# Chapitre 9

# La réflexivité pour une évolution des représentations dans le traitement automatique du langage naturel

## 9.1. Réflexivité, une introduction

L'énigme MU est présentée dans (Hofstadter 1986) pour illustrer les limites des systèmes formels. Au départ une question : "Pouvez-vous produire MU ?". Pour cela on dispose des éléments suivants :

L'axiome MI, les symboles M, U, I et les règles :

- Règle 1 : si vous possédez une chaîne se terminant par I, vous pouvez lui ajouter un U à la fin.
- Règle 2 : supposons que vous ayez Mx. Vous pouvez alors ajouter Mxx à votre collection.
- Règle 3 : si la chaîne des symboles III apparaît dans votre collection, vous pouvez former une nouvelle chaîne en remplaçant III par U.
- Règle 4 : si le groupe de symboles UU apparaît dans l'une des sous-chaînes, vous vous pouvez le supprimer.

La démarche raisonnable pour commencer à produire MU (en s'aMUsant) est de travailler à l'intérieur du système ainsi défini. C'est à dire à produire des dérivations successives à partir de MU :

règle 2 règle 2 règle 3
MI ----> MII ----> MIII ----> MIIIU ....

Jusqu'où peut-on aller dans cette dérivation sans perdre patience ? Il est préférable en fait d'examiner ce système et son fonctionnement à un niveau extérieur au processus de dérivation, pour évaluer pourquoi il est si difficile de produire MU. Si on regarde ce système de l'extérieur, on peut remarquer que :

- 1. Les nombres 2 et 3 jouent un rôle important, puisque les séries de trois I et de deux U sont éliminées.





- 2. Le système utilise des tendances opposées : des règles d'allongement (1 et 2), et des règles de raccourcissement (3 et 4).

Pour résoudre l'énigme il faut aller encore plus loin et utiliser un outil extérieur au système : la théorie des nombres. Si on compte le nombre de I des théorèmes du système, on s'aperçoit très vite qu'il n'est jamais nul : on ne peut pas éliminer tous les I. Les règles 1 et 4 n'ont aucun effet sur le nombre de I. Seules les règles 2 et 3 influent sur ce nombre. La règle 3 diminue de 3 le nombre de I. Dans une application de la règle 3, le nombre de I pourrait très bien être un multiple de 3 si le nombre initial de I était lui aussi un multiple de 3 : cette règle ne peut créer un multiple de 3 qu'à condition de commencer avec un multiple de 3. La règle 2 double le nombre de I. De même, elle ne produira pas de multiple de 3 s'il n'y en avait pas un avant.

Pour résumer :

- 1. Le nombre de I initial est 1, et 1 n'est pas un multiple de 3.
- 2. Deux des quatre règles n'ont aucun effet sur le nombre de I.
- 3. Les règles 2 et 3 modifient le nombre de I de telle sorte qu'aucun multiple de 3 ne peut être produit.

La conclusion s'impose : le nombre de I ne peut jamais devenir un multiple de 3 et il est impossible que le nombre de I soit nul. C'est la méta-représentation du système MU dans la théorie des nombres qui a permis de résoudre cette énigme. Il faut se reculer pour observer l'ensemble de la situation. Puis choisir la méta-représentation qui servira de base de travail.

# 9.2. Concepts fondamentaux de la réflexivité : l'apport de Pattie MAES (MAES 1987)

## 9.2.1. (Méta) Système (Réflexif)

**Réflexivité**

*"Il y a réflexivité chaque fois qu'un système s'applique à lui-même."*

On parle de système pour décrire un mécanisme lié à une certaine partie du monde (le domaine). Ce système comprend une représentation des objets liés à ce domaine et les moyens de travailler avec/sur ces objets. Un méta-système est un système dont le domaine d'activité est lui-même un système lié à un domaine. On dira qu'un système est réflexif s'il raisonne et agit sur lui-même. Un système est réflexif si le méta-système associé est ce même système.

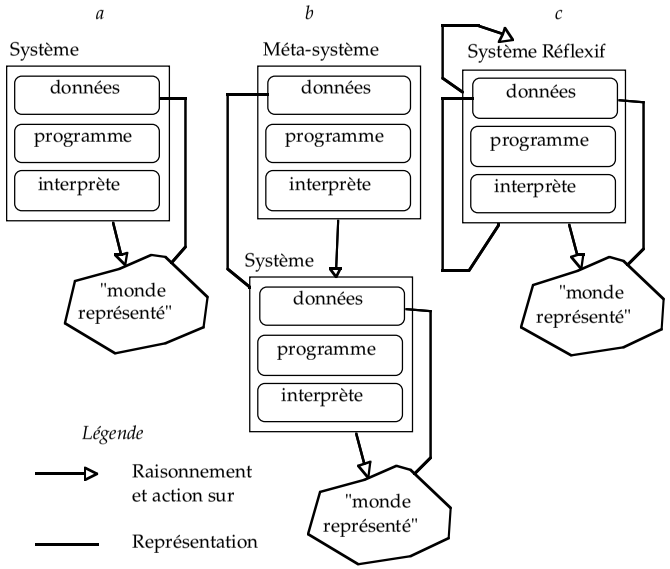



Figure 9.1 : (Méta) Système (Réflexif).

Ainsi on parle de réflexivité quand :

- Une partie d'un système fait référence au système dont elle fait partie, un énoncé déclaratif fait référence à une partie du système dont il fait partie :

  "Cette phrase contiendrait sept mots si elle en avait cinq de moins."

- Le système crée une copie de lui-même : dans les mécanismes cellulaires, les enzymes interviennent dans des créations de copie d'elles-mêmes (cf infra).

- Une partie A d'un système peut modifier une autre partie B de ce système, si A est capable d'apprendre à modifier B en vue d'obtenir de meilleures performances pour B.





- Le système s'observe sans se modifier pour évaluer son propre comportement (expertise de diagnostic, d'édition, d'interprétation...). On trouve très fréquemment dans les gros systèmes informatiques des outils de surveillance de l'activité de l'ensemble du système : CPU disponible, prise en compte des entrées sorties,...(que l'on trouve par exemple sous forme de "Monitoring", ou d'outils chargés de détecter des erreurs spécifiques (problèmes d'écriture sur disque...) et capables d'avertir le responsable (humain) du système pour qu'il intervienne.

La biologie moléculaire est d'ailleurs un très bon exemple de domaine où les phénomènes de différences de niveau de représentation agissent les uns sur les autres. On peut ainsi illustrer ce phénomène par l'exemple de la synthèse des protéines. Le gène provoquant la ou les protéines, fait partie d'un ensemble permettant de réguler la production des protéines en fonction des besoins de la cellule. Cet ensemble s'appelle l'opéron, il est composé d'une séquence d'ADN, appelé gène promoteur, d'une autre, le gène opérateur, et d'une troisième, le ou les gènes de structure. Un gène régulateur est associé à ce mécanisme. L'ADN se trouve dans le noyau des cellules. Il se compose de longues chaînes de molécules appelées nucléotides. Or si quelques protéines sont fabriquées dans le noyau, la plupart d'entre elles sont synthétisées à l'extérieur (dans le cytoplasme) pour y être utilisées. Il existe donc nécessairement des intermédiaires exécutant dans le cytoplasme le programme se trouvant dans le noyau. Une protéine est une suite d'acides aminés (aa). Avec les vingt aa existant, des milliers de protéines différentes peuvent se constituer : elles seront différentes de par leurs séquences d'aa et auront donc des rôles différents dans la cellule. Un programme inscrit sur le gène est très strict : il s'agit de déterminer avec précision l'ordre des aa composant la protéine produite. A chaque groupe de trois nucléotides correspond un aa particulier. En pratique, une copie du message du gène de structure est constituée sous la forme d'une molécule d'ARN-Messager (ARN-M). Cette molécule est capable de traverser la membrane nucléaire. D'autres intermédiaires (ARN de transfert) vont chercher dans le cytoplasme des aa qui s'assembleront selon la suite des nucléotides de l'ARN-M et donc de l'ADN. Ce qui est remarquable dans ce mécanisme, c'est que la cellule peut doser la production des substances dont elle a besoin. Lorsque la cellule n'a pas à produire une certaine protéine, la copie du gène de structure l'ARN-M ne peut pas s'effectuer. Lorsque la synthèse doit reprendre, les produits bloquant l'opéron du gène opérateur sont inactivés et l'ARN-M peut de nouveau produire.

## 9.2.2. Langages de programmation informatique et réflexivité

**Un programme réflexif se définit au travers d'une reproduction intérieure des représentations construites et des modifications qu'il y apporte.**

Plusieurs approches existent pour construire des systèmes réflexifs. Tout d'abord, certains systèmes incluent une part de réflexivité. Depuis peu les Systèmes d'Apprentissage sont implémentés en prenant en compte de manière adéquate les processus réflexifs. Il y a ensuite les systèmes qui font usage de processus réflexifs du fait qu'ils sont implémentés dans des langages de programmation qui fournissent des facilités pour l'utilisation de ces processus. LISP est un exemple d'un tel langage. Certaines fonctions lispiennes permettent un accès sur leur propre fonctionnement. Par exemple considérons les fonctions suivantes :

```
(defun integer-stream () '(0))
(defun integer-stream-element ()
   (let* (( list-of integers (integer-stream))
          (last-generated-integer (car list-of-integers)))
   (eval `(defun integer-stream ()
                  ',(cons (1+ last-generated-integer)
                     list-of-integers)))
   last-generated-integer))
```

Au premier appel de la fonction `integer-stream-element`, celle-ci retourne `0`, et elle redéfinit la fonction `integer-stream` qui devient :

```
(defun integer-stream () '(1 0))
```

et donc, au prochain appel de la fonction `integer-stream-element`, celle-ci retourne `1` ....

Il y a enfin les systèmes implémentés dans des langages dont la structure interne inclut une spécification explicite de la réflexivité (à l'image du langage 3-LISP, développé par B. C. Smith (1982)). Tout système implémenté avec un de ces langages peut raisonner et agir sur lui-même. On dit que de tels systèmes ont une architecture réflexive. Une caractéristique des langages construits avec une architecture réflexive est l'identité de représentation du niveau de l'activité liée à la réflexité et le niveau lié à l'exécution propre au système.

## 9.2.3. Architecture réflexive

On dit qu'un langage dispose d'une architecture pour X si le langage donne de l'importance au concept X et donc s'il fournit des primitives adéquates pour des activités réflexives. Ainsi un langage de programmation fournit une architecture réflexive si la réflexivité est un concept fondamental du système. Ceci implique que le langage produit du code spécifique au processus de réflexivité compréhensible par l'interpréteur de ce langage. Un langage possédant une architecture réflexive est tel que toute activité du système a une représentation d'elle-même au moment de son exécution. Une architecture réflexive permet par exemple d'utiliser des processus de réflexivité à un certain moment d'élaboration d'un programme pour par exemple mettre au point celui-ci. Une analyse automatique développée dans ce cadre pourrait ainsi construire des méta-représentations liées aux différents stades de l'analyse et ainsi permettre d'évaluer par exemple la circulation des traits manipulés par la grammaire. Les systèmes informatiques utilisent certains processus réflexifs pour par exemple : établir des statistiques sur les ressources utilisées ou disponible par ou pour le système, prévenir les utilisateurs si certaines erreurs sont détectées (problème d'écriture sur un disque...), permettre le traçage d'un programme, le "debuggage", etc. Mais à la différence des systèmes à architecture réflexive, les systèmes informatiques classiques n'ont pas été implémentés en prenant en compte la réflexivité comme un concept fondamental. Pour les systèmes à architecture réflexive, les processus de réflexivité sont développés de manière plus modulaire ce qui rend les efforts d'implémentation moins ardus et produit des systèmes plus élégants et plus ouverts. Le langage 3-LISP (Smith 1982), est conçu avec deux types de fonctions. Les fonctions simples construites avec `define-SIMPLE` ou `lambda-SIMPLE` et qui construisent des objets fonctionnels "classiques" pour le calcul informatique. Les fonctions réflexives construites avec `define-REFLECT` sont des fonctions dédiées au processus de réflexivité. Celles-ci ont accès à une représentation globale de l'environnement du système en activité. Ces fonctions travaillent sur des variables qui représentent l'environnement et peuvent les inspecter et les modifier.

## 9.2.4. Développement d'une architecture réflexive

### 9.2.4.1. Comment se fait l'auto-représentation du système ?

Etre capable de générer des informations sur l'état courant d'un processus en cours, cela signifie par exemple que l'interpréteur du langage qui dispose d'informations sur toutes les primitives du langage doit aussi être capable de construire des représentations de ces primitives lorsqu'elles sont instanciées (i.e une méta-représentation de ces primitives). De plus, il n'est pas suffisant de produire des informations sur l'état d'un système encore faut-il que cette représentation soit accessible. Comme on l'a dit précédemment le fait d'utiliser les mêmes schémas de représentation pour la partie réflexive et pour la partie exécution d'un système permet une interaction cohérente entre les deux niveaux de représentation. Il est cependant nécessaire d'établir des "frontières" entre ces deux niveaux de représentation. Ce que l'on peut illustrer de la manière suivante :





```
He said I'll do the job
He said : "I'll do the job"
```

Les guillemets ci-dessus introduisent cette différence de niveau entre les deux parties de l'énoncé. Dans le travail présenté dans DINOSOR (Fleury & al. 1992) on dispose de trois niveaux de représentation de la grammaire : une représentation associée à la grammaire initiale, une représentation associée à la grammaire normalisée et une représentation associée à la grammaire compilée. Chacune de ces représentations est attachée à un objet spécifique et les trois objets ainsi créés sont nettement dissociés.

### 9.2.4.2. Comment programmer la partie réflexive ?

Un système doit être capable de modifier ses propres auto-représentations. Il doit aussi être possible d'effectuer un va-et-vient permanent entre la phase d'exécution d'un processus et la phase de réflexivité liée à ce processus.

### 9.2.4.3. Quel lien existe-t-il entre le système et la partie réflexive ?

Les données liées au domaine de la réflexivité doivent être liées au système de telle sorte qu'une fois achevées les manipulations dans une phase d'activité réflexive, le retour à l'exécution d'un processus du système implique que le système ait accès à ces données mises à jour (et réciproquement). Un travail de maintenance permanente est donc nécessaire pour la cohérence des deux niveaux de représentation. Il faut pouvoir disposer de programmes qui peuvent modifier le flot de contrôle quand ils sont en train de fonctionner. Il ne suffit pas de constater que quelque chose ne fonctionne pas dans le déroulement d'un programme, il faudrait pouvoir y remédier : on retrouve là le problème de la mise au point des programmes ou encore, dans le cas des analyseurs, le fait de savoir pourquoi ça ne marche pas comme on le voudrait... Avec 3-Lisp, il est possible d'indiquer comment changer le processus d'interprétation : si le programme détecte un certain type d'erreurs, il peut changer le comportement de l'interpréteur de façon à remédier à cette erreur sans passer par un programmeur pour modifier le programme.

## 9.2.5. Utilisation de la réflexivité

Il faut pouvoir se servir de ce que l'on a obtenu dès qu'on l'a obtenu. Pour passer d'une étape à une autre, on se sert des résultats que l'on vient d'obtenir. On verra infra comment la mise en place d'une représentation de la grammaire sous une forme normalisée s'est révélée primordiale pour la mise au point des modifications apportées au code initial. On doit donc pouvoir accéder aux représentations du système lui-même à tout moment de l'activité du système. Ce qui doit être accessible au processus de réflexivité :

- Les programmes du système.
- Les données du système.
- Le programme d'un processus en cours d'exécution.
- Les données d'un processus en cours d'exécution.
- La réflexivité rend possible le contrôle du flux des calculs de manière modulaire.

Un système réflexif peut comporter plusieurs modules et l'un d'entre eux modifie un autre. C'est le cas d'un système capable d'apprendre contenant une partie performance qui résout les problèmes et une partie apprentissage qui modifie la précédente au vue de ses résultats. Le

système se modifie même si en réalité la partie qui modifie est distincte de la partie modifiée. Il est clair que pouvoir accéder aux représentations des programmes d'un système pendant leur déroulement est un moyen efficace de contrôle. La maintenance, l'acquisition et la communication de l'information contenue d'un système est une tâche qui suppose une connaissance importante sur toutes les activités d'un système donné et les interactions possibles entre certaines activités.

## 9.2.6. Dangers de la réflexivité

Si un système est capable de modifier son comportement, est-ce que cela entraîne un transfert de contrôle définitif de l'activité du système vers lui-même? Il faut ainsi remarquer que la réflexivité peut entraîner une perte d'informations. En effet si un système peut modifier son état courant quand il détecte une erreur, une incorrection ou quand les valeurs attachées à certaines variables provoque cette modification de l'état courant, on perd toute trace de l'environnement initial (à moins de mettre en place une nouvelle couche qui stocke les représentations des états précédents du système!). Or il arrive fréquemment que le programmeur (ou l'utilisateur) souhaite avoir accès à la représentation de l'environnement de l'état en cours du système pour évaluer par exemple la valeur attachée à une certaine variable. Ou, par exemple, lors d'une erreur du système qui nécessite sa réinitialisation, à la représentation du fichier généré par cet arrêt de la machine. Aussi l'auto-modification permanente d'un système peut entraîner une impossibilité d'accès aux représentations du système. On voit pourtant très bien l'intérêt que pourrait représenter l'existence de processus réflexifs notamment dans les périodes de mise au point de programme, c'est à dire à un moment où l'on souhaite avoir accès à tous les paramètres que le système en cours de mise au point manipule.

Un autre problème dans l'utilisation des processus réflexifs est celui de la gestion efficace des interactions entre les différents niveaux de représentation d'un système. Comment faire en sorte qu'il n' y ait pas de confusion entre les informations liées au système propre et les informations liées aux processus de réflexivité ? Dans le cas de l'analyse automatique, OLMES (Habert 1993) opère une transcription de la grammaire initiale. En plus de cette représentation de la grammaire, il faut pouvoir produire une représentation dédiée à la réflexivité et contenant les mêmes informations. Il convient donc d'établir une distinction nette entre ces deux niveaux de représentation. Même si le système en question est construit de telle sorte que les différents niveaux de représentation manipulent des primitives distinctes, il n'est pas évident que l'existence de ces deux niveaux de représentation rende la compréhension de tout ce qui est en jeu dans le système très claire pour le programmeur et/ou l'utilisateur.

## 9.2.7. Comment programmer la partie réflexive?

Les langages à architecture réflexive construisent une auto-représentation du système. Cette méta-représentation doit prendre en compte la formalisation des aspects sémantiques, syntaxiques, opérationnels, des phénomènes d'inférence propres au système. L'auto-référence peut-être implicite ou explicite. On parle de réflexivité implicite si le bon fonctionnement du système nécessite une activité réflexive permanente. Dans ce cas il est impératif d'assurer une cohérence constante et systématique entre le système et sa méta-représentation. (Le système s'applique à lui-même mais il ne le sait pas) On parle de réflexivité explicite si les processus réflexifs sont utilisés comme des fonctionnalités possibles mais non obligatoires. Ils interviennent que si l'on fait explicitement appel à eux. Il faut ensuite établir le type de lien entre le système et sa méta-représentation. Deux types de solution sont possibles :

- La réflexivité procédurale, le lien est implémenté.
- La réflexivité déclarative, le lien est spécifié.





### 9.2.7.1. Réflexivité procédurale

L'auto-représentation du système sert à implémenter le système. Le problème de la maintenance de cohérence des deux niveaux de représentations ne se pose donc pas. Il y a une seule représentation du système qui sert à l'implémenter et aussi à agir et réfléchir sur lui-même.

### 9.2.7.2. Réflexivité déclarative

Un système développé avec une architecture à réflexivité déclarative doit être capable de prendre en compte au moment de son exécution des contraintes et de poursuivre son exécution après avoir satisfait ces contraintes. Le système fournit les moyens de créer des méta-représentations et de récupérer le référent de celles-ci. La cohérence des deux niveaux de représentation est assurée par un mécanisme qui maintient la relation entre ces deux niveaux. Le niveau méta-représentation est dédié à la description du système.

## 9.2.8. Vers une mise en place de processus réflexifs dans une analyse automatique

Le choix de la méta-représentation à adopter est extrêmement important car il implique la mise en place d'outils réflexifs spécifiques. Il faut rappeler que les langages à architecture réflexive permettent une plus grande modularité dans l'implémentation. Ces langages doivent être ouverts à toute extension qui permette de fournir de l'information sur l'activité d'un processus quelconque d'un système. Sans revenir sur les problèmes cruciaux qui existent pour la construction de langage à architecture réflexive, on peut quand même conclure provisoirement en disant que :

- La réflexivité est un outil utile pour toute activité de surveillance, de mise au point, de présentation raisonnée des résultats.d'un système.
- Les processus réflexifs doivent être limités et déterminés par des choix précis : les besoins de l'utilisateur ou du programmeur définiront ces processus. Leur implémentation doit d'ailleurs être introduite de manière modulaire et/ou ponctuelle.
- La réflexivité est envisageable dans les programmes orientés Objet (PàO et PàP) en raison de caractéristiques propres à ces langages : ce type de langage permet de développer la mise en place de méta-représentations, comme nous le verrons infra. Nous verrons plus loin comment mettre en place des processus réflexifs au sein d'un analyseur syntaxique développé avec le langage orienté objet CLOS puis avec le langage à prototypes SELF.

Dans le cas de l'analyse automatique, les processus réflexifs peuvent être développés pour mettre en place des activités de suivi ou de mise au point de l'analyse. Pour cela, il importe de construire différents niveaux de représentation des connaissances manipulées par l'analyseur. On peut par exemple construire plusieurs représentations de la grammaire utilisée par l'analyse à différents stade de celle-ci. On peut aussi envisager de construire des niveaux de représentations dédiés au pistage des transferts d'informations entre les différents acteurs de l'analyse. Les outils mis en place doivent présenter, de manière raisonnée, les différents états pertinents de l'analyse.

## 9.3. Qu'est-ce qu'on ne peut pas faire sans la réflexivité ?

*"Le thème de la représentation des connaissances envahit aujourd'hui le champ des travaux en IA. Les ambiguïtés n'en sont pas pour autant levées : de quelles* ***connaissances*** *s'agit-il? Et qu'appelle-t-on* ***représentation*** *? Savoir répondre à une liste de questions ne signifie pas forcément* ***connaître*** *un sujet. Toute vraie connaissance suppose l'usage à bon escient des informations qui sont les nôtres. Quant aux représentations qu'on pourra se faire de ces informations, elles dépendront de nombreux facteurs dont chacun, isolément, pourra les orienter différemment : la situation, le domaine concerné, la personne visée, les notions organisant la connaissance, le* ***langage*** *dont on dispose,.... Les classifications de savoir sur un domaine n'ont ainsi d'autre vertu que celle de nous aider à* ***ranger nos idées.*** *Comment ensuite les organiser, les orienter, tel est effectivement le problème des métaconnaissances nécessaires pour apprendre à savoir utiliser les connaissances et, par suite, à* ***délier*** *les moteurs d'inférence inhérents à de multiples systèmes experts"* (Vignaux 1991).

La tradition langagière considère souvent qu'il est nécessaire de "prendre du recul" pour évaluer le cours des choses. Il va de même pour la mise en place de dispositifs informatiques qui tentent de réaliser une représentation matérielle de l'analyse de certains faits de langue. L'utilisation qui est faite de la réflexivité dans les langages informatiques de haut niveau est de permettre de "parler des objets" manipulés (par les programmes mis en oeuvre dans ce langage) plutôt que de parler aux objets (via l'envoi de messages). Dans certaines étapes de développement d'un programme, il est souhaitable de pouvoir examiner la structure des objets et donc de disposer d'outils qui permettent l'activité réflexive sur le programme en cours de développement. Cette étape peut aussi se révéler souhaitable lors de l'exécution d'un programme en vue d'améliorer les processus mis en oeuvre par le programme[1].

[1] Il reste un choix pour le programmeur entre une écriture pragmatique, plutôt behavioriste et une écriture réflexive : ce choix peut être déterminé par des raisons nombreuses et variées (à expliciter), dans un but défini (Pourquoi faire?).





## 9.4. Connaissance et réflexivité en IA

### 9.4.1. De la méta-connaissance

Comme il est très difficile d'envisager d'inclure au sein d'un même module l'ensemble des connaissances que l'on peut avoir sur le langage naturel, il semble nécessaire d'envisager une expression déclarative des connaissances pour le TALN. On sépare ainsi les connaissances de leur mode d'emploi. Une telle approche implique donc que le système qui les utilise puisse avoir des connaissances pour manipuler ces connaissances (pour recréer le lien entre les connaissances et leur mode d'emploi, il s'agit ici de construire le mode d'emploi). Quand un système raisonne sur ce qu'il fait, il lui faut des connaissances sur les connaissances, appelées méta-connaissances. Un système doit pouvoir manipuler les connaissances dont il dispose afin d'expliquer ce qu'il a trouvé, d'apprendre en comprenant les raisons de ses succès et de ses échecs. Cela demande de travailler à deux niveaux : une niveau de base dans lequel on essaie de résoudre un problème et un niveau méta dans lequel on observe ce que l'on fait pendant que l'on résout le problème en question. Il est souvent bon de manipuler l'énoncé pour résoudre un problème. L'énoncé d'un problème est un sujet d'étude, étude qui se fait grâce à des métaconnaissances. Il en est de même pour notre propre utilisation du langage naturel. Nous ne naissons pas avec la connaissance d'une langue mais avec des métaconnaissances qui nous permettent d'apprendre une langue. Un nouveau problème fondamental surgit à ce stade de développement d'une approche déclarative de manipulation des connaissances. L'utilisation d'un très grande nombre de modules de connaissance implique un très grand nombre de méta-connaissances sur ces modules de connaissances pour pouvoir les manipuler. Il s'agit donc dans une telle approche de découvrir des ensembles de métaconnaissances pour traiter les connaissances déclaratives. Cette tâche n'est évidemment pas triviale. D'ailleurs (Pitrat 1993) souligne très bien la difficulté de travailler sur les métaconnaissances dans le cadre du langage naturel : dans ce domaine de savoirs, il est en effet difficile de manipuler et de maîtriser un grand nombre d'éléments de manière efficace. Le problème principal étant de trouver des métaconnaissances capables de produire de nouvelles connaissances.

### 9.4.2. Faciliter la production de méta-connaissances sur les entités linguistiques et l'analyse : Illustration avec OLMES (1993)

OLMES[2] (Habert 1991b) est un analyseur général pour le langage naturel. Il peut être utilisé pour des grammaires hors contexte comme pour des grammaires d'unification (Shieber 1986a). Il appartient à la famille des analyseurs à graphe actif. OLMES est écrit en CLOS - Common Lisp Object System (Keene 1989), (Steele 1990) - le système à objets de Common Lisp. Le recours à la programmation à objets, et particulièrement à l'héritage multiple, a permis la production de points de vue différents sur le parsage. A chaque occurrence de symbole de la grammaire correspond une classe. Une règle est représentée de la façon suivante : une classe est créée pour la partie gauche. Elle en mémorise l'étiquette. Dans une version de OLMES aujourd'hui modifiée, c'etait la super-classe de classes représentant les éléments de la partie droite, qui, elles, mémorisent l'étiquette de chacun de ces éléments. La figure qui suit donne le sous-graphe de classes traduisant `GN -> Det N Adj.` `GN7`[3] est la super-classe des classes `Det8`, `N9` et `Adj10`.

2 Il est utilisé au Laboratoire d'Etude des Textes Politiques (INaLF - CNRS) pour catégoriser et structurer des corpus et sert de base à un projet de comparaison d'outils d'analyse de texte, projet commun à l'ELI (ENS de Fontenay St Cloud) et au Centre d'Analyse de Textes par Ordinateur de l'UQAM (Université du Québec à Montréal).

3 Le recours à l'héritage peut surprendre : un élément droit de règle est dans une relation Partie-De avec la partie gauche de la règle, et non dans une relation Est-un qui est la relation que traduit habituellement l'héritage. Ce choix avait pour but de faciliter la représentation d'une règle comme un tout structuré, en utilisant des traits standards (réflexifs d'ailleurs) de CLOS. Chaque classe connait en effet ses super-classes et ses sous-classes directes. Concrètement cela signifie que chaque partie gauche de règle connait les éléments de partie droite et inversement. A partir de tout élément d'une règle, il est ainsi possible à l'exécution de reconstituer la règle entière. Une représentation qui n'introduise pas une telle ambiguïté (Habert & Fleury 1993) a été mise en place depuis et sera présentée infra.

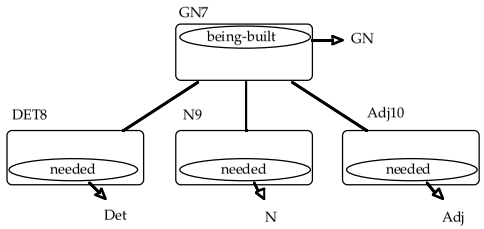



Figure 9.2 : Le sous-graphe de classes représentant `GN -> DET N ADJ`.

Une analyse montante de gauche à droite se déroule de la manière suivante. Chaque fois qu'est rencontré un mot ayant la même étiquette que l'une des classes correspondant à un terminal premier élément de partie droite de règle, une instance de cette classe est créée. Par exemple, si un Dét(erminant) est rencontré, une instance de `Det8` est créée, qui cherche le Dét(erminant) en cause (a) et le trouve. Cette instance crée alors une instance de la classe suivante dans le réseau, `N9`, qui cherche un N(om) et domine le Dét trouvé (b). Si l'instance de `N9` trouve un N(om), elle crée à son tour une instance de `Adj10`, cherchant un Adj(ectif) et dominant le Dét et le N trouvés aux pas précédents (c). Si cette dernière instance trouve un Adjectif, elle crée une instance de `GN7`, qui domine les mots ainsi reconnus (d), et qui déclenche de nouvelles règles, par le biais de la création de deux instances de `GN2` et `GN15`. Et la suite de l'analyse se poursuit de la même façon.

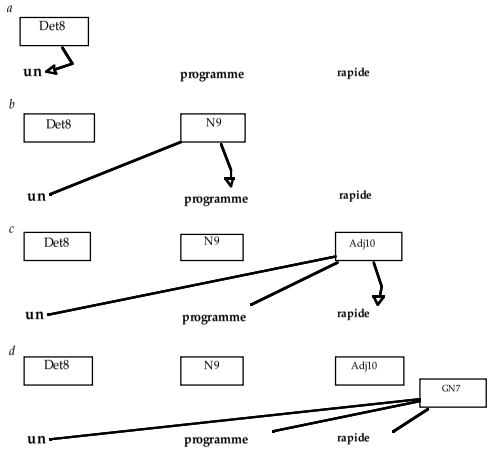



Figure 9.3 : Analyse par les instances d'un réseau de classes.

L'important est ici que l'entité "règle" est représentée en tant que telle, et peut donc être manipulée. La distinction est nette entre les choix que peut faire l'utilisateur en termes d'utilisation de la grammaire : direction d'exploration (gauche-droite, droite-gauche, bi-directionnelle) et stratégie (ascendante, descendante), choix qui se traduisent par des chaînages différents des classes représentant les éléments de la grammaire, et la structuration fondamentale de la grammaire par règles. Par ailleurs, une entité "méta-grammaire" est également créée, qui donne accès aux entités "règles" par le biais de leurs parties gauches.





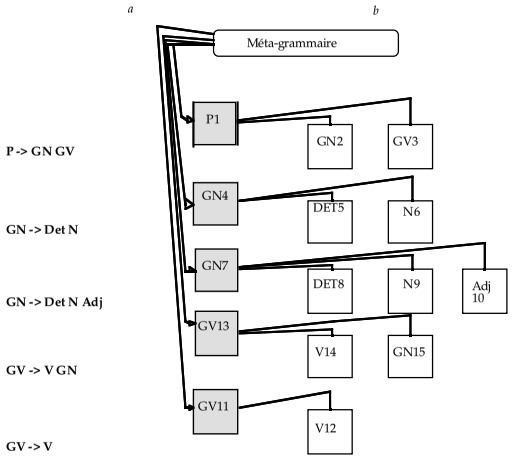



Figure 9.4 : Méta-grammaire et règles.

En OLMES, il y a donc une bijection entre les entités présentes à l'exécution et les éléments des règles définies par le linguiste, comme dans (Yonezawa & Ohsawa 1990). La transformation opérée dans (Yonezawa & Ohsawa 1990) est cependant à sens unique : elle manifeste le lien entre le comportement des unités de traitement et la grammaire de départ, elle ne permet pas pour autant une rétro-action sur la grammaire de départ à partir de l'observation de ce comportement. Le fait de constituer des méta-représentations des règles et de la grammaire en tant que telles rend possible au contraire en OLMES une telle action en retour. Toutefois, avant d'explorer les potentialités d'une telle architecture, il convient de réinterroger les notions de méta-connaissance et de réflexivité selon les cadres qui peuvent y faire appel.

## 9.4.3. Intelligence artificielle, formalismes d'unification, langages de programmation : des réflexivités distinctes ?

Le choix d'une architecture donnée pour un analyseur n'est pas purement technique. Il incarne également une certaine "idée" de la langue et de son traitement, tant naturel qu'automatique, et, partant, fixe le rôle que peuvent y avoir méta-connaissances et réflexivité. A l'inverse, il est nécessaire d'examiner le détail des architectures pour faire sortir les notions de réflexivité et de méta-connaissances de leur abstraction, et pour montrer les conditions techniques précises de leur emploi, au delà des espoirs ou des agacements que la perspective réflexive peut susciter. Ainsi se justifie la minutie du développement qui suit : limiter autant que faire se peut le risque de prendre ses désirs pour la réalité. Certains parseurs entendent fournir des représentations des structures de données présentes à l'exécution aussi "proches" que souhaitable des entités lexicales et syntaxiques définies par les utilisateurs linguistes. L'extension de telles architectures permet de produire des méta-connaissances sur les données linguistiques et éventuellement sur leur utilisation. Peut-être pourrait-on en effet contraster utilement les approches issues de l'intelligence artificielle, celles découlant des formalismes d'orientation linguistique et enfin celles autour des langages de programmation les plus évolués. En effet, si la recherche sur la réflexivité constitue une dimension de l'intelligence artificielle depuis une quinzaine d'années, si depuis près de dix ans, elle représente un courant dans la conception des langages de programmation, par contre, elle semble marginale dans l'évolution récente de la linguistique informatique. C'est ce constat que nous voulons

développer et questionner, en restant bien conscients des passerelles et des interférences entre des problématiques que nous présentons comme distinctes, voire divergentes.

### Domaines de la réflexivité.

Il n'est pas sûr que tous les types d'analyse aient à gagner à recourir à la réflexivité. Quel pourrait être son rôle dans un analyseur LR, par exemple ? Dans les analyseurs pour langages de programmation, une fois la grammaire définie, on n'a pas de raison, a priori, d'y revenir. On peut ne s'intéresser qu'au résultat : le compilateur produit pour le langage en cause. Pour le langage naturel, c'est rarement le cas. Les connaissances utilisées par l'analyseur sont sans cesse à affiner. La perspective réflexive paraît donc plus fructueuse pour les domaines où les connaissances résistent à la formalisation.

### Constituer et articuler des points de vue.

Le système multi-agents CARAMEL (Sabah 1990, 1993) fournit un bon exemple. Il est destiné à réaliser diverses tâches liées au traitement automatique du français. Il part de l'hypothèse que le type de connaissances à employer pour traiter une phrase ne peut être déterminé a priori, ce qui exclut un enchaînement fixe de couches (morphologique, syntaxe, sémantique, pragmatique, par exemple), et également une intégration fixée à l'avance des divers types de connaissances. Dans le but de réaliser *"une architecture où chaque module [puisse] avoir accès aux connaissances et aux résultats d'autres modules selon ses besoins"* (Sabah 1993), CARAMEL comprend alors un ensemble d'agents, simples ou complexes, qui sont autant de systèmes-experts coopérant à la construction du sens de l'énoncé. Les agents complexes peuvent déclencher d'autres agents. Ils constituent autant de *"méta-systèmes contrôlant un ensemble d'agents pour l'accomplissement de leur but"* (ibid.). Un superviseur général contrôle l'analyse. La réflexité est ici fondamentale : aussi bien au niveau du superviseur général qu'à celui de n'importe quel agent complexe, il s'agit d'évaluer les représentations déjà produites, les capacités des agents déclenchables par l'agent complexe concerné et partant, les meilleurs moyens d'obtenir le but fixé, en déclenchant tel ou tel agent, en gérant le va-et-vient d'informations entre les agents contrôlés, et en adaptant la stratégie selon les informations reçues des agents déclenchés. Par ailleurs, *"dans sa mémoire de travail, le superviseur construit /./ une représentation globale du déroulement dynamique du raisonnement (une indication des agents qui ont été nécessaires pour mener à bien une tâche donnée), ce qui lui permet de donner par la suite des explications sur la stratégie qui a été employée"*. CARAMEL permet d'articuler différents savoirs linguistiques et non-linguistiques.

## 9.4.3.1. IA : la réflexivité pour la modularisation et l'articulation des savoirs

L'intelligence artificielle, plus directement confrontée aux relations entre les différents composants du savoir linguistique, et entre celui-ci et le savoir "sur le monde", a cherché à articuler ces différents types de savoirs, en utilisant par exemple l'activation de modules spécialisés, la mise en oeuvre ou le retrait de tel ou tel paquet de règles, et à essayer, depuis longtemps, de recourir aux méta-connaissances et à la réflexivité. Le développement des systèmes-experts (Laurière 1987) en particulier a très tôt poussé à représenter les connaissances que les experts ont sur leurs propres connaissances, en les formulant sous la forme de méta-règles guidant le moteur d'inférence. S'est avérée aussi fondamentale la capacité pour un système-expert donné de pouvoir expliquer comment il arrive à une conclusion donnée.

## 9.4.3.2. Les formalismes d'unification : peu de réflexivité





Au rebours de cette représentation modulaire et réflexive des connaissances et des traitements, les formalismes dits d'unification tendent à intégrer au long de l'analyse les différentes facettes de l'analyse linguistique. Ainsi dans la grammaire syntagmatique guidée par les têtes, HPSG (Pollard & Sag 1987), un "signe", entrée lexicale ou syntagme, comprend volets phonologique, syntaxique et sémantique. En tant que tels, ces formalismes ne favorisent guère, voire ne permettent pas, la séparation en différents modules explicites, destinés à interagir. Tout se passe comme si l'accord sur la manière de représenter les connaissances linguistiques et de s'en servir que constituent les formalismes d'unification s'était accompagné d'un certain enfermement sur syntaxe et sémantique pris dans un sens relativement restreint, et parallèlement, d'une mise entre parenthèses de l'articulation des savoirs linguistiques entre eux et avec d'autres types de savoirs. La possibilité de fournir sous une forme relativement déclarative un certain nombre de contraintes sur la combinaison des constituants entre eux a relégué paradoxalement au second plan la réflexion sur les stratégies d'utilisation des connaissances linguistiques. Si bien qu'au total, l'intégration d'une approche réflexive aux analyseurs basés sur des formalismes d'unification paraît difficile, et en danger de rester "accessoire".

Les formalismes d'unification comprennent un certain nombre de principes, qui contrôlent la bonne formation des différents constituants. Au sein d'un formalisme donné, certains de ces principes peuvent être pensés comme universels, tandis que d'autres sont vus comme propres à une langue donnée. On trouve ainsi dans la grammaire lexicale-fonctionnelle, LFG (Bresnan 1982), les principes de complétude et de cohérence (toutes les "places" d'un prédicat verbal doivent être occupées, et il ne doit pas y avoir d'"argument" surnuméraire), dans la grammaire syntagmatique généralisée, GPSG (Gazdar & al. 1985), des principes d'instanciation de traits d'accord, de pied et de tête, ou les règles de précédence linéaire. Au delà de la règle générale que les principes doivent tous être simultanément satisfaits, la réalisation concrète de leur interaction, par exemple leur ordonnancement peut se révéler fort complexe, comme le montre S.Shieber (1986b). Pour tout dire, on voit réapparaître les problèmes de contrôle que la déclarativité des règles de réécriture annotées avait poussés dans l'ombre. Le Principle-Ordering Parser de S.Fong et R.Berwick (1991) a précisément pour objectif de permettre de tester des ordonnancements possibles des principes de la théorie du Gouvernement et du Liage en tenant compte de leurs dépendances logiques : *"Le PO-Parser a été délibérément construit d'une manière extrêmement modulaire pour permettre un maximum de flexibilité dans l'exploration des ordonnancements alternatifs de principes"*. Les principes d'une théorie gagneraient sans doute à changer de statut et à constituer un méta-savoir explicite et utilisable en tant que tel au cours de l'analyse. Un tel méta-savoir pourrait veiller par exemple à l'application des différents principes d'unicité de la théorie du Gouvernement et du Liage (un seul noyau, une seule catégorie X par domaine, un seul support de fonction dans S etc.). Face aux deux réalisations possibles du GV avec objet direct en français, clitique préposé ou GN postposé au verbe, le méta-savoir incluant ce principe d'unicité éliminerait une configuration des hypothèses possibles si l'autre s'est réalisée.

## 9.4.3.3. Réflexions sur la réflexivité

### Réflexivité approchée / implicite / explicite

Bascule d'un mode simple à un mode plus élaboré, envisageant davantage d'hypothèses, pondération des hypothèses, traitement des exceptions en tant que telles, adaptation aux énoncés traités, ces processus trouvent dans l'immédiat des approximations dans des analyseurs actuels : parsage auto-adaptatif (Lehman 1992), parsage probabiliste (Liberman 1991), dispositifs d'acquisition lexicale (Zernik 1991) ... Sans contester les apports de ce courant empiriste, numérique, non-symbolique, on peut faire l'hypothèse que l'intégration d'une approche réflexive dans la conception d'analyseurs peut permettre également de mettre en oeuvre ces dispositifs et ces fonctionnements, en leur donnant un statut plus clair. A ce titre,

la comparaison entre CARAMEL et TALISMAN (Stefanini & al. 1992) est éclairante. TALISMAN repose aussi sur une architecture multi-agents, pour les mêmes raisons que CARAMEL : *"compose[r] différentes phases d'analyse"*, *"ordonner convenablement les divers sous-ensembles de règles et /./ les activer au bon moment"*. Les deux systèmes s'inscrivent donc dans le cadre de l'Intelligence Artificielle Distribuée (Bond & Gasser 1988), où l'*"intelligence est répartie entre des agents. Ceux-ci sont des entités autonomes capables d'agir rationnellement sur elles-mêmes et sur leur environnement en fonction de leurs observations, de l'état de leurs connaissances et de leurs interactions"* (Stefanini & al. 1992). Alors que CARAMEL repose sur des tableaux noirs pour le partage d'informations, TALISMAN s'appuie sur la communication par messages entre agents. Pour éviter les problèmes de synchronisation, de concurrence, de partage de ressources que fait naître une trop grande autonomie des agents, TALISMAN se voit contraint d'ajouter des "lois". *"Les lois régulent et modifient les échanges de messages entre les agents. Elles sont explicites et modifiables "* (ibid.). Ces lois jouent en fait le rôle des superviseurs de CARAMEL : elles *"activent un agent spécialisé, en fonction d'indices prélevés dans le texte et de caractéristiques d'un problème d'analyse particulier"* (ibid.), elles évitent certains conflits entre agents et en gèrent d'autres. Il s'agit bien de méta-savoirs. Leur reconnaître ce statut pose de manière plus nette la question de la constitution de la méta-connaissance au cours des traitements, c'est-à-dire en définitive le problème d'architectures réflexives qui aident à dégager des modes de coopération optimaux entre les agents, en notant par exemple les conflits rencontrés, les duplications d'informations etc. Thématiser la réflexivité en tant que telle paraît en définitive plus fructueux que d'y recourir implicitement ou d'utiliser des approximations.

**Réflexivité naturelle et réflexivité artificielle**

Nous sommes restés précédemment (relativement) agnostiques sur le rapport entre la réflexivité "naturelle" dont semble faire preuve le locuteur lors de l'analyse d'un énoncé, et que marque la constante activité métalinguistique, et la réflexivité qui nous semble devoir intervenir dans les analyseurs. A cela, au moins trois raisons. Les arguments psycho-linguistiques sont tout d'abord fréquemment suspects : la diversité tant des hypothèses que de leur utilisation rend prudent. En outre, mimer le comportement humain n'est pas forcément la meilleure solution. Enfin, ce sont plutôt les raisons internes à la conception d'analyseurs de recourir à la réflexivité qui nous paraissent pertinentes. C'est a priori dans le cadre d'une vision modulaire de l'analyse automatique que les apports de la réflexivité sont les plus évidents. Pourtant, le courant actuel des grammaires d'unification peut également en bénéficier. Pour reprendre les termes de P. Maes (1987a), *"une implémentation réflexive ne contribue pas directement à la résolution des problèmes du domaine représenté par le système. Elle contribue plutôt à l'organisation interne du système ou à son interface avec le monde extérieur. Son but est de garantir un fonctionnement efficace et harmonieux des calculs liés au problème traité"*. Les divers cadres de pensée des travaux sur la réflexivité offrent donc des perspectives stimulantes. Pour reprendre P. Maes (1987a) : *"le calcul réflexif est tellement constitutif des systèmes en vraie grandeur qu'il doit être développé comme un outil fondamental dans les langages de programmation"*. Pourquoi n'en irait-il pas de même pour l'analyse automatique du langage naturel ? En gardant en mémoire l'avertissement de J. Pitrat (1992) *"Travailler sur l'utilisation des métaconnaissances est un détour qui aura des conséquences pour les systèmes manipulant les langues naturelles, mais nous ne devons pas en attendre des résultats spectaculaires à court terme"*.





# 9.5. Tradition de la programmation réflexive

## 9.5.1. Réflexivité et Méta-niveau dans les langages orientés Objet

On étudie ici certains langages dotés d'une capacité d'"introspection" permettant à un système conçu dans de tels langages de s'adapter et de s'optimiser. Est visée une adaptation non seulement statique mais aussi dynamique (au cours de l'évolution des calculs). On entend en outre contrôler le comportement d'un système non pas à un très bas niveau d'expression mais à un niveau au moins équivalent à celui du système lui-même. Un méta-niveau représente et décrit alors l'exécution du niveau de base, c'est-à-dire le déroulement du programme. Il est possible que le méta-niveau ait lui-même un méta-niveau et ainsi de suite, ce qui construit éventuellement ce qu'on appelle une tour réflexive (d'autant qu'il y a identité de langage entre niveau de base et méta-niveau).

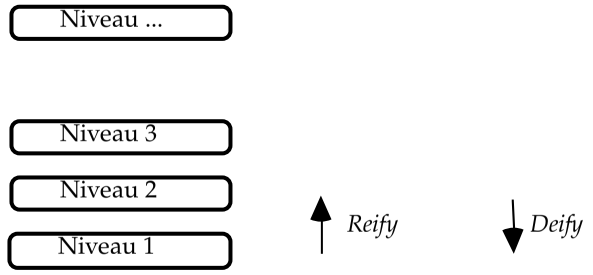



Figure 9.5 : Une tour réflexive.

Le fait qu'un langage soit explicitement doté d'une méta-représentation de lui-même permet par exemple d'obtenir un langage qui puisse évoluer de manière contrôlée, et servir ainsi de terrain d'expérimentation tout en gardant les avantages de la standardisation. C'est le cas par exemple en CLOS où les classes sont elles-mêmes des instances de métaclasses qui déterminent leur comportement et leur structure. Créer de nouvelles métaclasses permet alors d'introduire de nouveaux comportements (autres schémas de calcul de l'héritage multiple, classes connaissant leurs instances ...), sans modifier pour autant le fonctionnement normal du système (Cointe & Graubé 1988, Graubé 1989, Kickzales & al. 1991). La plupart des langages orientés Objets mettent en avant les notions conjointes de Classes et Métaclasses. Cette approche s est révélée une solution adéquate pour mettre en place des mécanismes de réflexivité (cf infra avec CLOS), que l on peut qualifier de réflexivité structurale. Cela signifie que toutes les données présentes dns le système sont réifiées (données et programmes). Le développement des langages à prototypes tels Self a mis en avant de manière plus précise la notion de reflexivité comportementale qui introduit la notion de réification de l exécution d un objet : cet aspect n a pas été résolu dans le premier type de langage et est en cours de développement avec le second (Self (the Self Group 1987-1996), Moostrap (Cointe & Mulet 1993)). Il importe de préciser tout de suite que le développement de réflexivité et d architectures à méta-niveaux ne signifie qu il soit nécessaire de développer ou utiliser des langages qui soient eux-mêmes complètement réflexifs. Il s agit plus précisément que ces langages permettent de spécifier ou d engendrer des activités réflexives pour répondre à des sollicitations précises et à des besoins particuliers prédéfinis.

## 9.5.2. Clos et la Réflexivité

CLOS encourage un style de programmation qui fasse de chaque classe une sorte de boite noire à laquelle les utilisateurs n'ont accès que par l'interface fourni par les fonctions génériques sans qu'ils puissent connaître le détail de l'implémentation. CLOS ne force cependant pas à séparer nettement les comportements "privés" de la classe, accessibles par ses instances et les instances de ses sous-classes, et les savoir-faire "publics", utilisables par le tout-venant, sauf à recourir au Meta-Object-Protocol (MOP par la suite).

### 9.5.2.1. Méta-Object

### 9.5.2.2. Le MOP de CLOS

CLOS est défini en termes d'objets : aux classes, aux définitions de champs, aux méthodes, aux fonctions génériques correspondent des méta-objets qui réalisent le méta-protocole[4]. Ce méta-protocole permet à l'utilisateur de se définir ses propres méta-classes (par exemple des classes qui connaissent toutes leurs instances) et d'écrire des extensions portables de CLOS. L'apport CLOS est que ces sur-langages gardent comme sous-bassement les potentialités de CLOS, tout en les redéfinissant. L'utilisation du MOP pour construire une couche supplémentaire de la représentation des informations dans l'analyse syntaxique faite par Olmes a permis d enrichir considérablement la connaissance de l'utilisateur sur ce qui est en jeu dans l'analyse syntaxique (-> annexe chapitre 9 : trace d'une analyse). Le Méta-protocole de CLOS est construit de telle sorte que l utilisateur de processus réflexifs peut spécifier les éléments sur lesquels il a besoin de définir une utilsation de tels mécanismes. *"For this class, this slot, perform slot access in this different way"* plutôt que *"To all accesses, this different way"* (Kiczales 1990). La structure méta définie en CLOS s inscrit dans le même modèle que celui dans lequel CLOS est construit. Les éléments de base de CLOS (classes, méthodes, fonctions génériques) sont définis comme des objets : i.e, chaque classe, chaque fonction générique, chaque méthode est représenté sous la forme d un métaobjet. Le terme ɔClasse métaobjet réfère aux classes dont les instances sont des métaobjets. Chaque classe est représentée par une Classe métaobjet qui enregistre ses superclasses, les slots définis dans la classe et les slots hérités. Et tous les mécanismes qui définissent les comportements des objets (héritage de slot, structure des instances, accès au slot, initialisation d instance...) sont codés au niveau du métaprotocole par le biais de fonctions génériques qui agissent sur les Classes métaobjet.

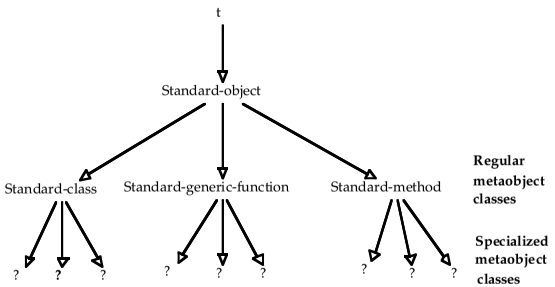



Figure 9.6 : Classes MetaObject en CLOS.

Il est possible de d'envisager de mettre en place des outils de "surveillance" de toutes les informations présentes au cours de l'activité d'un processus en cours. Par exemple si on envisage de ne manipuler au cours de l'analyse que des objets, il est possible d'établir un historique des

[4] *"Les protocoles méta-objets sont des interfaces qui donnent aux utilisateurs la possibilité de modifier de façon incrémentale le comportement du langage et son implémentation, aussi bien que la possibilité d'écrire des programmes dans ce même langage... L'implémentation de ce langage est elle-même structurée comme une programme orienté objet... Les éléments de base du langage de programmation, classes, méthodes et fonctions génériques sont accessibles comme le sont les objets"* (Bobrow & al 90).





accès aux champs des classes manipulées. Si les traits utilisés dans la grammaire sont associés à des objets, on peut ainsi tracer la circulation de ces traits au cours de l'analyse. Le MOP fournit cette possibilité en utilisant ce que (Bobrow & al. 1990) nomment le `Monitoring-Slot-Access`. Pour ce faire il faut définir un métaobjet. Les classes dont on souhaite faire un historique de l'accès à leurs champs héritent de cette classe. L'utilisation habituelle du protocole d'accès permet ainsi de mettre en place ce travail d'enregistrement des accès aux champs. La généricité étant ce qu'il convient d'utiliser sous CLOS, on peut aussi envisager de construire un compteur qui enregistre le nombre de fois où une fonction générique et les méthodes associées sont appelées. Pour ce faire on peut en effet définir au niveau du MOP des fonctions génériques et des méthodes qui réalisent cette incrémentation. On peut ainsi établir complètement l'historique des interactions entre les objets. Ces informations ajoutées aux informations fournies par le `Monitoring-Slot-Access` pourraient donc produire une importante trace de l'activité d'un analyseur syntaxique. Il serait alors possible d'établir un relevé de toutes les opérations exécutées et d'identifier par exemple les performances de ces opérations.

### 9.5.2.3. Les limites de `change-class`

Si on change la définition d'une classe donnée, cette opération peut conduire à perdre des informations mémorisées par des instances déjà créées (Habert 1996). A moins de définir des prologues ou des épilogues qui accompagnent les redéfinitions de classe (Ibid.). Ces accompagnements posent malgré tout des problèmes d'une extrême finesse, et impliquent une attention particulière pour mettre à jour les classes existantes. Surtout si on envisage de modifier dynamiquement les définitions des structures définies dans un processus de classification automatique ou de gérération dynamique des savoirs associées. De plus changer la définition des classes existantes nécessite un méta-regard sur le processus de représentation en cours pour évaluer les interactions ou concurrences à l'oeuvre dans les savoirs à représenter et dans les objets déjà exitants. Une maîtrise totale des savoirs en jeu implique un méta-contrôle sur les objets construits ou à construire (et donc les méta-outils associés) qu'il n'est évidemment pas trivial de définir. S'il est possible de changer la définition d'une classe et de propager ce changement aux instances d'icelle (en respectant les mises en garde précédentes), il est aussi possible de changer la classe d'une instance donnée (Ibid.). La fonction générique `change-class` est l'outil qui réalise cette opération. Cette fonction générique a malgré tout des limites. Elle opère un transfert d'information entre les attributs communs aux deux classes en jeu (les valeurs des attributs de la classe d'origine passent dans les attributs de la nouvelle classe), et initialise les attributs spécifiques à la nouvelle classe avec les valeurs définies par défaut dans celle-ci. Par contre les attributs spécifiques à la classe d'origine sont oubliés. Cette opération peut donc conduire à une perte d'information. Et il n'est pas sûr que cette perte d'informations soit justifiée/voulue dans tous les cas. On peut ainsi vouloir, pour une instance donnée, changer sa classe pour y préciser certaines caractéristiques que l'on ne possède pas encore (du type : `moulin` devient `+humain` dans `moulin à paroles`), tout en conservant certaines informations déjà présentes.

### 9.5.3. Self et la réflexivité

Comme on l a vu dans la présentation de Self faite précedemment, une des caractéristiques de Self est que les objets manipulés sont complètement déterminés par les comportements qui leur sont associés. Cette priorité de représentation permet une extension et une flexibilité dans le développement de nouvelles applications. Self dispose aussi d une facette fondamentale : on peut souhaiter pouvoir examiner ou manipuler les objets au cours d un processus de calcul. On utilise les objets comme des données et non plus seulement comme des entités associées à la représentation d un concept. Ce type de réflexivité permet de ɔparler à propos d un objet plutôt que de lui parler (via l envoi de messages en Self). Cette action est faite en Self avec des objets appelés miroirs.

### 9.5.3.1. Réflexion de structure en Self

**La notion de reflect**

Un objet Self est essentiellement un dictionnaire (association attributs et valeurs). Les attributs peuvent être constants ou modifiables, parents ou non, privés ou publics. Ces informations sont conservées dans des entités appelées `maps`. Une réflexion de structure est introduite en Self par des entités appelées `miroirs`. Ces entités rendent visibles au niveau du langage les informations contenues dans la représentation interne des objets. Un `miroir` sur un objet `x` permet essentiellement de faire des requêtes sur la représentation interne de cet objet. Dans la réalité, les `miroirs` sont des objets qui ne contiennent aucune information en eux-mêmes. Un `miroir` est simplement un receveur capable de répondre à un ensemble de primitives du système qui donnent accès en lecture aux informations sur l'objet.

### 9.5.3.2. Réflexion de comportement

Dans les langages à taxonomie de classes, les classes et les méta-classes définissent la réflexion de structure. Dans les langages à prototypes de type Self, la notion de classe n existe pas pour définir un modèle d auto-représentation des objets. (Cointe & al. 91) propose une implémentation de la réflexion de comportement en Self. Ce protocole est basé sur l utilisation de méta-objets pour décrire le comportement des objets lors de la réception des messages. A chaque objet Self est associé un méta-objet dont la fonction est d'expliciter la manière dont l'objet se comportera à la réception d'un message. Le problème de la régression infinie par le lien "méta" est réglé par l'introduction un méta-objet de base, `BasicMetaObject`, qui s'admet lui-même comme son propre méta-objet et qui définit les comportements de base pour la recherche des attributs et leur application. Le lien "méta" peut être explicite (via un attribut `metaobject`) ou implicite (dans ce cas l'objet a pour méta-objet `BasicMetaObject`).

A l'envoi d'un message, l'interprète effectue les opération suivantes :

1. Il envoie au méta-objet du receveur le message

```
`lookup: From: SendingMethodHolder:`
```

effectuant la recherche de l' attribut et retournant un descripteur de l'attribut représentant l'attribut trouvé.

2. Il envoie ensuite à ce descripteur d'attribut le message

```
`slotApplyTo: WithArgs:`
```

qui déclenche l'exécution du contenu de l'attribut trouvé.

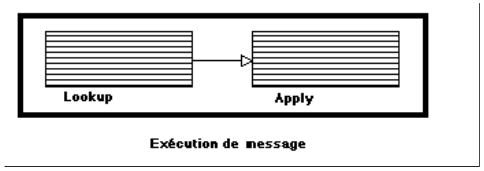



Figure 9.7 : Exécution de message en Self.





Ce protocole réifie l'exécution des messages en paramétrant la recherche des attributs et leur application. Les messages

```
`lookup: From:SendingMethodHolder:`
                et
`slotApplyTo: WithArgs:`
```

fournissent les points d'entrée nécessaires pour introduire les modifications voulues par l'utilisateur. Le protocole de réflexion de comportement est donc basé sur l'utilisation de méta-objets pour décrire le comportement des objets lors de la réception de message. L'exécution d'un message est décomposé en deux phases : une phase de recherche de l'attribut correspondant au sélecteur (`lookup`) et une phase d'application de l'attribut trouvé dans le contexte de l'objet receveur. Cette extension pose des problèmes fondamentaux sur le développement de protocole de réflexion de comportement : la notion de lien méta est à étudier, une réification totale des attributs semble difficile à mettre en place en Self. La réflexivité comportementale pose deux problèmes fondamentaux. D une part, ce type de mécanisme doit décrire le comportement d un objet agissant sur d autres objets. D autre part, on doit pouvoir disposer de moyens pour intervenir sur l exécution d un processus affectant les objets du système pour éventuellement modifier le cours des évènements.

### 9.5.3.3. Moostrap : un noyau réflexif pour un langage à prototypes

(Cointe & Mulet 1993) propose, avec le langage à prototypes réflexif MOOSTRAP, une nouvelle approche pour une étude de la réflexion de comportement dans un langage à prototype. Il ne s agit plus de construire une extension d un langage existant mais de construire directement un langage dans lequel le protocole de réflexion de comportement s inscrit comme un choix fondamental dans le développement d un langage à prototypes.

# 9.6. Outils disponibles pour le TALN

## 9.6.1. Un lien Méta

**Méta-classe : homogénéité du paradigme, toute entité est un objet, ie une instance d une classe.**

Définir un lien méta est assurément le plus sûr moyen pour pouvoir disposer de mécanismes qui coopèrent dans un système de représentation des connaisssances réflexif. La question fondamentale que pose ce type de système est le type de distinction qui doit exister entre le niveau méta et le niveau de base. On peut définir un système à architecture méta :

1 ) On met en place un niveau méta de référence. Il s agit de définir les mécanismes d inteaction de référence entre le niveau méta et le niveau de base.

2 ) On dispose de plusieurs méta-niveaux indépendants (par rapport au lien méta qu ils entretiennent avec le niveau de base) et il s agit dans ce cas de définir les interactions entre ces différents méta-niveaux.

Dans le premier cas de figure, on définit un certain méta-niveau et on s y tient, et parallèlement on met en place les processus réflexifs sur le niveau de base. Dans le second, chaque type de problème étudié dans le niveau de base définit son propre méta-niveau et les processus réflexifs évaluent les interactions entre ces méta-niveaux.

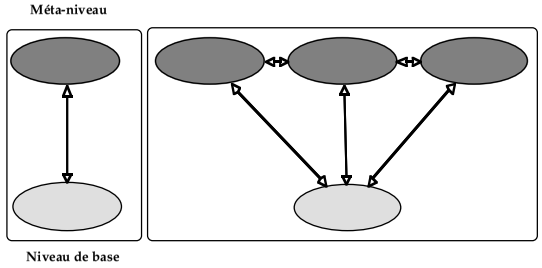



Figure 9.8 : Un lien Méta.

## 9.6.2. Analyse Automatique et Méta-niveau

OLMES offre des points d'entrée distincts pour le linguiste qui affine la grammaire et pour le programmeur qui met au point l'analyseur. Les entités actives lors de l'analyse combinent en effet de nombreuses facettes structurelles ou comportementales : structure de base d'une unité de traitement (type de constituant cherché...), structure et comportement nécessaires aux relations de dominance, mémorisation d'hypothèses syntaxiques et capacité à les déclencher si nécessaire, direction d'exploration (gauche-droite ou inversement), étape dans la vérification d'une règle ou construction du constituant correspondant à la vérification complète de la règle, chaînage avec les étapes précédente et suivante... L'héritage multiple sert alors à organiser le comportement et les structures utilisés par les éléments de la grammaire. Chaque classe créée pour une occurrence de symbole dans la grammaire hérite en effet de différentes classes abstraites qui représentent les différentes facettes du comportement et de la structure de cet élément de la grammaire pendant l'analyse. L'héritage a pour rôle ici de structurer les méta-connaissances. La tâche du compilateur est alors, au vu de la grammaire et des choix de l'utilisateur, de donner les ancêtres adéquats à chaque classe correspondant à un élément de la





grammaire, de manière à lui fournir les structures et les comportements pertinents. Ce recours intensif à l'héritage multiple fait coexister des points de vue différents et indépendants sur une règle de grammaire représentée par un graphe de classes et sur son utilisation possible. La multiplicité, organisée, des points de vue possibles sur le réseau de classes permet de créer différents protocoles d'accès nettement distincts. L'utilisateur en particulier doit seulement connaître l'existence du réseau de classes et d'une bijection entre les occurrences de symboles dans la grammaire et les noeuds de ce réseau. Il lui faut connaître en outre un sous-ensemble des facettes que peut comporter un noeud de ce réseau, les facettes les plus proches de sa propre représentation de ce qu'est une grammaire. Enfin, il lui faut connaître la partie correspondante du protocole qu'emploient au moment de l'analyse les entités produites par la compilation, pour pouvoir y greffer traçage et pas à pas, par le biais de prologues et d'épilogues. En CLOS, il est possible en effet d'ajouter des prologues et/ou des épilogues aux fonctions disponibles (cf supra) : ces ajouts sont utilisés pour produire des effets de bord, et, sauf exception, ne changent pas pour le reste le fonctionnement ou la valeur retournée par les fonctions en cause (Muller & al. 1989). Grâce à ce savoir à dessein restreint, et sans devoir pour autant connaître l'ensemble des choix faits pour l'implémentation, l'utilisateur dispose de "vues" orthogonales pour pister le comportement de la grammaire au cours de l'analyse. Dans ce contexte, "orthogonal" signifie qu'il est possible d'observer une facette particulière d'un élément du réseau sans pour autant ouvrir la "boîte noire" et sans être confronté à des détails implémentatoires non pertinents. L'accès donné à l'implémentation se situe à un niveau d'abstraction suffisamment élevé (Kickzalez & al 1991). Il s'oppose aux deux approches qui sont disponibles habituellement lorsque, contrairement à toute attente, l'analyseur ne produit pas de dérivations pour la phrase proposée ou produit des dérivations erronées. La première revient à ouvrir la "boite noire" que constitue l'analyseur : on a recours alors aux outils de traçage et de mise au point du langage d'implémentation. La seconde approche consiste à enrichir la boite à outils de fonctionnalités de traçage spécifique (Boguraev & al. 1988), (Bouchard & al. 1992a). Ces deux approches présentent pourtant de sérieux inconvénients. La première force le linguiste utilisateur à manipuler des structures de données sans rapport évident avec les entités définies dans la grammaire et le lexique, et qui de toute façon sortent de son domaine de compétence propre. La seconde l'oblige à maîtriser une nouvelle couche logicielle qui s'ajoute à l'analyseur proprement dit.

Donnons quelques exemples d'utilisation de ce type de pistage :

Lorsque l'utilisateur veut connaître l'utilisation effective des règles de la grammaire sur un corpus donné, le compilateur ajoute un prologue déclenché à la création des instances de la classe correspondant au premier élément d'une règle, et un autre à la création de celle correspondant à la construction du constituant (dans l'exemple choisi et présenté infra, à chaque création d'une instance de `DET5` et de `GN7`). Ces prologues incrémentent des compteurs, ce qui permet de savoir le nombre de fois où chaque règle a été déclenchée et le nombre de fois où elle a réussi. Les résultats fournissent des aperçus instructifs sur l'organisation de la grammaire et permettent de l'améliorer, en ordonnançant les règles par ordre de probabilité. Tracer une règle donnée se traduit par l'ajout de prologues et d'épilogues aux fonctions de combinaison des entités représentant les constituants déjà trouvés avec celles qui représentent les constituants dont la présence est en cours de vérification.

L'insertion de points d'arrêt dans de tels ajouts permet d'examiner en mode pas à pas le fonctionnement de la règle, et donne l'occasion d'explorer à loisir l'environnement d'exécution précisément quand et où c'est pertinent. En outre, il est souvent important dans des formalismes qui associent des contraintes arbitraires aux éléments des règles, comme c'est le cas pour les formalismes d'unification, de pouvoir pister non une règle ni même un ensemble de règles, mais des éléments au sein de différentes règles qui ont recours à un même attribut, de manière à observer la percolation de cet attribut au cours de la dérivation. Un pistage lié aux règles n'est pas approprié : il produit trop d'information. Dans notre modèle, il est aisé d'attacher des instructions de pistage aux seules entités pertinentes, et ceci à travers toute la grammaire.

Cette approche de la mise au point est efficace du point de vue de l'utilisateur, dans la mesure où elle repose sur une relation causale et immédiate entre la grammaire de départ et les entités présentes à l'exécution. (Carroll & al. 1991) indiquent d'ailleurs qu'un travail efficace n'est possible que si l'on peut revenir des règles fautives pendant l'exécution à l'endroit de la grammaire où elles sont définies. Par contraste, le traçage proposé par exemple par (Bouchard & al. 1992) permet de voir le graphe évoluer, pas de suivre une règle dans son fonctionnement, tandis que (Boguraev & al. 1988) font le lien entre la grammaire et ses résultats par le biais d'étiquettes attachées aux règles et l'ajout de ces étiquettes aux structures produites... En outre, la distinction est claire entre les traitements normaux et les comportements ajoutés en phase de mise au point.

Au total, par la constitution de points de vue, on peut voir dans une approche réflexive une façon de donner une meilleure prise sur des systèmes touffus. Pour reprendre les termes de (Sabah 1993) : *"[La méta-représentation] est /./ sélective (ce qui permet une réduction de la complexité du système. En outre, le contenu de la méta-représentation dépend du type de raisonnement qui va l'utiliser : elle est spécialisée (un système 'conscient' a donc la possibilité de disposer de plusieurs méta-modèles du même objet). En conséquence, cette représentation est partielle (ce n'est pas une simple image complète du système, mais une représentation des seuls éléments pertinents pour la tâche considérée)"*.





# 9.7. Réflexivité : modification des programmes

## 9.7.1. Hétéro-modifications

### 9.7.1.1. Savoir ce qui se passe, vers des traces pour linguistes

**Etape de mise au point de traceurs.**

Cette phase de mise au point du code de Olmes a permis l affinement de la présentation des résultats dans le but prédéfini de fournir des informations pertinentes pour les premiers utilisateurs de OLMES à savoir des linguistes. Il fallait mettre en place des mécanismes qui soient capables de tracer les points cruciaux de l analyse et de fournir des résultats intermédiaires et terminaux pertinents pour les utilisateurs. De plus, la mise au point progressive d un méta-niveau de l analyse a permis de garder au delà de l analyse des informations présentes au moment de l analyse et donc de pouvoir procéder à une rétro-lecture de ces informations à l issue de l analyse.

**Les traceurs.**

Les traceurs pour l analyse avec OLMES utilisent principalement la potentialité offerte par CLOS d ajouter ou de supprimer des méthodes.

1. Ajout de méthodes :

CLOS permet de réaliser l ajout d une méthode à une fonction générique en aval de la création de ces deux entités. Cet ajout se fait grâce à la fonction générique `add-method`. Cette fonction générique réalise les vérifications de congruence de lambda-list, d existence d une méthode utilisant la même forme de discrimination et de l unité d utilisation de cette méthode.

2. Retrait de méthode :

Les traceurs peuvent à certains moments produire des informations non-pertinentes : on doit pouvoir les activer et les désactiver en fonction de certains besoins particuliers. CLOS offre la possiblité d exécuter une extraction de méthode en utilisant la fonction générique `remove-method`. Cette action est rendue possible car CLOS dispose de plus d une fonction générique (`find-method`) capable de retrouver une méthode utilisant une certaine forme de discrimination sur ses arguments.

L ajout et le retrait de méthodes `before` et `after` s est révélé être un outil fondamental pour fournir des résultats au cours de l analyse. On rappelle que la combinaison des méthodes par défaut en CLOS (dite standart) accepte quatre catégories de méthodes, les méthodes primaires, les post-méthodes (`:after`), les pré-méthodes (`:before`) et les méthodes englobantes (`:around`). Les outils de traçage manipulent les post et pré méthodes qui permettent des effets de bord. Les méthodes définies pour l'analyse automatique peuvent être étendues en y adjoignant des pré-méthodes ou des post-méthodes dédiées à fournir des résultats ou des informations sur les objets manipulés par ces méthodes tout au long de leur activité. Les traceurs mis en place ont été développés pour fournir aussi bien des résultats terminaux que des résultats intermédiaires d une analyse. Un exemple du code défini pour les traceurs est donné en annexe, accompagné des différents résultats produits correspondants aux différentes étapes de développement de ces outils de traçage (-> annexe partie 3 chapitre 9 :

une analyse complète est donnée précédée du code complet de l  interface des outils associés à OLMES).

## Un méta-niveau dans OLMES.

La première phase de développement d'un méta-niveau pour OLMES (DINOSOR (Fleury & al. 1992), (GASPAR (Fleury 1992)) a permis d'introduire la nécessité de disposer d'informations sur la grammaire compilée à l'issue de l'analyse. Ce travail a donc consisté à introduire une méta-grammaire pour l'analyse. Il s'agissait de "garder" un point d'entrée sur la grammaire à l'issue de l'analyse. Cette phase de développement d'un méta-niveau a permis de mettre en lumière l'importance de travailler à un méta-niveau pour la mise au point des modifications opérées sur le code initial : c'est grâce au développement d'un méta-niveau intermédiaire (la grammaire normalisée) que la mise au point de la méta-grammaire a été facilitée. Le schéma ci-contre illustre cette première phase de développement d'un méta-niveau pour l'analyse :

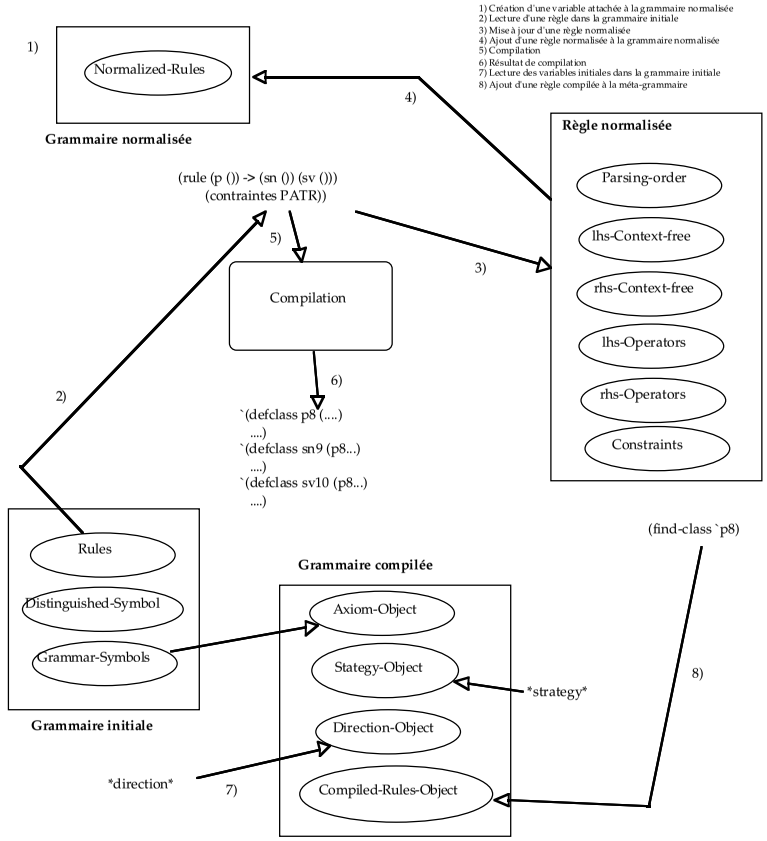



Figure 9.9 : DINOSOR 92 : un premier méta-niveau pour OLMES.





Cette phase de développement a permis de mettre en évidence les défauts de cette présentation (<- le problème mentionné de dépendance hiérarchique entre partie gauche d'une règle et partie droite). La solution retenue dans (Habert & Fleury 1993) maintient la définition d'un méta-niveau pour la grammaire comme suit :

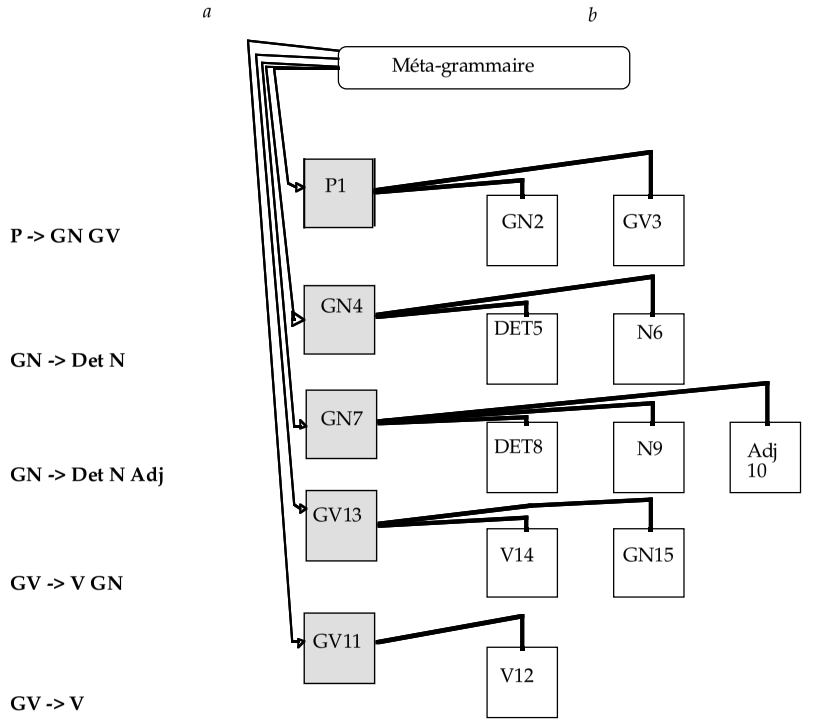
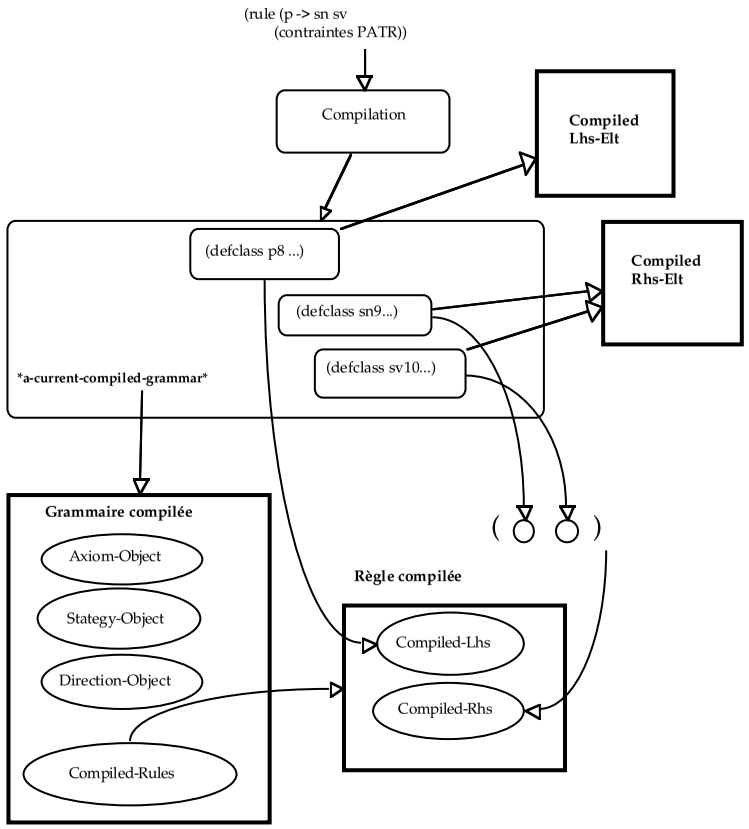



Figure 9.10 : OLMES 93 : une méta-grammaire pour OLMES.

Cette version a permis de prendre en compte de manière fondamentale la nécessité de mettre en place un méta-niveau pour l'analyse (Habert & Fleury 1993). Les défauts déjà mentionnés sur cette version ont été repris et une version ultérieure a amplifié cette notion de méta-niveau. Le point principal a été de ne plus considérer la partie gauche d'un règle comme la super-classe des éléments de la partie droite d'une règle.




**Compilation avec OLMES**

Figure 9.11 : OLMES 94 : un méta-niveau pour l'analyse avec OLMES.

Le fait d'éclater la règle en autant d'objets qu'il y a d'éléments dans cette règle a permis notamment dans le travail de traçage de pouvoir isoler la création de chacun de ces éléments pour par exemple pouvoir compter le nombre d'instanciation de ces éléments. Cette phase de mise au point d outils pour une lecture et une compréhension active de l analyse a aussi pris en compte la nécessité de disposer d une interface graphique des résultats fournis par les outils développés.

### 9.7.1.2. De la méta-connaissance à la réflexivité

Montrons avec OLMES comment passer de la production de méta-connaissances à une réelle réflexivité, c'est-à-dire à la modification dynamique du comportement de l'analyseur en fonction de ces méta-connaissances. L'objectif d'une approche réflexive est en effet que chaque entité active non seulement possède les moyens d'observer son comportement à l'exécution, mais qu'elle utilise ces "notes" pour modifier, si nécessaire, ce comportement. CLOS donne des outils pour une telle approche réflexive. Il dispose en effet d'un Méta-Protocole (Kickzales & al.





1991) : chaque entité de la couche objet-classe, méthode ou champ - est une instance d'une méta-classe, donc, en d'autres termes, un objet. Parce que chaque classe est également un objet, les classes représentant la grammaire de départ possèdent une mémoire propre, et peuvent par conséquent noter leurs observations sur l'analyse. Le nombre de règles déclenchées et de succès pour une règle peut être mémorisé "au sein de chaque règle". La figure qui suit montre les changements qui en résultent pour les classes `GN7` et `Det8` (<- exemple présenté supra) et la mémorisation des essais et des succès dans la version de OLMES présentée dans (Habert & Fleury 1993) :

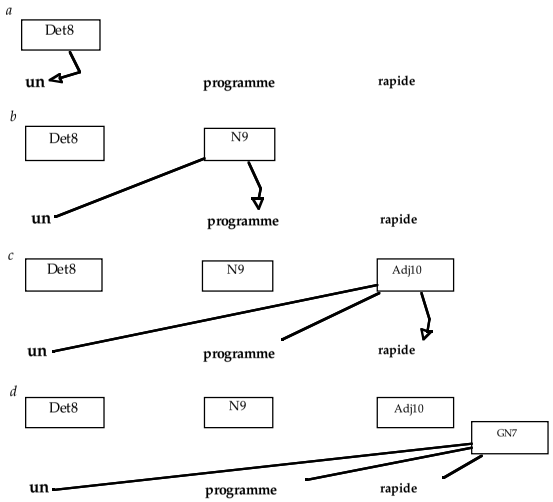



Figure 9.12 : Analyse par les instances dans un réseau de classe.

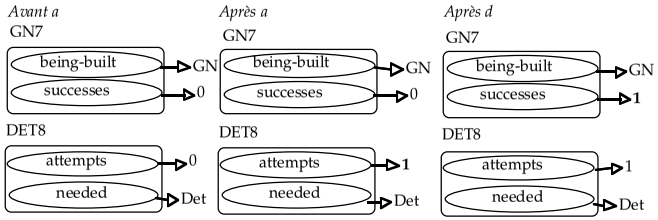



Figure 9.13 : Mémorisation des essais et des succès dans les classes d'une règle.

De nouvelles méta-classes peuvent alors servir à affiner le comportement de ces unités. C'est cette piste prometteuse qui peut permettre la réorganisation du réseau par lui-même en fonction des probabilités attachées à chaque règle après le passage des règles sur un corpus d'essai et l'observation des déclenchements et réussites des règles. Les transformations apportées au modèle de (Yonezawa & Ohsawa 1990), le lien entre la grammaire de départ et les entités qui la représentent lors de l'analyse, permettent précisément cette action en retour de l'analyse sur la grammaire.

## 9.7.2. Auto-modification

### 9.7.2.1. `Gaspar` : Un analyseur à mots non entièrement définis

## Comment définir des "démons" capables de redéfinir les comportements généraux à factoriser

Pour reprendre ce que nous disions dans les chapitres précédents (<- partie 2 chapitres 4, 6) sur la (re)construction d'une représentation sémantique que l'on peut attacher à `moulin`, il s'agit ici d'évaluer comment il serait possible d'étendre cette représentation sémantique aux noms d'instruments : comment mettre en oeuvre une catégorisation plus générale tout en préservant une évolution potentielle de cette représentation initiale ? La représentation sémantique proposée supra tend à identifier (à isoler) les comportements sémantiques que l'on peut attacher à `moulin` et qui permettent de construire des (débuts d') interprétations possibles qu'il reste possible d'affiner. Si ces comportements traduisent des opérations sémantiques plus générales qu'il serait possible de factoriser pour une classe de noms particuliers (en particulier pour les noms d'instrument), il conviendrait de restructurer l'organisation des représentations sémantiques attachées aux entités représentées pour disposer d'une représentation ouverte et cohérente du domaine à représenter. La figure ci-dessous rappelle l'opération de recatégorisation présentée.

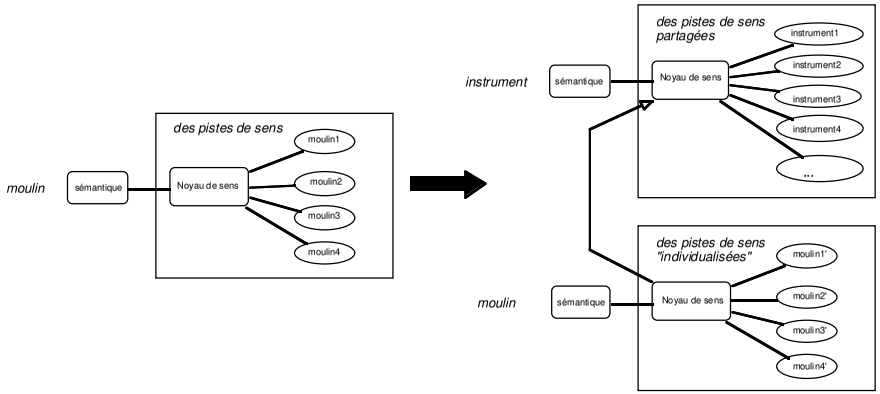



Figure 9.14 : `Moulin` et noms d'instrument, des comportements sémantiques partagés.

Si une approche prototypique se révèle adéquate pour rendre compte d'un micro-phénomène particulier, il convient de ne pas perdre de vue la place de ce micro-phénomène dans un système plus complexe, ou tout au moins de dégager de l'étude particulière de celui-ci, les éléments qui le dépassent.

## Du méta-regard sur l'analyse à l'automodification des représentations

Le problème à résoudre peut se poser de la manière suivante :

Tel type de comportement sémantique attaché à `moulin` se révèle être un comportement sémantique plus général pour une classe de noms plus large. Est-il possible de réorganiser cette part de savoir plus général ? Comment effectuer cette réorganisation du savoir ? Quel est l'impact d'une telle réorganisation sur les savoirs représentés ? Il convient pour cela de disposer de savoirs capables de détecter que tel comportement attaché à `moulin` se retrouve sous une forme assez proche dans la représentation sémantique attachée à un autre nom d'instrument. Puis d'étendre ce rapprochement de comportements partagés à un ensemble d'entités représentées. La difficulté ne s'arrête évidemment pas là, il s'agit ensuite de définir





les modalités d'une réorganisation des savoirs partagés. La question de savoir si un tel aménagement est souhaitable n'est d'ailleurs pas à écarter : ne risque-t-on pas de définir des regroupements aléatoires dont la pertinence n'est pas justifiée? Il peut en effet y avoir des recoupements entre les comportements sémantiques de plusieurs classes de noms, il faut pouvoir y retrouver une cohérence à moins d'y perdre toute volonté de classer les choses "raisonnablement".

## Un méta-niveau pour l'analyse avec `Gaspar`

Utilisation de la réflexion de structure en Self

L'approche présentée supra (<- partie 2 chapitres 7, 8) réalise une génération dynamique des représentations prototypiques des savoirs linguistiques à partir de données recueillies sur un corpus. Celle-ci met en avant la nécessité de disposer d'un méta-niveau de représentation pour la gestion des objets créés ou à modifier. On doit pouvoir ajuster :

Les représentations des objets associés à des unités de langue (ajout ou retrait d'informations).
Les processus qui agissent sur ces unités.
Les liens qui existent entre ces unités (modifier, ajouter des liens d'héritage ou autre sur ces unités).

Etant donné la variation potentielle des structures représentées, on doit pouvoir connaître la structure d'un objet donné à un moment particulier de son activité. Si l'on veut modifier cette structure, on doit disposer d'un méta-regard sur la composition de cet objet : les attributs qui le composent, les liens d'héritage qui lui sont associés... L'interface graphique de Self-4.0 permet à tout moment d'agir sur la représentation d'un objet donné : ajouts ou suppressions d'information, modifications de valeurs ou des comportements... On peut donc agir directement et interactivement sur les objets représentés. Pour construire un programme de TALN capable d'agir dynamiquement sur ces mêmes objets, on doit disposer d'outils qui produisent les mêmes résultats mais qui agissent de manière transparente pour permettre une modification dynamique des représentations construites. On présente ci-dessous une suite d'opérations qui permettent d'agir sur les objets représentés sans passer par l'interface graphique de Self (-> annexe partie 3 chapitre 9 : trace des (méta-)opérations via `Gaspar`). On choisit volontairement cette solution pour donner à penser ce que le programme construit réalise lui de manière dynamique.

Utilisations de méta-savoirs avec Gaspar

1. Un méta-regard sur les objets

Dans les différentes phases de travail présentées (<- partie 2 chapitres 6, 7 et 8), les structures de représentation construites pour représenter les unités de langue peuvent évoluer à tout moment de leur existence : on peut ajuster, remodeler la représentation des objets en y ajoutant, en y supprimant ou en y modifiant les informations associées. Il est donc impossible de prévoir toutes les facettes d'un objet donné : on ne prévoir toutes les évolutions que peuvent subir les représentations des unités de langue. Par contre, il reste possible à tout moment de connaître la structure de cet objet. Le dispositif permet à tout moment, via les (méta-)outils construits, d'interroger les objets pour connaître leur structure. On donne en annexe (-> annexe partie 3 chapitre 9) une présentation du prototype lexical associé au mot `lesion` en utilisant les outils disponibles avec `Gaspar`. En utilisant l'interface graphique de Self, on dispose aussi des outils définis par cette interface pour obtenir une présentation interactive de la structure interne d'un objet. Il reste cependant important de pouvoir travailler avec ces savoirs structurels dans un programme en cours d'exécution (de manière transparente pour l'utilisateur) et donc de pouvoir récupérer ces informations de manière automatique si on doit agir sur celles-ci. On a vu aussi

que l'on pouvait pour un ensemble de prototypes donnés construire un pôle destiné à porter les comportements communs à ces prototypes. La construction de ce pôle s'accompagne de la mise en place de nouveaux liens entre les prototypes concernés et le pôle construit. Si l'on s'intéresse à l'un des prototypes de la famille, et si l'on veut connaître les attributs que celui-ci délègue à d'autres objets, il faut pouvoir détecter parmi ses attributs ceux qui révèlent un lien de délégation vers un autre objet pour examiner les attributs délégués. Pour réaliser ce genre de requête sur un prototype Self, on utilise des outils disponibles avec Self qui agissent à un méta-niveau et qui permettent de décrire et de modifier la structure d'un objet donné.

2. Sélection des savoirs à manipuler

L'environnement mis en place avec Self facilite les manipulations concrètes et directes sur les entités représentées. Un utilisateur peut "communiquer" avec les agents du programme directement pour les modifier éventuellement sans avoir à connaitre précisément les mécanismes qui permettent ces modifications. Un programme Self est constitué d'entités "vivantes" et non d'entités inertes et immuables. Pour changer un objet, il est inutile de recompiler, il suffit de modifier l'objet suivant les besoins. Notre approche expérimentale s'accorde bien avec la possibilité dans l'environnement programmatique choisi de modifier interactivement les représentations informatiques créées. L'interface de Self permet une visualisation des objets, et l'utilisateur peut "ausculter" les contenus des différents attributs ou des méthodes associés aux objets. Il peut aussi agir sur ces différentes facettes d'un objet. Nous avons déjà indiqué que certaines phases de travail manuel peuvent être réalisées pour affiner les résultats construits quand une action directe sur les représentations produites est nécessaire. Il n'en reste pas moins qu'une part importante de notre programme gère automatiquement les savoirs rencontrés. Il doit donc pouvoir sélectionner les savoirs à traiter s'il doit agir sur eux. Pour réaliser ce type de sélection, le dispositif utilise les méta-outils mis en place par Self pour interroger la structure d'un objet donné. Prenons un exemple :

**Sélectionner le destinataire d'un message**

Si un attribut de même nom est disponible dans un prototype et dans un ou plusieurs de ses parents, on peut sélectionner le récepteur auquel on veut adresser une requête sur cet attribut. Dans notre travail de classement on a montré précédemment que les processus définis mettent en place un réseau complexe de liens de délégation entre les prototypes de mots. Il est donc nécessaire de pouvoir retrouver et utiliser les liens de délégation définis pour adresser les messages pertinents aux objets adéquats. De plus, ces liens de délégation ne sont pas figés de manière définitive. Il est donc important de pouvoir sélectionner le destinataire d'un message si un objet donné délègue des comportements à différents parents : un mot peut déléguer des comportements (des arbres) à plusieurs parents, il convient donc de pouvoir retrouver les différents chemins possibles pour récupérer le comportement adéquat (<- partie 2 chapitre 7). Il convient pour cela de pouvoir :

1. Connaître pour un prototype les liens d'héritage qui lui sont associés.
2. Récupérer tous ses parents.
3. Retenir le destinataire visé.

Self permet d'interroger la structure de tous les objets disponibles dans le système : on peut ainsi déterminer si un attribut donné dans un objet est un attribut parental, (i.e. cet attribut induit un lien de délégation entre l'objet visé et un autre objet). En outre, on dispose avec Self de primitives qui permettent de "forcer" l'envoi d'un message à un parent donné. On peut donc adresser le message souhaité au parent visé en utilisant par exemple la syntaxe suivante :

```
'message' sendTo: Objet DelegatingTo: parentChoisi.
```

3. Ajouter de la délégation





Self ne dispose pas de classes mais permet de construire des liens d'héritage dynamiquement entre les objets visés. Le classement construit sur les prototypes lexicaux présenté supra (<- partie 2 chapitre 8) construit des hiérarchies locales entre un sous-ensemble de prototypes et un pôle de comportements partagés par ces prototypes. Cette opération réalise en fait une construction dynamique de liens hiérarchiques entre objets. Ajouter un lien de délégation à un prototype est une opération qui consiste à établir un lien entre deux objets, on indique au prototype fils qu'il délègue des informations (attributs ou méthodes) à un prototype père. Ceci n'est qu'une application naturelle de Self qui permet à tout moment d'ajuster la représentation des objets disponibles. Le dispositif réalise automatiquement cette mise en place de délégation entre prototypes (-> annexe partie 2 chapitres 8, 9); les processus mis en place agissent à un méta-niveau pour évaluer les modifications effectuées par ces opérations dynamiques de changement d'état d'un objet.

4. Supprimer la délégation

Si on peut ajouter de la délégation, l'opération inverse est elle-aussi possible. Il convient simplement de connaître le lien à supprimer (i.e. le(s) prototypes fils et le(s) prototype(s) père concerné(s)). On dispose avec Self de mécanismes qui permettent d'interroger les attributs d'un objet, on peut en particulier savoir si un attribut donné est un attribut qui délègue des comportements. Le dispositif utilise donc ces mécanismes pour réaliser des opérations de suppression de lien de délégation entre objets. Dans le système `Gaspar`, on dispose par exemple de processus capables de supprimer automatiquement les liens de délégation de même nature (i.e. les mêmes comportements étant portés par deux ou plusieurs pôles différents).

5. Un Méta-traçage

Le travail sur des processus réflexifs pour un traitement automatique passe impérativement par une phase de développement de processus de traçage des activités d'un programme de traitement automatique. Dans les différentes phases de travail présentées, les structures de représentations construites peuvent subir des manipulations diverses : elles peuvent être structurellement modifiées ou affinées. On conçoit donc qu'il soit utile, voire indispensable, de pouvoir suivre ces évolutions potentielles. A cet effet, les processus réflexifs disponibles avec le cadre de représentation choisi ont permis de développer des outils de suivi ou de mise au point de l'analyse. Ces outils présentent, de manière raisonnée, les différents états pertinents des traitements réalisés et des résultats construits à chaque étape. Les différents outils créés au cours de ce travail ont d'abord été conçus comme des modules indépendants capables de gérer eux-mêmes le traçage de leurs activités. Le développement d'un méta-niveau de travail a conduit à modifier cet isolement initial. En fait, chaque module reste indépendant en ce qui concerne son éventuel traçage mais utilise un module de traçage qui se situe à un méta niveau.

```
GasparMetaLook queryForActiveTrace ActiveOutputTrace <- true/false
```

6. Un Méta-regard pour maintenir la cohérence des objets construits

On a vu précedemment que pour contrôler la génération des prototypes, des mécanismes de vérification de la structure interne des objets construits ont été définis : un prototype sera "cohérent" s'il est issu des différentes phases de construction des prototypes de mots ou des prototypes d'arbres. Ce qui indiquera qu'il possède tous les facettes nécessaires pour être considéré comme valide dans l'environnement de `Gaspar`. Pour ne pas multiplier inconsidérement les prototypes incohérents, on dispose d'un mécanisme qui permet de "détruire" tous ces prototypes incomplètement construits ou tout au moins de les "poubelliser". Ces opérations sont définies là aussi en utilisant les outils de méta-représentation disponibles avec Self qui permettent de supprimer tous les attributs à un objet donné.

## Classification automatique et processus réflexifs

Dans l'état actuel du classement effectué, une intervention manuelle est indispensable pour interpréter les résultats construits. Les limites de l'automatisation marquent le champ de travail qu'il reste à effectuer manuellement pour mener à bien le classement des unités lexicales. Le classement à mettre en place est fortement dépendant des informations qui ont permis de définir ce classement. Il reste donc indispensable de s'interroger sur la nature des informations linguistiques qui sont manipulées pour ce classement. Les informations recueillies actuellement en amont doivent être renforcées et affinées. Notre travail doit d'une part préciser quelle est la nature des informations à traiter et suivant les informations manipulées, quel type de classement on obtient, et quel type de classement veut-on ou peut-on obtenir sur la base de ces informations. Il doit aussi mettre en place des processus très fins de classement capables de mettre de l'ordre dans les structures construites. Plus précisément, puisque le classement présenté supra met en place des liens de délégation entre des prototypes et des pôles de comportements partagés, une automatisation plus fine de ce classement doit passer par la mise en place d'un vaste et complexe réseau des liens de délégation. Il reste donc à évaluer les interactions et concurrences qu'un tel réseau induit. Il ne suffit pas d'ajouter des liens de délégation dans tous les sens pour produire le classement escompté. Il faut pouvoir évaluer les implications de la mise en place effective de plusieurs liens de délégation sur un même prototype donné : il faut par exemple s'assurer que l'on ne produit pas arbitrairement de l'ambiguïté dans les liens mis en place. La question de l'attestation ou de la non-attestation, par le système lui-même, de ses propres représentations est un point crucial pour l'ajustement conceptuel. On conçoit donc qu'il soit nécessaire de disposer de processus qui agissent à un méta-niveau et qui soient chargés de ce contrôle (Bachimont 1992). Les outils présentés précédemment ne constituent qu'une première phase de développement de tels outils de contrôle.

## Mettre en place la génération de nouveaux savoirs

Notre travail vise à permettre aux objets créés d'apprendre par ajustements successifs de nouvelles connaissances. Dans les chapitres précédents, les prototypes lexicaux apprennent dynamiquement les savoirs syntactico-sémantiques qu'il est possible de leur associer. Cet apprentissage est réalisé sous la forme d'une relation d'apprentissage d'un maître à un élève. Le maître en l'occurrence se matérialise tout d'abord sous la forme des informations initiales recueillies sur un corpus donné. Le classement produit peut aussi conduire l'utilisateur à intervenir directement sur les résultats construits pour affiner ces résultats. Cet apprentissage via l'acquisition de connaissances proposées par un maître ne suffit pas (Pitrat 1995). Un programme de TALN doit pouvoir construire de nouvelles connaissances à partir des connaissances connues. Nous avons déjà indiqué les difficultés fondamentales que posent cette dernière assertion. Nous nous limiterons ici à présenter quelques exemples simples qui illustrent cette génération de nouveaux savoirs à partir de savoirs existants.

Affiner les processus autodescriptifs des objets

Les objets construits doivent évaluer leurs fonctionnements et leurs comportements par rapport aux autres objets. Il convient donc de mettre en place des processus fins pour permettre aux objets construits de donner une bonne description d'eux-mêmes afin qu'ils puissent comparer cette description à celles des autres objets. Si deux objets (représentant des mots ou des arbres) doivent "dialoguer" pour évaluer leurs descriptions respectives et celles de leurs comportements, il convient que ces objets puissent activer des processus capables de produire ces descriptions.De plus, dans la mesure où les objets construits sont susceptibles d'évoluer, il convient que les processus d'autodescription des objets construits puissent être affinés dynamiquement pour tenir compte de ces évolutions potentielles. Self permet la mise en place





de mécanismes qui réalisent cette description des attributs d'un objet donné à tout moment de son existence dans le système. La mise en oeuvre de telles opérations sur tous les objets du système construit reste à faire. De même il convient de poursuivre la mise en place de processus capables d'interroger plus précisément les attributs (de comportements) attachés aux mots et à ceux des arbres associés si l'on veut donner du sens aux savoirs ainsi représentés.

Création de pôles de comportements

La phase de travail qui consiste à repérer les savoirs communs puis à définir et à créer les pôles de comportements partagés peut être assimilée à une première étape dans la construction de nouveaux savoirs. Celle-ci reste malgré tout à affiner et à développer. Ces regroupements de savoirs partagés par un ensemble de prototypes lexicaux ou par un ensemble de prototypes d'arbre ne sont en effet pas identifiés ou nommés. L'opération qui consiste à évaluer la ressemblance comportementale entre les mots ne qualifie pas les ressemblances constatées.

Création automatique de prototypes de mot au cours de l'analyse

On a déjà indiqué que si au cours de l'analyse le parser rencontre un mot nouveau, il construit automatiquement des prototypes lexicaux (pour chaque catégorie déjà recensée) pour représenter ce nouveau mot. De plus ne sachant pas si ce nouveau mot est forcément un représentant d'une des catégories grammaticales déjà reconnues, le parser construit aussi un prototype lexical de catégorie inconnue pour représenter ce mot nouveau. Ce dernier type de prototype pourra entrer dans toutes les compositions syntaxiques sollicitées par d'autres prototypes d'arbres. Cette phase de génération automatique de nouveaux prototypes de mot au cours de l'analyse utilise les représentations déjà construites pour en produire dynamiquement de nouvelles : le parser prédit différentes représentations prototypiques pour ce mot nouveau en tenant compte des représentations prototypiques de mot qu'il connaît déjà. la mise en place de ce type de processus reste encore à affiner. Il faut en particulier pouvoir enregistrer si les représentations construites par défaut ont permis de donner une solution satisfaisante ou non et dans tous les cas garder trace de ces résultats.

Des règles pour la génération des nouveaux objets

La phase de génération de prototypes mise en place supra (<- partie 2 chapitre 8) reconstruit systématiquement et de toutes pièces un nouveau prototype s'il n'existe aucun représentant de sa famille catégorielle sans vérifier plus avant si les prototypes existants sont capables de fournir des informations pour la génération de ce nouvel élément. Il serait en fait possible de modifier cette génération de prototype en utilisant de manière plus fine les savoirs déjà construits. Ce qui permettrait de plus d'améliorer les performances du système construit en limitant par exemple le nombre d'écritures sur disque d'informations générées pour réaliser la génération d'un nouvel objet. En effet, en examinant plus précisément la structure des objets déjà construits, il ne serait plus nécessaire de reconstruire de toutes pièces un nouvel objet; en s'inspirant de la structure des objets construits, il suffirait d'ajuster ces structures pour produire de nouveaux objets en utilisant par exemple les règles de génération suivantes :

Méta-Règle n 1 :

Si un prototype d'arbre a x composants
S'il n'existe pas de moule d'arbre à x composants
Alors créer ce moule en s'appuyant sur ce prototype

Méta-Règle n 2 :

Si un Objet Ae (Arbre élémentaire) a x composants.
S'il n'existe pas de moule d'arbre à x composants.
Alors créer ce moule en s'appuyant sur Ae.

Ce travail reste à définir et à mettre en place.

Créer un dictionnaire d'erreurs

Nous avons déjà indiqué qu'une part de notre travail d'interprétation des structures construites passe par une phase de travail manuel qui va consister à agir directement sur les objets construits pour affiner leurs représentations. Afin de garder trace de ces interventions manuelles (faite au méta-niveau que constitue le méta-regard de l'utilisateur), toute intervention manuelle ou provenant du programme sur les représentations sera enregistrée dans un dictionnaire. Le méta-niveau permet dans ce cas d'évaluer les représentations disponibles et de les corriger si nécessaire, tout en gardant trace de ces interventions sur celles-ci. **Cette trace des erreurs assimile l'apprentissage mis en place à un processus qui enregistre le positif et le négatif**.





# 9.9. Bilan des activités gaspariennes et méta-perspectives

## 9.9.1. Activités gaspariennes

On présente ci-dessous un schéma global des différentes activités définies via `Gaspar`.

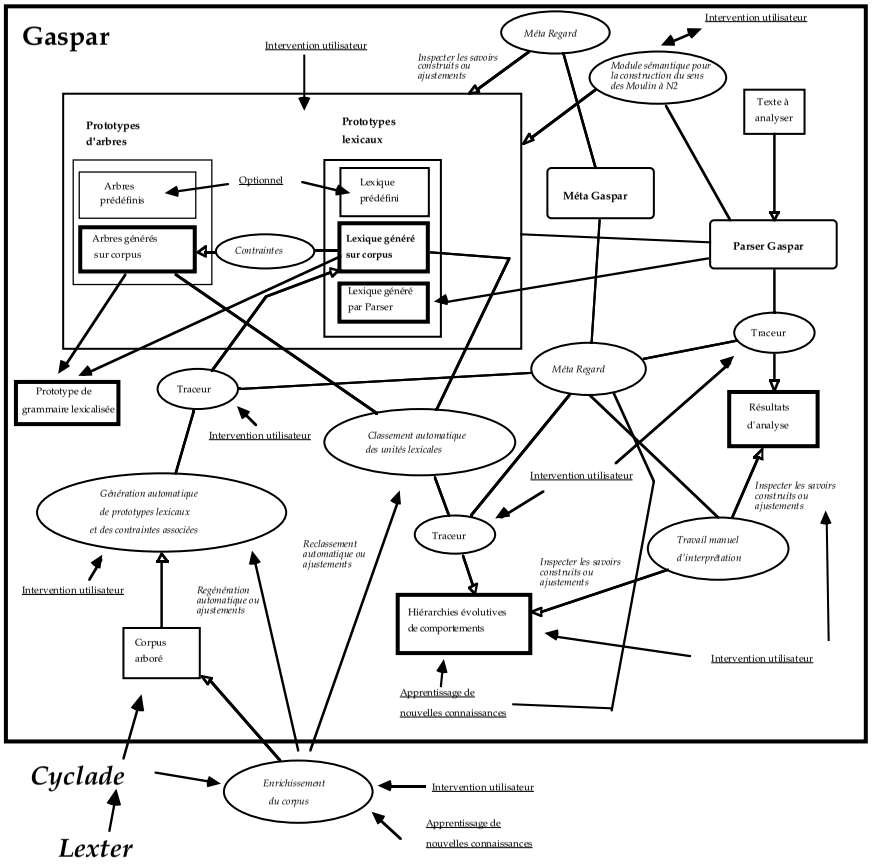



Figure 9.15 : Schéma global des activités gaspariennes.

Ce schéma ne rend évidemment pas bien compte du fait que de cette hétérarchie enchevêtrée naît de la représentation. Il met simplement en avant la complexité des mécanismes mis en place et de leurs connexions. La myriade de prototypes qui résultent du travail de génération

automatique de prototypes de mots et d'arbres est en effet difficilement réductible à une présentation schématique.

Le dispositif `Gaspar` (sous Self-4.0) est composé de plusieurs modules (<- partie 2 chapitre 8 : catégorisation des attributs Gaspar) :

Un module de calcul sémantique des séquences du type `moulin à N2` avec intervention éventuelle de l'utilisateur pour affiner les résultats (<- partie 1 chapitre 1, 4 <- partie 2 chapitres 5, 6, 7).

Un module pour la génération automatique de prototypes de mot et de prototypes d'arbre à partir de savoirs extraits sur corpus (<- partie 2 chapitre 8).

Un module de classement des prototypes construits ou disponibles (<- partie 2 chapitre 8).

Un module méta (<- paragraphes précédents).

Ce dernier module regroupe différents processus qui sont utilisés par les différents modules mis en place. Il permet d'activer des méta-regards sur les savoirs représentés : il intègre notamment les processus de traçage que l'utilisateur peut activer sur toutes les opérations disponibles dans chaque module. Ces processus de traçage décrivent les différentes étapes des opérations déclenchées. L'utilisateur a aussi la possibilité d'intégrer automatiquement une présentation des résultats construits dans l'interface graphique de Self. Il peut ainsi avoir une présentation complète des représentations manipulées ou construites. **Ces objets sont alors manipulables et modifiables à volonté et de manière dynamique**. Ce méta-module peut aussi activer des actions de modification dynamique des représentations construites : ces processus peuvent agir de manière transparente pour l'utilisateur.

Un module d'analyse syntaxique (<- partie 2 chapitres 5, 6).

La parseur permet plusieurs alternatives lors d'une analyse :

Il peut travailler (à la demande) avec des contraintes prédéfinies (<- partie 2 chapitre 5, 6) ou avec les contraintes générées automatiquement (<- partie 2 chapitre 8).

Il peut utiliser les différents opérateurs mis en place pour la construction automatique d'arbres par substitution ou adjonction (<- partie 2 chapitre 8).

Il peut activer (à la demande) le module particulier de calcul sémantique sur les séquences du type `moulin à N2` avec intervention éventuelle de l'utilisateur pour affiner les résultats.

Le parseur peut enfin utiliser (à la demande) les représentations déjà construites pour en produire dynamiquement de nouvelles (<- partie 2 chapitre 8 et paragraphes précédents).

Un unifieur est en cours d'implémentation. Il utilisera les contraintes associées aux mots et aux arbres; ces contraintes étant soit issues du travail d'extraction de savoirs à partir de corpus soit mises à jour dynamiquement sur les représentations construites (par l'utilisateur par exemple). Cette implémentation doit permettre, en particulier, de restreindre le nombre de prototypes construits mais non pertinents à l'issue d'une analyse (notamment quand la phrase à analyser contient des mots inconnus (-> annexe partie 3 chapitre 9 : analyse avec `Gaspar`)). La figure qui suit présente plus précisément les activités gasparienes mises en place pour l'analyse :





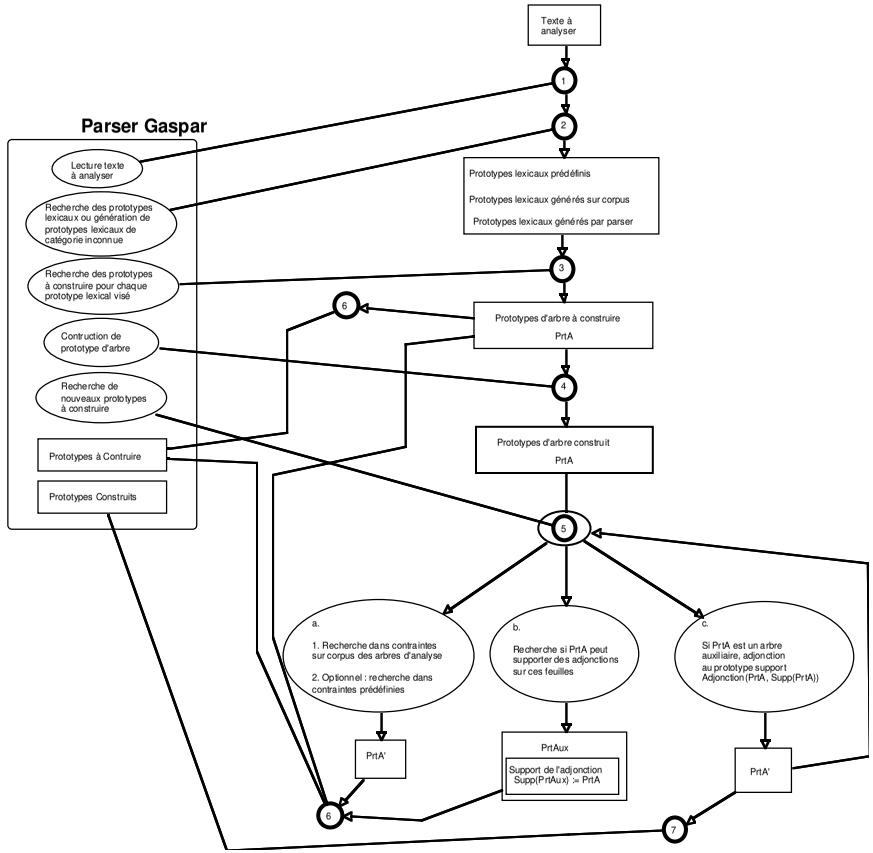



Figure 9.16 : Activités gasparienes pour l'analyse.

## 9.9.2. Perspectives

Les différentes phases de travail présentées ont mis en avant des points cruciaux pour la mise en place de processus de représentation des faits de langue.

La matière à représenter est "vivante" : elle évolue en permanence, dans le temps et dans ces mises en action. Les outils de représentation doivent donc ne pas figer les savoirs représentés : les structures de représentation construites doivent restées ouvertes et évolutives. La mise en place de telles structures de représentation évolutives pour un traitement automatique du langage naturel pose le problème fondamental du contrôle permanent à définir pour maintenir une cohérence dans le domaine représenté. Et on perçoit aisément les difficultés auxquelles on est confronté pour réaliser un dispositif de traitement automatique : la langue articule un grand nombre de connaissances et "se joue" de leurs représentations.

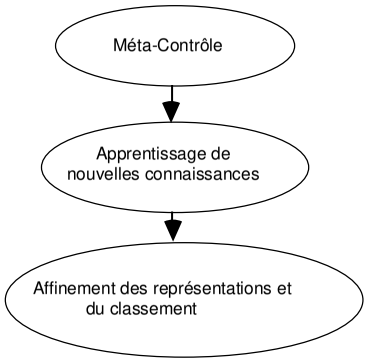



Figure 9.17 : Méta-perspective.

Notre travail n'est qu'une illustration de ces problèmes. En restreignant la couverture linguistique, on a pu mettre en avant l'immensité de la tâche à accomplir. Ce travail doit être poursuivi.

En particulier, le développement d'outils pour la représentation et pour le classement dynamiques des savoirs linguistiques doit être étendu. Ce travail est contraint par une double nécessité. D'une part, il est nécessaire de mettre en place des outils de contrôle capables à tout moment d'évaluer les différents types de savoirs manipulés et leurs interrelations existantes ou à venir. D'autre part, la réalisation d'un classement automatique général des unités lexicales semble hors d'atteinte tant qu'on ne pourra pas construire de nouvelles connaissances capables d'enrichir les savoirs déjà établis : en sachant de plus que ce classement à atteindre ne sera jamais un résultat définitif. Il reste malgré tout possible d'affiner les processus de classement mis en place et de fait d'affiner le classement visé (en connaissant ses limites) tout en continuant à travailler sur les métaconnaissances à l'oeuvre dans les faits de langue utilisés.