d'analyse" (catégorie, trait , sens).

Pour dériver par exemple dans G la phrase :

Jean voit Marie.

on commence par appliquer le dictionnaire sur cette séquence :

(npr, () ,j) (vt ,(clasv:actif, tc:fini), λ(a x y)VOIR(a, x, y)) (npr, (), m)

puis on applique "en arrière" la règle :

(R2). N2 ∅ npr
()
()
$*S_0$ λ(P) P($*S_1$)

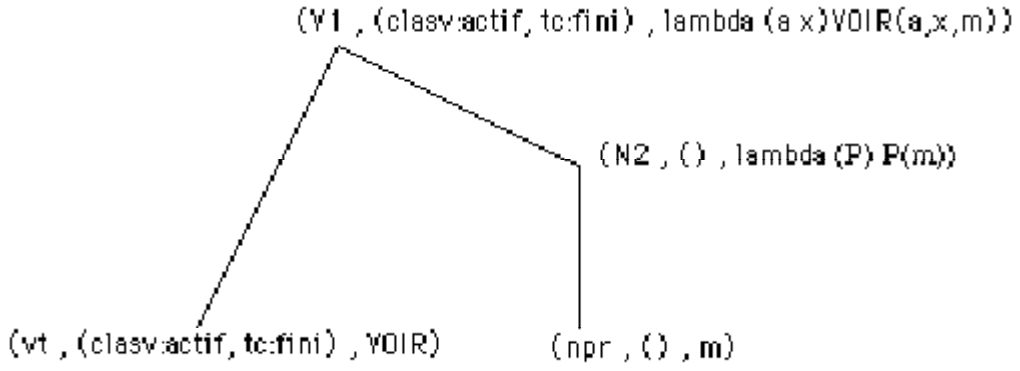
sur le dernier triplet. Celui-ci devient (N2, (), λ(P)P(m)) puisque $*S_1$ représentant le sens de (npr, (), m) vaut m. Soit

(SQ) (npr, (), j) (vt ,(clasv:actif, tc:fini), λ(a x y)VOIR(a, x, y)) (N2, (), λ(P) P(m))

On peut ensuite appliquer "en arrière" la règle (R_1) sur les deux derniers triplets. Le test (R1)2 est satisfait puisque le triplet (vt,..) a un trait clasv actif. Les instructions (R1)1,3,4 indiquent comment construire la catégorie, la structure de traits et le sens du nouveau triplet à partir des deux anciens triplets. Ce qui donne :

(npr, (), j) (V_1, (clasv:actif, tc:fini), λ(a x) VOIR(a, x, m))

On voit ainsi comment pas-à-pas construire la dérivation :



A cette présentation traditionnelle en linguistique, nous pourrions opposer une présentation mathématique plus homogène dans laquelle les règles opéreraient sur les triplets. En effet, si nous écrivions la règle (R_1) ainsi :

(RT1) (vt,*U_1,*S_1) (N2,*U_2,*S_2) ∅ (V1,*U_1,λ (a x) *S_2(λ (y) *S_1(a,x,y)))

Pour effectuer un pas de la dérivation, il suffirait d'unifier[4] le membre gauche de la règle avec deux triplets de la séquence (SQ) par la substitution σ :

σ(*U_1) = (clasv:actif, tc:fini)

σ(*S_1) = λ (a x y)VOIR(a,x,y)

σ(*U_2) = ()

σ(*S2) =λ (P) P(m)

Puis les deux triplets seront remplacés par le membre droit de la règle[5] sur lequel nous aurons appliqué σ.

Pour construire *les arbres de dérivation* dans le premier élément des triplets, il suffirait d'écrire (R1) ainsi :

(*A_1,*U_1,*S_1) (*A_2,*U_2,*S_2) ∅ (V1(*A_1,*A_2), *U_1, λ (a x)

*S_2 (λ (y) *S_1(a,x,y)))

avec les conditions :

racine(*A_1) = vt

racine(*A_2) = N2

*U_1:clasv = actif

---

[4]Unifier deux termes e_1 et e_2 au sens du "pattern matching", c'est trouver une substitution σ (de variables par des termes) telle que σ(e_1) = σ(e_2)

[5]On peut effectuer un pas de dérivation u t_1...t_n w u t_0 w (sur une séquence de triplets) si il existe une règle p_1...p_n ∅ p_0 (du type de (RT1)) et une substitution σ telle t_1= σp_1 ,... t_n = σp_n et t_0 = σp_0.

Le pas de dérivation serait :
(npr(jean),(),j) (vt (voir), (clasv:actif,tc:fini), λ(a x y)VOIR(a,x,y)) (N2(npr(marie)), (), λ(P) P(m)) (npr(jean),(),j) (V1(vt (voir), N2(npr(marie))), (clasv:actif,tc:fini), λ(a x)VOIR(a,x,m))

On obtient ainsi en fin de dérivation l'arbre de dérivation de la phrase dans le premier élément du triplet.

Nous avons donc là, des grammaires de semi-Thue sur des triplets appartenant à des algèbres hétérogènes (nous donnons quelques indications sur ces structures un peu plus bas).

Nous remarquerons par ailleurs, qu'il n'y a pas de raison de limiter les analyses à des triplets - nous pourrions par exemple introduire un quatrième champ contenant des informations phonétiques ou mélodiques.

Nous donnons en annexe plusieurs exemples de ce genre de grammaires.

1.1. Les structures de traits

Nous reprenons la définition de Gazdar et al. (1988) : une structure de traits (par abr. ST) est un ensemble de paires trait-valeur (tr,val), notée aussi tr:val, où la valeur val peut être soit une constante soit une autre structure de traits.

Exemple :
α = (cln:hum, acc:(nb:sg, gr:fem))
avec pour traits : cln (classe), acc (accord), nb (nombre), gr (genre)
La structure de traits α associe à chaque trait de type 0 une constante
(cln∅hum, nb∅sg, gr∅fem)
à chaque trait de type 1 une structure de trait (acc∅(nb:sg, gr:fem))
α peut donc être considéré comme une *fonction partielle* sur l'ensemble des traits TRT. Nous l'étendrons en une fonction totale en attribuant la valeur () pour les traits "non-définis".

On écrira donc son application aux éléments de son domaine ainsi :

α(cln) = hum

α(acc) = (nb:sg,gr:fem)

α(acc)(nb) = sg

Par convention on notera aussi α(acc)(nb) par α(acc, nb) ou α:acc:nb en notation préfixée. Nous appellerons acc:nb un chemin dans α.

La syntaxe des structures de traits est donc

STRU $\varnothing$ (PAIR*)

PAIR $\varnothing$ trait : VAL

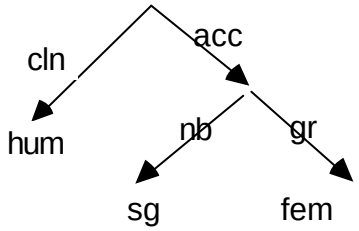
VAL $\varnothing$ cst / STRU

Il est possible de définir sur l'ensemble des structures de traits STRU une relation d'ordre 1 (de subsomption). La borne supérieure associée à cette relation est appelée l'unification " (cf. définition dans Shieber 1986).

α 1 β ssi $\beta = \alpha$ " β

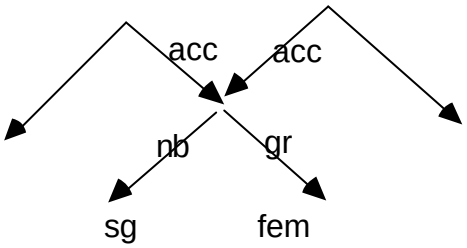
Un moyen simple de se représenter ces structures de traits est de les considérer comme des graphes orientés sans circuit[6].

Par exemple, la ST= (cln:hum, acc:(nb:sg,gr:fem)) se représente par



[6]Shieber1986 et Gazdar et al. 1985 imposent à ces graphes de ne pas posséder de circuit (on ne peut pas revenir à un sommet en suivant un chemin orienté). Cette contrainte est nécessaire pour empêcher que l'algorithme d'unification ne boucle indéfiniment sur des équations du type R = (t:R). Cependant Aït-Kaci & Lincoln [1988] ont réussi une généralisation de la notion de structures de traits qui lève cette contrainte.

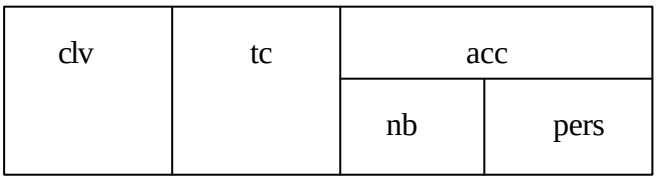
Les traits servent à étiqueter les arcs orientés vers les noeuds qui portent les valeurs associées. Ces graphes ne sont pas des arbres lorsque deux structures partagent la même sous-structure :

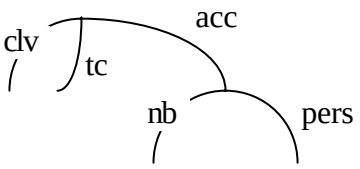


Des syntagmes de contenu lexical différent mais de même catégorie syntaxique seront associés en général à des structures de traits de même "ossature"- i.e. qu'ils porteront les mêmes traits affectés de valeurs différentes. Nous introduirons la notation suivante de ces "ossatures" : à la structure de traits

(clv:mod,tc:fini,acc:(nb:pl,pers:3))

on associe le **schéma de traits**





Il est défini par l'*ensemble* de tous les chemins (clv, tc, acc, acc:nb, acc:pers).

La manipulation des structures de traits s'inspire des grammaires d'unification. Cependant, l'association dans notre grammaire d'une règle sémantique à une règle syntaxique, nous a amené à modifier la présentation des opérations.

En effet, la non directionnalité des arguments dans les grammaires d'unification (comme en Prolog) est source de quelques difficultés lorsque l'on désire définir des tests purs. C'est-à-dire qu'une équation

ST:tr1 = a1 (1)

va s'interpréter de deux façons :

- soit la structure ST possède un trait tr1 dans son domaine et (1) teste alors si la valeur de tr1 dans ST est bien a1 :

(tr1:a1...):tr1 = a1 est vrai

(tr1:b1...):tr1 = a1 est faux

- soit ST ne possède pas de traits tr1 et (1) retourne alors une nouvelle spécification de ST :

si ST est (cat:v)

(1) va donner

pour ST la valeur (cat:v, tr1:a1)

C'est-à-dire que l'unification =, au lieu d'être un opérateur de borne supérieure, construisant une ST à partir de deux ST, modifie le (les) opérande(s).

Ce procédé a l'inconvénient d'*interpréter l'absence d'un trait comme un ordre d'affectation de ce trait.*

Cette assimilation de deux opérations différentes (un test et une affectation) peut être très gênante dans les calculs sémantiques. La nécessité de les distinguer clairement apparaît dans des problèmes comme le suivant.

Très souvent une même structure syntaxique est associée à plusieurs interprétations sémantiques : une même expression de lieu peut s'interpréter comme un lieu d'aboutissement ou un lieu de déroulement suivant la classe du verbe. En contrôlant l'interprétation sémantique par des tests sur les structures de traits on peut ainsi fusionner plusieurs règles possédant la même règle de réécriture mais associée à des opérations sémantiques différentes.

Voyons par exemple le cas du complément de temps en français. La parenté syntaxique des compléments d'instant, d'intervalle et des subordonnées en *quand*, est manifeste :

Marie est venue	à 8 heures	(1)
	en mai	(2)
	quand Paul était parti	(3)

Comme l'interprétation itérative ou singulative de ces compléments, dépend entre autre du temps verbal, il est préférable de suspendre partiellement leur interprétation jusqu'au niveau V2 :

(R3) V2 ∅ V1 COMP-t

là où on peut tester si le trait tc de temps du V1 appartient à l'accompli :

V1:tc {pc,ps} (pc passé composé, ps passé simple, liés au singulatif)

ou bien à un temps de l'itératif tel que le présent ou l'imparfait :

V1:tc {pres,impf}

De plus la règle de combinaison des représentations du sens de V1 et des COMP-t va aussi différer suivant qu'on a affaire à des instants, des intervalles ou des subordonnées (Renaud 1988a). La règle sémantique associée à la règle (R3) va donc comporter un test du genre :

si V1:tc {pc,ps} et COMP-t:clprep = à alors ..."interprétation de (1)"

Mais pour que ce test soit faux non seulement avec une préposition différente (telle qu'en (2)), mais aussi avec une conjonction (comme en (3)), il faudrait que la conjonction de subordination *quand* soit marquée d'une manière complètement ad hoc d'un trait prépositionnel clprep quelconque différent de "à".

Nous préférons pour notre part, ne pas devoir marquer explicitement toutes les absences de propriétés, mais plutôt de considérer qu'une absence de trait équivaut à une absence de propriété.

A cette fin nous distinguerons l'opérateur d'unification traditionnel =, d'un opérateur d'identité noté == servant uniquement à des tests.

Exemples :

L'unification (t1:a1):t2 = a2 modifie la ST d'origine et retourne

(t1:a1, t2:a2)

alors que l'opération d'identité (t1:a1):t2 == a2

fourni un test pur qui retourne en l'occurrence la valeur faux, comme l'aurait fait

(t1:a1,t2:a3):t2 == a2 .

Nous généraliserons ce procédé et définirons un test comme un terme fonctionnel comportant une fonction retournant vrai ou faux appliquée à des arguments formés de structures de traits, de constantes ou d'ensemble de constantes. Les tests ne modifient pas les structures de traits ST sur lesquelles ils s'appliquent.

Exemples de tests :

. nul(ST:clv)

teste si le trait clv est défini dans la structure de traits ST. Cette opération se fait en calculant d'abord la valeur de ST:clv puis en vérifiant que cette valeur est nulle. Ainsi nul((tc:pc):clv) procède à l'évaluation de (tc:pc):clv soit en l'occurrence le vide (), et retourne vrai puisque nul(()) est satisfait.

. l'identité

ST:tc==impf

est vraie si l'évaluation de ST:tc donne "impf" sinon est faux.

. l'appartenance

appart(ST:tc, {pc,ps})

teste si le l'évaluation de ST:tc appartient à l'ensemble {pc,ps}.

Ce test est équivalent à la disjonction

(ST:tc==pc) ≈ (ST:tc==ps)

. l'unifiabilité

unifiable(ST_1:acc, ST_2:acc)

est vrai si l'évaluation de ST1:acc et ST2:acc donne deux graphes unifiables

Enfin, ces tests fonctionnels sont combinables par les fonctions booléennes traditionnelles : ET, OU, NON ...

nul(ST:clv) OU appart(ST:tc, {pc,ps})

L'ensemble de ces tests définissent un **langage de contrainte :**

test(vrai) est vrai

test(T1 OU T2) est vrai ssi test(T1) est vrai ou si test(T2) est vrai.

...

test($F(A_1,...A_n)$) est vrai pour une fonction récursive F (retournant vrai ou faux et définie par ailleurs dans le langage fonctionnel LFT), ssi l'application de F à la valeur des arguments $A_1,...A_n$ donne vrai.

La définition des tests n'est cependant pas suffisante pour assurer la **transmission des traits** dans l'arbre de dérivation. Depuis les premiers travaux de Gazdar sur le sujet, on connaît tout l'intérêt de disposer de ces techniques pour exprimer les contraintes linguistiques de longue portée dans les langages "indépendants du contexte" qui sont comme leur nom l'indique par essence "localistes". Martin Kay a été le premier en 1985 a introduire l'outil fondamental de transmission des traits qui est l'*unification*.

Un autre problème rencontré par l'unification est celui de la disjonction. Si l'on veut marquer les groupes verbaux du trait des verbes composés, dans les exemples suivants :

Jean est venu
Jean était venu
...
Jean doit venir
Jean devait venir
...

on doit introduire une suite d'alternatives :

V1 ∅ VAUX V1

(*U_1:tc=pres ↔ ∗U_2:tc=ppe ↔ *U_0:tc=pc)

≈ (*U_1:tc=impf ↔ ∗U_2:tc=ppe ↔ *U_0:tc=pqpf)

≈ (*U_1:tc=ps ↔ ∗U_2:tc=ppe ↔ *U_0:tc=pa)

≈ (*U_1:tc=fut ↔ ∗U_2:tc=ppe ↔ *U_0:tc=fa)

≈ (*U_1:tc=pres ↔ ∗U_2:tc=infini ↔ *U_0:tc=pres)

≈ (*U_1:tc=impf ↔ ∗U_2:tc=infini ↔ *U_0:tc=impf)

≈ ....

La première instruction indique que le passé composé est formé d'un auxiliaire au présent et d'un verbe au participe passé etc...

Les temps des VAUX et du verbe principal étant montés à partir du lexique jusqu'au niveau du V1, la tentation est forte de les combiner directement par une fonction :

(R4) V1 ∅ VAUX V1

U_0:tc tcomp(U_1:tc, ∗U_2:tc)

tcomp est une fonction définie dans le langage fonctionnel LFT qui permet un *calcul direct et rapide* du résultat :

X	Y	tcomp(X,Y)
pres	ppe	pc
impf	ppe	pqpf
...	...	...

On définit comme un opérateur d'unification ordinaire lorsque ses deux membres sont des chemins dans des ST :

*U_0:Ch_0 *U_1:Ch_1 si chemin(Ch_1,*U_1,Val) et chemin(Ch_0,*U_0,Val)

(la valeur de *U_1(Ch_1) est transmise à *U_0(Ch_0))

Par contre, si le membre droit de est un terme fonctionnel composé :

*U F(A_1,A_2,...,A_n)

chacun des arguments A_i est évalué, puis le résultat de l'application de F sur leur valeur est unifié à *U (comme en (R_4)).

A la différence des tests, ces fonctions ne retournent pas vrai ou faux mais une nouvelle structure de traits ou une constante (dans le cas *U:tc $F(A_1,A_2,...,A_n)$).

Remarque : Depuis les travaux de Kay 1979, Bresnan 1982, et Shieber 1986 de nombreuses tentatives pour trouver un fondement théorique solide à l'unification ont vu le jour. Nous pourrions citer entre autres : Kasper & Rounds 1986, Aït-Kaci 1986, Johnson 1988, Dawar et Vijay-Shankar 1989. Mais c'est un article tout récent qui a emporté notre adhésion la plus complète sinon notre enthousiasme. Smolka (à paraître) donne enfin "la" sémantique tant cherchée. Fin 90, soit deux ans après les premières version du présent article, nous avons implanté les grammaires de contraintes de Smolka. Grâce à la négation et à la disjonction, il devient possible de traiter par une même opération les tests et les affectations sans devoir marquer les absences de propriété par des valeurs ad hoc explicites. Cependant la nature du fondement théorique de l'unification n'affecte pas la structure du modèle linguistique et des algorithmes que nous présentons ici-même.

1.2. Les langages fonctionnels typés

Nous avons choisi comme métalangage de représentation du sens les langages fonctionnels typés LFT. Ils sont constitués par un lambda-calcul typé étendu par des fonctions récursives (pour une définition précise des LFT cf. Wikström 1987, Paulson 1987 ; pour une motivation linguistique cf. Renaud 1988b).

Les types TYP sont construits à partir de types élémentaires :

TYP_e = {évén, prop, ind...} (évén désigne les événements, prop les propositions, ind les individus...),

si α, β TYP alors $(\alpha \varnothing \beta)$ TYP (formation de types fonctionnels),

et si α TYP alors list(α) TYP (formation de type de listes[7]).

Les expressions bien formées d'un LFT sont soit des termes typés TER_α soit des définitions DEF.

Les **termes** se construisent à partir des termes primitifs (des constantes ou des variables CST_α " VAR_α 1 TER_α) par les opérations suivantes

- application : f $TER_{(\alpha\varnothing\beta)}$ t TER_α f(t) TER_β.
- sélection : t TER_{prop} u, v TER_α si t alors u sinon v TER_α
- abstraction : x VAR_α t TER_β (lambda (x) t) $TER_{(\alpha\varnothing\beta)}$

En vue d'une utilisation linguistique ultérieure, nous introduirons un ensemble de variables *distinguées* S = {*S_i : i N} qui serviront à transmettre les représentations du sens dans l'arbre de dérivation (de même que les variables distinguées *U_i transmettront les structures de traits).

Les **définitions** sont des termes auxquels on a attribué un nom :

c CST_α t TER_α def c est t DEF (def pour définition)

C'est par leur intermédiaire qu'on peut introduire n'importe quelle fonction récursive dans les termes :

si t est un terme contenant l'atome f_1 défini par d = def f_1 est t_1 alors deb t où d est un terme de même type que t

t TER_α d DEF deb t où d TER_α (deb pour début)

Les définitions sont donc incluses dans les termes, mais pour la clarté de la présentation, nous préférons les donner à part, à la suite des règles.

---

[7]Tous les éléments d'une liste de type list(α) ont le même type α. Si l'on veut des éléments de types différents, il faut définir les paires et le n-uplets cf Wikström 1987,Paulson 1987.

Voyons quelques exemples.

Certaines règles grammaticales ne font intervenir que les opérations de lambda-calcul : l'application, la sélection et l'abstraction. Ce fut le cas des règles (R1) et (R2) données précédemment.

Par contre, si l'on désire construire des représentations un peu plus complexes, il peut être très pratique d'introduire des fonctions récursives.

Soit par exemple, l'analyse de l'expression de temps "en mai 1789" dans "Ils se sont réunis en mai 1789". Supposons que l'on choisisse de représenter les instants par des triplets (an,mois,jour) et les intervalles par des paires de triplets représentants le début et la fin de l'intervalle. Alors le mois de mai, 5 ème mois de l'année, sera représenté par l'intervalle commençant le 1er mai et se terminant le 31, soit ((5,1),(5,31)), alors que l'année 1789 recevra la représentation

((1789,1,1),(1789,12,31)).

Maintenant pour obtenir le syntagme "mai 1789" il faut une règle du genre

(R5) DT ∅ mois NU

$*S_0$ <<= comb($*S_1$,$*S_2$)

avec mois ∅ janvier / février..., NU nombre (l'année),

où la fonction récursive "comb" appliquée aux interprétations de "mai" et de "1789" donne l'interprétation de "mai 1789" :

soit

comb(((5,1),(5,31)), ((1789,1,1),(1789,12,31))) = ((1789,5,1),(1789,5,31))

avec la définition suivante dans le LFT :

```
def  comb est
   lambda (x,y) si  long(car(x)) = long(car(y))  alors  x
   si caar(y)=caadr(y) alors  list(cons(caar(y), car(comb(x,cdr(y)))),
                                   cons(caar(y),cadr(comb(x,cdr(y)))))
```

long calcule la longueur d'une liste; car, cdr, cons et list sont définis comme en LISP (pour plus de détails voir notre article de 1988b).

Le fait de pouvoir introduire des fonctions récursives (telles que "comb") dans les représentations sémantiques permet en quelque sorte de "tout faire quand on veut et où on veut".

Deux types d'opérations sont effectuées sur les expressions du LFT.

La **bêta-réduction** qui est une simple substitution[8] :

$(\lambda(x)\ t)\ e\ \ 5\beta\ \ t[x \blacklozenge e]$

et l'**évaluation**[9] qui "calcule effectivement la valeur de l'expression".

Par exemple

$((\lambda(x\ y)(+\ x\ y))\ 2\ 3)\ 5\beta\ \ (+\ 2\ 3)$

$(\text{eval}\ \ ((\lambda(x\ y)(+\ x\ y))\ 2\ 3)\ ) = 5$

Dans la grammaire définie plus bas, les termes de type prop TER_{prop} représentant le sens des phrases seront des expressions de logique des prédicats. Il faudra donc bien veiller à ne pas évaluer des expressions telles que VOIR(a,jean,marie), puisque VOIR ne recevra pas de définition dans le LFT. Par contre, comme nous l'avons vu, pour aboutir à une expression élégante dans TER_{prop}, il est souvent nécessaire d'introduire dans les calculs intermédiaires des fonctions sur les TER_{α}, définies dans le LFT. Ces fonctions elles, seront évaluées dès que leurs arguments seront spécifiés. Nous venons de voir le cas de la fonction comb qui combine immédiatement les intervalles. Mais parfois aussi, l'évaluation des fonctions du LFT peut être différée. Par exemple, le lexème "le lendemain" recevra une interprétation du genre $(\lambda(r)$ adjour(1,r)), où r est le point de référence temporel. L'expression adjour(1,r) ne sera évaluée qu'à partir du moment où dans les calculs, r deviendra connu. Si par exemple, on apprend au cours de l'analyse que r = le 1er octobre 1988 = (1988,10,1) alors l'évaluation de adjour(1,(1988,10,1)) donnera (1988,10,2) soit le 2 octobre 1988. Par

---

[8] $t[x \blacklozenge e]$ signifie qu'on substitue le terme e aux variables libres x dans t.

[9] L'évaluation fonctionne comme en Lisp. La définition de la fonction "eval" des LFT dans les structures d'interprétation a été donnée dans notre article de 1988b.

contre, si r reste indéfini, l'expression adjour(1,r) restera un prédicat logique (une formule atomique).

Précisons aussi que dans les dérivations grammaticales, toutes les bêta-réductions possibles seront toujours effectuées. Les formes réduites sont en effet toujours plus courtes et de surcroît *les formes irréductibles sont uniques*.

Les représentations du sens des phrases et du texte seront stockées dans une base de connaissances sur laquelle est installé un démonstrateur automatique. Celui-ci, en effectuant des inférences et en répondant aux questions sur le texte, permet d'évaluer empiriquement les hypothèses sémantiques avancées dans l'étude linguistique.

Remarque :

On pourrait se demander pourquoi nous avons choisi de traiter sur des plans différents les informations syntaxiques, les structures de traits et le sens. Il est en effet tentant de ramener les triplets d'analyse à des structures de traits. Par exemple, le triplet associé à "mai 1789"

(DT,(cl:interv),((1789,5,1),(1789,5,31)))

pourrait être représenté par la ST :

(cat:DT,cl:interv,sens:((1789,5,1),(1789,5,31)))

Les adaptations formelles que devraient subir les grammaires généralisées pour pouvoir travailler sur ce genre de ST seraient très faciles à faire.

Cependant sur le plan linguistique, le premier champ des triplets d'analyse regroupe les informations habituellement rattachées aux arbres de dérivation alors que le second rassemble des informations plus ou moins hétéroclites et même souvent ad hoc, car liées aux problèmes de restriction de sélection et d'interprétation sémantique. De plus sur le plan informatique, les parseurs KN (que nous allons voir en 2) traitent de manière totalement différentes ces deux champs.

Enfin, il est légitime de traiter à part le champ sémantique car nous allons y effectuer des opérations beaucoup plus élaborées que la seule unification. En effet, la règle V1∅ vt N2

*U0:clv *U1:clv,

$*S_0 <<= \lambda$ (a x) $*S_2$ (λ (y) $*S_1$(a, x, y))

dans une grammaire d'unification deviendra :

X0:cat=V1, X1:cat=vt, X2:cat=N2,

X0:clv=X1:clv

X0:sens=λ (a x) appl(X2:sens,(λ (y) appl(X1:sens,(a, x, y)))).

La dernière unification effectuera seulement une *substitution* de X2:sens et de X1:sens dans la formule. Pour effectuer toutes les opérations de bêta-réduction et d'évaluation, il faudrait rajouter un module de traitement de l'informations sémantiques à la sortie de la grammaire d'unification. Mais il est clair que ce procédé conduirait à des formules si longues qu'elles seraient complètement impossibles à manipuler "à la main". C'est pourquoi nous préférons effectuer dès que possible toutes les opérations de simplification sémantique afin de faciliter le travail linguistique de mise au point des grammaires (avant toute implantation).

1.3. La grammaire généralisée

Nous sommes maintenant en mesure de rassembler les différentes notions introduites précédemment dans un modèle formel unique.

Une grammaire généralisée

G = (GA , UNIF , LFT , DICO , p)

est formée

- d'une **grammaire algébrique** GA = (CAT , LEX , P_0) :

  avec CAT l'ensemble des catégories syntaxiques (simples) telles que N2, N1, n, V2, V1, vt.. LEX l'ensemble des catégories lexicales (simples) telles que n, vt...Pour la grammaire GA, LEX constitue l'ensemble des symboles terminaux, alors que pour la grammaire G se

ne sont que des préterminaux. L'ensemble des règles de réécriture P_0 est constitué par la relation

P_0 1 (CAT \ LEX) ^ CAT* (par ex. : V2∅ V1 N2)

- d'une **structure d'unification** UNIF = (STRU , L_c,) :
  formée d'un ensemble de structures de traits STRU et d'un langage de contraintes L_c et de l'opérateur de transmission de traits.
- d'un **langage fonctionnel typé** LFT construit à partir de types élémentaires choisis en fonction du domaine sémantique couvert par G. Le LFT est formé de termes TER et de définitions DEF.
- d'un **dictionnaire** DICO :

A chaque acception d'une entrée lexicale est associé un triplet formé d'une information sur la catégorie lexicale, d'une structure de traits et d'une représentation du sens.

Par exemple :

```
devoir    (vaux ,
           (clv:modal,tc:infini) ,
           (lambda (q) (lambda (a x) DEVOIR(a,cp(q,*génera, x)))))
```

(avec *génera une fonction génératrice de constantes)

Soit Σ l'ensemble des entrées lexicales (ou lexèmes). DICO définit une relation non fonctionnelle entre une entrée et un triplet. Nous appellerons ces triplets "(syntaxe, traits, sens)" des **triplets d'analyse** ou des **analyses** :

DICO 1 Σ ^ TRIPL

avec TRIPL = CAT ^ STRU ^ LFT

- enfin un ensemble **de règles** p assure la cohésion de ces différentes structures.

Une règle est un quadruplet R = (r , γ , ω , λ) formé :

1. d'une règle de réécriture r de la grammaire algébrique GA :
   r P_0 notée d'une manière générale par X_0 ∅ X_1 X_2 ....X_n

2. d'une contrainte γ du langage des contraintes L_c sur la structure de traits STRU. γ comporte des variables réservées $*U_i$ pour i = 0...n, associées à chaque catégorie X_i.
3. d'un ensemble de transmission de traits ω TRANS composé d'expressions du type $*U_0$:Chem δ avec "Chem" un chemin dans la structure de traits de la racine X_0 et δ un terme dont l'évaluation retourne une valeur constante ou une structure de traits. Le terme ω permet de calculer la structure de traits de X_0 en fonction des structures de traits de X_1, X_2,...X_n toujours par l'intermédiaire des variables $*U_1$,...$*U_n$.
4. d'un opérateur λ dans le langage fonctionnel typé LFT permettant de calculer la représentation du sens de X_0 en fonction des représentations du sens de X_1...X_n (par l'intermédiaire des variables réservées $*S_1$ ...$*S_n$).

La cohérence de ces quatre types d'information est assurée en imposant une correspondance entre les catégories syntaxiques CAT, les schémas de traits SCHEM et les types sémantiques TYP.

Pour la *structure d'unification*, nous définirons une application

h_u : CAT∅SCHEM

permettant d'associer à la règle $X_0 \varnothing X_1\ X_2$ ....X_n des contraintes sur les structures de traits de chaque X_i. Tout terme $*U_i$:tr_1:...tr_n devra comporter un chemin appartenant au schéma $h_u(X_i)$:

tr_1:..tr_n $h_u(X_i)$

(nous rappelons que les schémas de traits sont les "ossatures" de ST, cf § 1.1).

Exemple : avec V2 ∅ vt N2 on pourra avoir des contraintes sur les chemins associés à vt faisant intervenir les traits de classe verbale clv ou d'accord acc

$*U_1$:clv , $*U_1$:acc si h_u(vt) = (clv, tc, acc, acc:nb, acc:gr)

alors que les chemins associés au N2 ne pourront pas concerner clv mais seulement l'accord

*U_2:acc:nb si h_u(N2) = (acc, acc:nb, acc:gr).

En ce qui concerne l*a partie sémantique* de la règle, les conditions à imposer sont encore plus strictes (cf Montague in Thomason 1974). Pour les catégories majeures (N,N1,N2,V,V1,V2), il doit y avoir une correspondance h_s entre les catégories et les types

$$h_s : CAT \varnothing TYP$$

(étendable en un homomorphisme d'algèbres). Cette fonction permet d'associer à la règle $X_0 \varnothing X_1\ X_2\ ....X_n$
l'opération λ du LFT appliquant des termes de types $h_s(X_1)$,...$h_s(X_n)$ dans des termes de type $h_s(X_0)$

$$\lambda : TER_{hs(X1)} 6 ...6 TER_{hs(Xn)} \varnothing TER_{hs(X0)}$$

i.e. λ est une fonction de $*S_1,...*S_n$ avec $*S_i$ $VAR_{hs(Xi)}$

Pour les autres catégories, il est préférable de prendre une correspondance plus faible :

$$h_s : CAT\ 6\ STRU \varnothing\ TYP$$

Par exemple, les compléments de temps (hier, à huit heures, depuis deux ans etc...) bien que pourvus d'interprétations totalement différentes seront regroupés sous la même catégorie. Il est préférable dans ce cas de leur attribuer des types différents et de contrôler le bon typage des règles d'interprétation par des tests sur les structures de traits.

Ces deux fonctions h_u, h_s sont liées **au principe de compositionalité**. Ce sont elles qui assurent une transmission "en douceur" des structures de traits et des représentations de sens dans l'arbre de dérivation.

Dérivations dans G

Nous allons par analogie avec les dérivations dans la grammaire algébrique GA construire des dérivations dans la grammaire généralisée G.

De même que les règles de P_0 (de type $X_0 \varnothing X_1 X_2 ....X_n$) servent à définir une relation $_{GA}$ dans l'ensemble des formes de L_{GA} , de même allons-nous établir à partir des règles de p (formées de quadruplet (r, γ, ω ,λ)) une relation $_G$ sur l'ensemble des séquences de triplets d'analyse TRIPL*.

Soient les n+1 triplets $(X_i, \theta_i, \sigma_i)$ TRIPL , i = 0...n.

On définit la relation $_G$ sur la séquence de triplets de la façon suivante :

α $(X_1, \theta_1, \sigma_1)$ ...$(X_n, \theta_n, \sigma_n)$ β $_G$ α $(X_0, \theta_0, \sigma_0)$ β

ssi il existe une règle (R) de p de la forme suivante

$(X_0 \varnothing X_1 X_2 ....X_n, \gamma, \omega, \lambda)$ telle que :

- la valeur de la contrainte $\gamma[*U_i \blacklozenge \theta_i]$ pour i = 1...n, soit non nulle (c'est-à-dire que la contrainte est satisfaite lorsqu'on instancie γ par les valeurs θ_i)
- la structure de traits θ_0 soit construite à partir de θ_1,...θ_n suivant les instructions contenues dans ω. Pour ω_i = $*U_0:tr_1:...tr_k$ exp on obtient

  $\theta_0:tr_1:...tr_k = exp[*U_i \blacklozenge \theta_i]_{\ i = 1...n}$
- la représentation du sens σ_0 soit calculée en effectuant tout d'abord les substitutions

  $\lambda[*S_i \blacklozenge \sigma_i]_{\ i=1...n}$ (et éventuellement les substitutions $[*U_i \blacklozenge \theta_i]$)

  puis en évaluant les expressions marquées et enfin en effectuant toutes les bêta-réductions possibles.

  Nous noterons * la fermeture réflexive et transitive de .

Nous avons donné une présentation ascendante de la dérivation, mais il suffirait de prendre la relation inverse $^{-1}$ pour retrouver une présentation descendante plus traditionnelle.

Pour analyser une phrase $\phi = a_1...a_n \quad \Sigma^*$, on commence donc par construire des séquences de triplets en appliquant le dictionnaire sur chaque lexème a_i. Soit (c_i , u_i , s_i) un triplet associé à l'entrée a_i. A ϕ correspond en général, plusieurs séquences de triplets, soit

$(c_1 , u_1 , s_1)...(c_n , u_n , s_n)$ l'une d'entre-elles.

Si il existe une dérivation

$(c_1 , u_1 , s_1)...(c_n , u_n , s_n) \quad * \quad (PH, u, s)$

conduisant au triplet de catégorie phrase PH, alors $\phi = a_1...a_n$ est **une phrase du langage L_G** et s est la représentation de son sens.

Lorsque les règles de p ne comportent pas de contrainte (i.e. la contrainte γ = vrai) et qu'un bon typage supprime tout effet filtrant de la sémantique, le langage engendré L_G est identique au langage algébrique L_{GA} de la grammaire algébrique associée GA. Par contre, ***la présence de contraintes augmente la capacité générative de G.*** Nous donnons en annexe une grammaire généralisée engendrant le langage dépendant du contexte $\{a^n b^n c^n : n>0\}$. Suivant Shieber 1986, la possibilité de filtrage par unification permet d'engendrer n'importe quel langage récursivement énumérable.

2. Parseurs et générateurs

Maintenant que nous disposons d'un modèle linguistique précis, nous allons voir comment il est possible de faire le parsage des langages qu'il engendre. Pour pouvoir travailler sur des séquences de triplets, il va nous falloir procéder à quelques aménagements des parseurs de langages algébriques. Nous ne présenterons ici que deux exemples que nous espérons suffisamment typiques pour illustrer la méthode.

Rappelons-nous qu'il existe deux grands types de générateurs :
- les générateurs travaillant globalement sur l'ensemble des règles, sur le mode des compilateurs. Ils utilisent des algorithmes assez complexes pour construire des tables de parsage. Les parseurs ascendants sont dits LR(1), les descendants LL(1). Nous avons implanté une adaptation de chacun d'eux, mais nous ne présenterons pas les LL(1) qui sont moins rapides que les "grammaires à clauses définies" (nous avons faits des tests), et qui sont plus difficiles à mettre en oeuvre.
- les générateurs travaillant règle par règle et qui s'apparentent aux interpréteurs. Ces programmes de génération en général extrêmement simples, se limitent à une traduction immédiate des règles grammaticales dans la "syntaxe du parseur". Nous présenterons les parseurs descendants du type "grammaire à clauses définies" GCD de Pereira et Warren 1980 - mais il est aussi possible d'adapter les parseurs ascendants de Matsumoto et al. 1984,1987 ou les parseurs par diagrammes (les 'chart parsers').

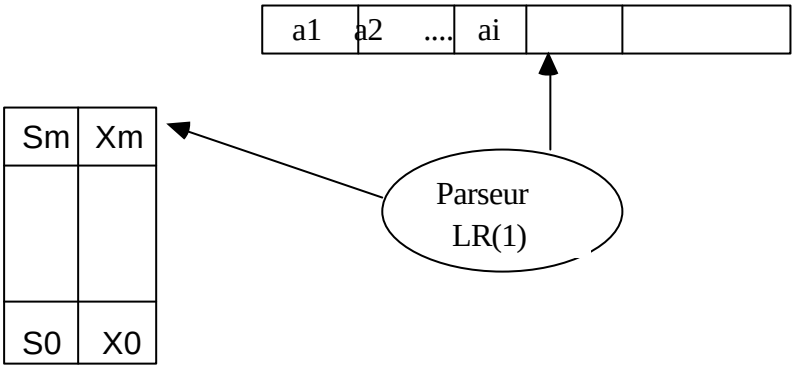
2.1. Un parseur ascendant et son générateur

La technique de parsage LR des langages algébriques est remarquablement efficace. Elle réalise en quelque sorte la méthode la plus générale d'analyse par déplacement-réduction, sans recours au backtracking.

Mais pour être applicable aux langues naturelles, elle doit être étendue pour permettre :

- d'analyser les langages ambigus[10] (et donc non déterministes)
- de mener en même temps une analyse syntaxique et sémantique.

Nous supposerons connu la technique **des parseurs LR** (Aho, Sethi, Ullman 1986). L'analyseur LR traditionnel travaille sur une séquence d'entrée constituée par une suite de lexèmes et une pile double d'états et de symboles.



pour une grammaire algébrique (CAT , Σ , P_0)

$a_1...a_n$ Σ^*

$X_0...X_m$ $(\Sigma \text{ ' } CAT)^*$ $s_0...s_m$ ETAT

Pour pouvoir analyser les langages engendrés par les grammaires généralisées, nous allons faire subir quelques modifications aux parseurs LR(k).

Pour une grammaire généralisée G = (GA , UNIF , LFT , DICO , p), nous définissons un **parseur $KN(1)_G$** tel que

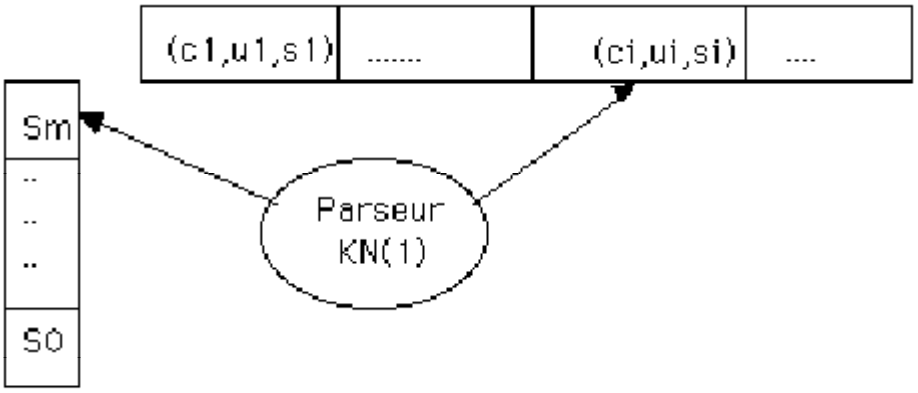
[10]La paternité de l'emploi des parseurs LR pour l'analyse des langages ambigus est généralement attribuée à Tomita 1982, 1987. Mais nous n'avons pris connaissance de ces travaux qu'après avoir écrit notre programme. Notre source d'inspiration a plutôt été Deredec [Plante 1988] qui nous a incité à travailler sur des séquences de n-uplets complexes.

- le parseur $KN(1)_G$ possède **une table de parsage** identique à la table de parsage LR(1) construite à partir de la grammaire algébrique GA = (CAT, LEX, P_0)
- le parseur $KN(1)_G$ travaillera sur cette table et sur une **séquence d'entrées** *constituée de triplets d'analyse*

$(c_1 , u_1 , s_1)...(c_n , u_n , s_n)$ TRIPL*

construite à partir de la séquence de lexèmes $a_1...a_n$ Σ^* et du DICO.
- la **pile** ne comportera que des états (les symboles associés à chaque état demeurent dans la séquence d'entrée).
- les **déplacements** se font en avançant un pointeur sur la séquence d'entrée.
- les **réductions** se font en dépilant et en *modifiant* la séquence d'entrée. Pour réduire la règle

$(X_0 \varnothing X_1 X_2 ....X_n, \gamma, \omega, \lambda)$ p le parseur KN vérifie d'abord si la contrainte γ est vérifiée. Si elle l'est, il dépile n états de la pile et construit un nouveau triplet $(X_0 , \theta_0 , \sigma_0)$ à partir des n derniers triplets pointés (θ_0 est calculé à partir de ω et σ_0 à partir de λ, comme nous l'avons vu dans les dérivations généralisées au paragraphe précédent). Ces n derniers triplets sont finalement remplacés dans la séquence par le triplet $(X_0 , \theta_0 , \sigma_0)$.
- lors de **conflits** réduction/réduction ou déplacement/réduction toutes les actions possibles sont effectuées en pseudo-parallélisme



Nous allons voir sur un exemple très simple comment fonctionne ce parseur. Pour être aussi bref que possible, nous prendrons une grammaire formée de deux règles, hors du domaine linguistique.

Les expressions telles que 100/4/2 (cent divisé par quatre divisé par deux) sont ambiguës. En effet :
(100 / 4) / 2 = 25 / 2 = 12,5
(100 / (4 / 2)) = 100 / 2 = 50
Prenons pour grammaire de ce langage :
$G_1 = (GA_1 , STRU_1 , LFT_1 , DICO_1 , p)$

1. E ∅ E op E
   NON(*U_2:opr==divi) OU (*U_3:div==+)
   *U_0 *U_1
   *S_0 <<= *S_2 (*S_1 , *S_3)
2. E ∅ nb
   vrai
   *U_0 *U_1
   *S_0 <<= *S_1

$DICO_1$ est constitué des entrées suivantes :
cent (nb, (div:+), 100)
quatre (nb, (div:+), 4)
zéro (nb, (div:-), 0)
/ (op, (opr:divi), /)

+ (op, (opr:plus), +)

La structure de traits a pour but d'exclure les expressions comportant 'zéro' pour diviseur. La contrainte de la première règle impose que "si l'opérateur est une division (avec la trait (opr:divi)) alors le diviseur doit être différent de zéro (ie. porte le trait (div:+))".

Pour construire le parseur de G_1, on commence par construire la table de parsage de la grammaire algébrique GA_1 (avec §§ pour symbole de fin de séquence).

tab	nb	op	E	§§
S0	S1		S2	
S1		red2		red2
S2		S3		succès
S3	S1		S4	
S4		S3,red1		red1

Le parseur de G_1 devant travailler sur une séquence de triplets, on appliquera $DICO_1$ à la séquence terminale puis on lui adjoindra un marqueur de fin de séquence §§. Pour

cent / quatre / deux

on obtient

(nb, (div:+), 100) (op, (opr:divi), /) (nb, (div:+), 4) (op, (opr:divi), /) (nb, (div:+), 2) (§§)

Une *configuration* est constituée par la pile des états (avec le haut à droite), un pointeur sur la séquence et la séquence d'analyse.

S0 0 (nb, (div:+), 100) (op, (opr:divi), /) (nb, (div:+), 4) (op, (opr:divi), /) (nb, (div:+), 2) (§§)

Le pointeur 0 indique le premier triplet (nb, (div:+), 100). La table donne tab(S0,nb)=S1 soit un déplacement :

S0 S1 1 ..... idem .....

Suite à ce déplacement, l'action tab(S1 , op) = red2 est la réduction de la règle 2 (E ∅ nb, vrai , $*U_0$ $*U_1$, $*S_0$ <<= $*S_1$) . Elle s'effectue

en modifiant la catégorie de la première analyse, en diminuant le pointeur de 1, et en dépilant de 1 :

S0 0 (E, (div:+), 100)(op, (opr:divi), /) (nb, (div:+), 4) (op, (opr:divi), /) (nb, (div:+), 2) (§§)

suivi de trois déplacements :

S0 S2 S3 S1 3 ..... idem .....

et d'une nouvelle réduction de (R2) :

S0 S2 S3 2 (E, (div:+), 100)(op, (opr:divi), /) (E, (div:+), 4)
(op, (opr:divi), /).....
S0 S2 S3 S4 3 ..... idem .....

A ce stade le tableau indique deux actions : un déplacement vers S3 et une réduction (R1). Les deux actions sont effectuées en pseudo-parallélisme.

- La *réduction* teste successivement si le triplet de l'opérateur ne comporte pas le trait (opr : divi) et si le triplet de "quatre" comporte le trait (div:+). Puisque cette dernière condition est satisfaite, le test réussit. Il autorise donc la transmission de la structure de traits et la construction du sens par la division du sens de (E,(div:+),100) par le sens de (E,(div:+),4), soit 100/4 = 25.

S0 0 (E, (div:+), 25) (op, (opr:divi), /) (nb, (div:+), 2) (§§)

La poursuite de cette analyse conduit à la configuration :

S0 S2 1 (E, (div:+), 12,5) (§§)

soit un succès (d'après la table).

- le *déplacement* donne

S0 S2 S3 S4 S3 4 (E, (div:+), 100)(op, (opr:divi), /)
(E, (div:+), 4)(op, (opr:divi), /)....

la poursuite de cette analyse donne :

S0 S2 1 (E, (div:+), 50) (§§)

soit un succès (toujours d'après la table).

Nous donnons en annexe une session de travail effectuée sur cette grammaire (avec un générateur et un interpréteur écrit en Le-LISP).

Si on désire **construire les arbres de dérivation**, il suffit de prendre pour premier élément de l'analyse (c_i , u_i , s_i) non plus une catégorie mais l'arbre de dérivation de GA_1. Par exemple, dans la dérivation précédente, avant la réduction de (R1), on serait dans la configuration suivante :

S0 S2 S3 S4 3 ((E(nb(cent)), (div:+), 100) (op(/), (opr:div), /) (E(nb(quatre)), (div:+), 4) ..

Le dictionnaire dans ce cas, comportera des entrées avec des arbres comme premier élément :

cent (nb(cent), (div:+), 100)

La réduction de (R1) ensuite greffera les arbres des trois derniers triplets sous la catégorie E :

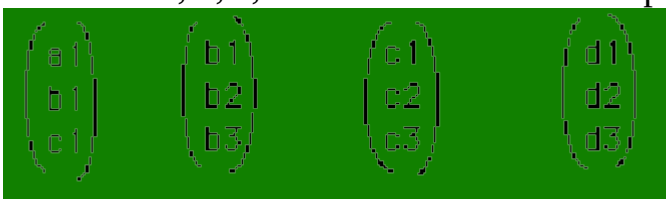
S0 0 ((E(E(nb(cent)),op(/),E(nb(quatre))), (div:+), 25) .....

Nous avons ainsi un parseur extrêmement *souple*. Si nous cherchons avant tout à faire de la compréhension automatique, nous lancerons le parseur sans lui faire construire d'arbre, ce qui lui permet d'aller plus vite. Si par contre, nous nous intéressons aussi à l'analyse syntaxique, il suffira de lancer le parseur en lui demandant de construire les arbres (et éventuellement de donner les règles sans opérateur sémantique).

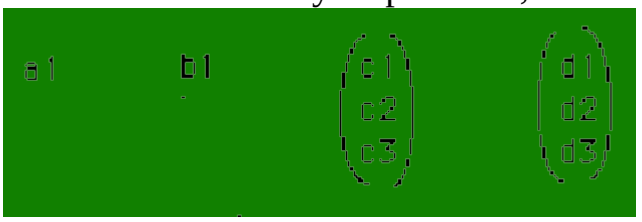
Sur l'analyse de l'expression ambiguë précédente, nous pouvons observer que le parseur lorsqu'il rencontre la possibilité d'une deuxième analyse, tient compte de tout le travail effectué antérieurement. Chaque analyse n'est pas produite indépendamment à partir du début. Les réductions des deux premiers termes "nb" (par E∅nb) ne sont faites qu'une seule fois. Cependant la réduction du troisième terme "nb" (par E∅nb) est faite deux fois (une fois dans chaque analyse). Nous ne réalisons donc que partiellement les objectifs de Tomita [1987] dans son "parseur efficace des grammaires algébriques". En travaillant sur des triplets, il est bien sûr facile d'adapter sa technique à nos besoins. Mais l'algorithme de Tomita suppose que les actions soient effectuées en se déplaçant dans un graphe orienté sans circuit, une structure beaucoup plus lourde à manier que les arbres que nous utilisons. Pour des exemples simples, il est probable que le temps perdu à essayer de faire des "convergences" n'est pas compensé par le temps gagné (lorsque ces convergences réussissent) à éviter de refaire les mêmes opérations. De plus en linguistique, il est monnaie courante que des arbres de dérivation identiques soient associés à des représentations du sens différentes. Et

alors, très souvent là où le parseur syntaxique de Tomita pourrait entreprendre des analyses en commun, nous nous trouverions avec des triplets dont la partie sémantique différente obligerait à poursuivre des analyses indépendantes.

Remarquons pour finir, que la possibilité de traiter de la polysémie[11] introduite dans la définition du dictionnaire, n'est pas lourdement pénalisée par un accroissement du temps de calcul. Si par exemple, les quatre entrées a, b, c, d ont chacune trois acceptions différentes :



au lieu de lancer les 81 analyses possibles, on commence par analyser



et comme le parseur ne regarde qu'un élément en avant, il est possible qu'il élimine la séquence a_1 b_1 et par là les 9 analyses associées[12].

[11]Plus précisément, la possibilité d'associer à un seul lexème plusieurs triplets de catégorie lexicale et de sens différents, ou de même catégorie mais de sens distincts.

[12]cf. Tomita où cette technique est clairement expliquée ; nous avions personnellement adopté une procédure semblable il y a plusieurs années, avec des ATN.

Le générateur de parseurs

La conception du générateur de parseurs découle directement de toutes les considérations précédentes. C'est une adaptation très simple des générateurs de parseurs LR(1) (cf. Aho, Sethi, Ullman 1986; Tremblay, Sorenson 1985) aux grammaires généralisées.

Nous avons vu comment associer une grammaire algébrique GA à la définition d'une grammaire généralisée G. Il suffit donc de reprendre l'algorithme d'un générateur de parseurs LR(1) et de le faire travailler sur GA. Mais nous lui faisons construire une table de parsage mieux étoffée : lorsque le générateur décèle une *réduction*, il note dans la table non pas la règle de GA mais la règle de G (avec les informations sur la structure de traits et la sémantique). Il convient bien sûr de construire le parseur dans une syntaxe qui soit acceptée par l'interpréteur. Nous donnons en annexe quelques exemples simples de parseurs construits par notre générateur (sans prétention linguistique, ils sont donnés uniquement à titre d'exercices formels).

Nous remarquons pour finir qu'à toute grammaire généralisée G correspond[13] *un parseur KN(1) reconnaissant le langage engendré par G.*

2.2. Un parseur descendant et son générateur

Il est si facile de profiter de l'interpréteur de PROLOG pour construire des interpréteur de parseurs que Pereira et Warren ont qualifié le leur du nom de grammaire (à clauses définies GCD). Nous allons faire subir quelques petites adaptations à ce parseur pour pouvoir effectuer des dérivations dans les grammaires généralisées présentées en §1.

Les parseurs descendants du type GCD ont le grave défaut de perdre énormément de temps dans des *backtrackings* intempestifs et *d'interdire d'utiliser les règles récursives à gauche* (comme N1 ∅ N1 P2) sous

[13]La démonstration assez technique ne peut être donnée ici.

peine de tomber dans des boucles infinies. Malgré ces faiblesses, nous mentionnons cependant ce genre d'analyse pour bien mettre en évidence la possibilité de construire automatiquement des parseurs qui ne dépendent que des propriétés universelles du modèle linguistique et non pas des propriétés particulières d'une langue donnée.

Le générateur de parseur descendant que nous avons construit (en Mac PROLOG) accepte les règles de la grammaire généralisée, dans le même format que le générateur de parseur ascendant. Mais au lieu de travailler sur l'ensemble des règles, il effectue ses traductions règle par règle. Par exemple la règle (R1) donnée en §1 :

(R1) 1. V_1 ∅ vt N_2
2. *U_1: clasv = actif
3. *U_0 *U_1
4. *S_0<<= λ (a x) *S_2 (λ (y) *S_1(a, x, y))

sera traduite dans la clause PROLOG :

```
'V1'(L1,L3,'V1'(A1,A2),U0,S0) :-
    lexi(vt,L1,L2,A1,U1,S1),
    'N2'(L2,L3,A2,U2,S2),
    test(U1:clasv '==u' actif),
    U0<=U1,
    S0<<=lambda([A,X],appl(S2,[lambda([Y],appl(S1,[A,X,Y]))])).
```

Les variables L_i reçoivent les séquences de triplets d'analyse, obtenues comme nous l'avons déjà vu en appliquant DICO sur la phrase d'entrée. Par exemple, pour analyser "Marie voit Paul", le syntagme "voit Paul" donnera la séquence d'analyse

```
[[vt,vt(voir),[acc:[nb:sg,per:3],tc:pres],lambda([A,X,Y],voir(A,X,Y))],
 [npr,npr(Paul),[acc:[nb:sg,per:3]],paul]]
```

Les A_i sont les arbres construits, les U_i les structures de traits et les S_i les représentations du sens (l'opérateur : est noté & dans le listing 'grammaire 5' de l'annexe).

.Le prédicat "lexi" vérifie que la catégorie lexicale vt est la catégorie du premier triplet d'analyse :

```
lexi(Cat,[[Cat,A,U,S]|R],L,A,U,S).
```

et retourne l'arbre A, la structure U et le sens S.

."test" est un prédicat PROLOG dont la définition a été esquissée en § 1.1.

. est un opérateur PROLOG, réalisant l'unification des ST.

.<<= est aussi un opérateur PROLOG effectuant les opérations du langage fonctionnel LFT .

Pour l'application par exemple, on a la substitution des valeurs LV dans les variables LV de Tm :

```
R<<=appl(lambda(LV,Tm),LV) :- R<<=Tm.
```

alors que pour les termes composés recevant une définition dans le LFT, on a :

```
Sortie<<=Ter :-
        Ter=..[Fct|Larg],
        lbd_fct_aux(Fct),       /*teste si Fct a reçu une définition dans le LFT */
        évalsém(Larg,Lval)      /* évalue la liste des arguments */
        append(Lval,[Sortie],Arg),
        K=..[Fct|Arg],      /* Fct  fonction à n arguments A1,..  An,
                                    est en PROLOG
        K.                          un prédicat à n+1 arguments
                                    Fct(A1,..An,Sortie) */
```

Ce genre d'interpréteur de parseur bien que relativement lent, a cependant l'avantage de permettre des mises au point progressives.

Conclusion :

Le modèle linguiste que nous proposons est le fruit de plusieurs années d'études en prise directe avec la réalité linguistique. Il découle

plus particulièrement de nos efforts pour concevoir une sémantique de la localisation et du temps.

Les générateurs et interpréteurs de parseurs ont par contre, été conçus récemment et en moins de temps (mais à des époques différentes, ce qui explique certaines distorsions dans les notations des listings de l'annexe). Ils nous sont très utiles pour mettre aux point de grosses grammaires dans lesquelles il peut s'avérer difficile de juger des effets parfois très indirects et lointains, provoqués par de petites retouches locales. Et bien évidemment ils permettent un gain de temps fantastique quand il s'agit "d'implanter" sur machine une étude linguistique. Ils devraient en outre, être d'utilisation aisée pour les linguistes puisque leur mise en oeuvre ne requiert aucune connaissance sur les techniques de parsage. Ils ont enfin un caractère universel dans la mesure où ils utilisent les propriétés générales d'un modèle linguistique bien défini.

Francis RENAUD
EHESS-CRLAO

ANNEXE

Les programmes du générateur de parseurs KN(1) ont été écrits en Le-Lisp sur Macintosh. Les parties GRAMMAIRE, DICO et FONCTIONS sont les listings des fichiers d'entrée du générateur. Les programmes ont été écrits à une époque où nous préfixions les opérateurs (comme en Lisp) de la grammaire généralisée. Les sections PARSEURS sont les sorties du générateur qui deviennent les fichiers d'entrée de l'interpréteur. Enfin, les ANALYSES correspondent à des sessions de travail sur l'interpréteur. Les grammaires 1 à 4 illustrent l'emploi du générateur de parseurs KN.

Les programmes du générateur de parseurs descendants ont été écrits en Mac Prolog sur Macintosh. La grammaire 5 est une illustration très simple de ce genre de générateur.

Références bibliographiques

AHO A.V., R. SETHI, J.D. ULLMAN (1986), *Compilers. Principles, Techniques, and Tools*, Addison-Wesley, 796 p.

AÏT-KACI H. (1986), "An Algebraic Semantics Approach to the Effective Resolution of Type Equations", *Theoretical Computer Science* 45, pp. 293-351.

AÏT-KACI H., P. LINCOLN (1988), "A Natural language for Natural Language", MCC Technical Report Number ACA-ST-074-88.

AUTEBERT J-M. (1987), *Langages algèbriques*, Masson, 278p.

ALLEN J. (1987), *Natural Language Understanding*, The Benjamin/Cummings Pub. Co., 574 p.

BOLC L. (ed)(1987), *Natural Language Parsing Systems*, Springer-Verlag, 367 p.

BRESNAN J. (ed.) (1982), *The Mental Representation of Grammatical Relations*, The MIT Press, 874 p.

DAWAR A. & K. VIJAY-SHANKER (1990), "An Interpretation of Negation in Feature Structure Descriptions", *Computational Linguistics*, Vol. 16, n° 1, pp. 11-21

GAZDAR, KLEIN, PULLUM, SAG (1985), *Generalized Phrase Structure Grammar*, Basil Blackwell, 274 p.

GAZDAR G., C. MELLISH (1989), *Natural Language Processing in PROLOG*, Addison-Wesley, 504 p.

GAZDAR, PULLUM, CARPENTER, KLEIN, HUKARI, LEVINE (1988), "Category Structures", *Computational Linguistics* , Vol. 14, n° 1, pp. 1-19. Traduction dans Miller & Torris 1990, chap 6.

HARRISON M.A. (1978), *Introduction to Formal Language Theory*, Addison-Wesley, 594 p.

JOHNSON M. (1988), *Attribute-Value Logic and the Theory of Grammar*, CSLI Lectures Notes 16, Stanford Univ.

KAPLAN R.M. & BRESNAN J. (1982), "Lexical-Functional Grammar: a formal system for grammatical representation", in J. Bresnan (ed.) *The Mental Representation of Grammatical Relations*, pp. 173-381, MIT Press.

KASPER R.T. &W. ROUNDS (1986), "A Logical Semantics for Feature Structures", *Proceedings of the 24th Annual Meeting of the ACL*, Columbia Univ., pp. 257-265 Traduction dans Miller & Torris 1990, chap 7.

KAY M.(1979), "Functional grammar", In *Proceedings of the Fifth Annual Meeting of Berkeley Linguistics Society.*

KAY M. (1985), "Parsing in Functional Unification Grammar" in D.R. DOWTY, KARTTUNEN, A. ZWICKY (eds), *Natural Language Parsing*, pp. 251-278, Cambridge Un. Press.

MATSUMOTO Y., M. KIYONO, H. TANAKA (1985), "Facilities of the BUP parsing system", in V. DAHL, P. SAINT-DIZIER (eds), *Natural language understanding and logic programming*, pp. 97-106, Elsevier, North-Holland.

MATSUMOTO Y., R. SUGIMURA (1987), "A Parsing System Based on Logic Programming", in *IJCAI 87*, Vol. 2, pp. 671-674, Morgan Kaufmann.

MILLER Ph. & TORRIS Th. (eds.) (1990), *Formalismes syntaxiques pour le traitement automatique du langage naturel*, Hermès, 359p.

MOSHIER M.D. & ROUNDS W.C. (1987), "A Logic for Partially Specified Data Structures", *ACM Symposium on the Principles of Programming Languages*, pp. 156-167.

OEHRLE R.T., E. BACH, D. WHEELER (eds) (1988), *Categorial Grammars and Natural Language Structures*, Reidel, 524 p.

PAULSON L.C. (1987), *Logic and computation. Interactive proof with Cambridge LCF*, Cambridge Univ. Press, 302 p.

PEREIRA F. C. N., D.H.D. WARREN (1980), "Definite Clause Grammars For Language Analysis - A survey of the formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence*, 13, pp. 231-278.

PLANTE P. (1988), "Déredec : atelier de programmation pour l'analyse et la modélisation de systèmes symboliques". UQAM, Université du Québec, Montréal.

RENAUD F. (1988a), *Pour une sémantique opératoire*, Editions Langages Croisés - CRLAO (épuisé).

RENAUD F. (1988b), "Présentation d'un modèle linguistique basé sur les langages fonctionnels typés", *Intellectica*, 6, 1988/2.

SHIEBER S. (1986), *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Notes 4, Chicago U. Press.

SMOLKA G. (1988), "A Feature Logic with Subsorts", LILOG report 33, IWBS, IBM Deutschland (à paraître dans *Journal of Automated Reasoning"*).

SMOLKA G. (1989),"Feature Constraint Logics for Unification Grammars", (à paraître dans *Journal of Logic Programming*)

TOMITA M.(1982), *Efficient Parsing for Natural Language*, Kluwer Academic, 201p.

TOMITA M. (1987), "An Efficient Augmented-context-free Parsing Algorithm", *Computational Linguistics*, Vol. 13, n° 1-2.

TREMBLAY J-P., P.G. SORENSON (1985), *The theory and practice of Compiler Writing*, McGraw-Hill, 796 p.

THOMASON R.H.(ed.)(1974), *Formal Philosophy, select papers of Richard Montague*, Yale Univ., 369p.

WIKSTRÖM A.(1987), *Functional Programming Using Standard ML*, Prentice Hall, 446 p.