Actes de langage et jeux de dialogue
Rapport de stage - DEA RCFR

Christophe Delord

delord@enseeiht.fr
Laboratoire G.R.A.A.L.
(Groupe Raisonnement, Action et Actes de Langage)
Responsable de stage : Fabrice Evrard
ENSEEIHT-IRIT
2, Rue Charles Camichel - BP7122 - 31071 TOULOUSE Cedex 7

1^{er} septembre 1998

Résumé

Nous présentons ici un modèle pour le dialogue et les agents du dialogue. Le modèle est basé sur une structure particulière du dialogue : les jeux de dialogue. Les agents seront modélisés par leurs états mentaux et leurs capacités cognitives. Ce modèle a été implémenté en Prolog et plusieurs extensions sont envisageables.

Je tiens tout d'abord à remercier mon responsable de stage Fabrice Evrard pour son accueil au sein du laboratoire du G.R.A.A.L. à l'ENSEEIHT.

Je remercie aussi Nicolas Maudet pour ses relectures et ses commentaires d'une valeur inestimable.

Merci enfin à Giacomo Mazzotti qui fut pendant deux ans un binôme très sympathique et à Christophe Burgalat, Yves Marcet et Yannick Yrle pour leur compagnie et leur correspondance électronique.

Table des matières

Première partie

Présentation d'un modèle pour le dialogue

Chapitre 1

Introduction

1.1 Introduction

L'étude du dialogue est intéressante en informatique, non seulement pour la compréhension de son fonctionnement, mais aussi pour rendre plus intuitive une interface homme-machine par exemple. Le dialogue est plus qu'une simple succession d'actes de langage. En effet, le dialogue est une structure complexe qui nécessite des moyens d'analyse similaires à un système de règles de production pour son interprétation et des techniques inspirées de la planification pour sa génération. Notre but est donc de présenter un modèle de dialogue et des agents du dialogue en utilisant des techniques de l'intelligence artificielle.

A l'origine, nous nous sommes inspirés des travaux de (Beun, 1996) pour l'écriture des règles et la structuration du processus de génération d'actes de langage. Ces travaux ont été étendus pour hiérarchiser le dialogue en utilisant la structure de jeux de dialogue proposée par (Levin and Moore, 1980), (Mann, 1988).

Nous voyons donc deux éléments essentiels dans le dialogue : les agents et les jeux. Les agents du dialogue seront représentés par leurs états mentaux et leurs capacités cognitives (système inférentiel, axiomatique). Les jeux de dialogue permettront de représenter les comportements des agents en fonction de leur contexte cognitif et de l'état courant du dialogue. Nous présenterons finalement quelques extensions au modèle actuel. On considérera deux types de jeux : les jeux intentionnels, fortement dépendants du contexte cognitif et les jeux conventionnels, basés sur des schémas conventionnels du dialogue. Les jeux permettront de modéliser non seulement ces deux comportements, mais aussi des comportements intermédiaires. On montrera aussi l'utilité des logiques non classiques (probabilistes, non-monotones, ...), de la planification dans le dialogue.

L'implémentation de ce système a été réalisée en SWI-Prolog (version 2.9.10).

Dans un premier temps, nous présenterons la structure globale de notre modèle avant de détailler les outils utilisés et les évolutions envisagées.

1.2 Etat de l'art

(Wittgenstein, 1957) proposait de définir la notion de jeux de langage comme un processus à base de règles. Plus récemment, des travaux basés sur la théorie des jeux et sur les travaux de (Wittgenstein, 1957) ont conduit à la notion de jeux de dialogue. Selon (Burton and Brna, 1996), deux types d'approches permettent de définir la notion de jeux de dialogue :

- (Levin and Moore, 1980) et (Mann, 1988) définissent les jeux de dialogue comme étant des ensembles de règles décrivant le comportement des agents. Le jeu de dialogue est donc une structure partagée par les agents.
- La théorie des jeux permet de définir autrement les jeux. L'idée est de considérer les actes de dialogue comme des coups dans un jeu et d'authoriser ou d'interdire certains coups en

tenant compte des coups précédemment joués et du contexte cognitif (des états mentaux des agents). Cette vision des jeux de dialogue se retrouve par exemple dans les travaux de (MacKenzie, 1979) et (Beun, 1991; Beun, 1996).

Le modèle présenté ici est basé principalement sur la deuxième approche. Nous allons utiliser les actes de dialogue pour définir des coups et des ensembles de règles qui vont permettre aux agents de choisir les coups à jouer en fonction de l'état courant. Ces ensembles de règles seront structurés en jeux de dialogue. Pour nous, le dialogue sera construit à partir de plusieurs jeux, les agents pouvant jouer des jeux différents selon leurs états mentaux.

En ce qui concerne la représentation des agents, nous utiliserons des notions provenant des systèmes multi-agents. En particulier, nous verrons qu'il est nécessaire pour un agent d'avoir une représentation mentale de l'autre agent (accointances en multi-agent (Gleizes and Fade, 1998)).

Chapitre 2

Représentation des connaissances

2.1 Etats mentaux

Sans entrer dans les détails pour l'instant, nous utiliserons dans la suite une logique modale. La base de connaissances d'un agent sera un ensemble d'expressions qui utilisent des modalités pour typer les états mentaux (croyances, désirs, ...). On pourra par exemple utiliser les modalités du tableau 2.1.

Modalité	Exemple	Signification
B	$B_x p$	L'agent x croit p
MB	$MB_{x,y}p$	Les agents x et y croient *mutuellement* p
KIF	$KIF_x p$	L'agent x sait si p
K	$K_x p$	L'agent x sait p
W	$W_x p$	L'agent x désire que p devienne vrai

TAB. 2.1: Exemples de modalités

Il existe plusieurs axiomatiques possibles utilisant cette logique modale. Il faut tout d'abord remarquer que ces modalités ne sont pas toutes indépendantes. On a en effet :

$$\begin{cases} MB_{x,y}p \equiv B_x p \wedge B_y p \wedge MB_{x,y}(B_x p \wedge B_y p) \\ KIF_x p \equiv B_x p \vee B_x \neg p \end{cases}$$

Il est important de remarquer que la croyance mutuelle ne peut pas être considérée comme un état mental. En effet, sa définition fait intervenir les deux agents simultanément. Donc si l'on veut éviter l'utilisation de structures partagées[1], il faudra considérer des croyances sur ce que sont les croyances mutuelles : $B_x MB_{x,y}p$.

Le savoir K est une modalité plus forte que la croyances B. On ne peut savoir que des choses qui sont vraies alors que les croyances peuvent être en conflit avec l'état du monde.

En ce qui concerne les axiomatiques, on évoquera rapidement le système $S5$ (voir (Chellas, 1980)) par exemple. Ce système, adapté à notre domaine d'application pour modéliser les croy-

[1]Voir « Récapitulatif des choix effectués », page 35

ances, peut s'écrire :

$$(K) \quad \frac{B_xP \wedge B_x(P \rightarrow Q)}{B_xQ}$$

$$(T) \quad \frac{K_xP}{P}$$

$$(4) \quad \frac{B_xP}{B_xB_xP} \qquad \text{(introspection positive)}$$

$$(5) \quad \frac{\neg B_xP}{B_x\neg B_xP} \qquad \text{(introspection négative)}$$

L'axiome (T) ne s'applique pas à la croyance, alors que les axiomes (K), (4) et (5) s'appliquent aussi au savoir.

Ce système de règles est bien adapté à la croyance, mais il existe d'autres modalités, dont l'intention (modalité W). Dans la suite, on présentera une approche intentionnelle de la modélisation d'agents. L'intention doit donc pouvoir être produite et utilisée par des règles.

$$(W_+) \quad \frac{W_xKIF_xQ \wedge B_x(P \rightarrow Q)}{W_xKIF_xP} \qquad \text{(apparition directe d'une intention)}$$

$$(W'_+) \quad \frac{B_x?_yQ}{B_xW_yKIF_yQ} \qquad \text{(apparition d'une croyance sur les intentions de l'autre agent)}^{2}$$

$$(W_-) \quad \frac{KIF_xQ}{\neg W_xKIF_xQ} \qquad \text{(disparition d'une intention)}$$

Ces règles expriment l'apparition et la disparition d'intentions particulières, à savoir l'intention de connaître la valeur de vérité d'une formule.

2.2 Actes de langage - Actes de dialogue

En linguistique, la notion d'actes de langage (voir (Searle, 1969)) est connue depuis longtemps. Pour présenter l'intérêt de la notion d'actes de langage, on se reportera à l'essai de philosophie du langage de (Searle, 1969). Pour résumer voici la quatrième de couverture du livre :

> *Les individus communiquent, disent des choses, posent des questions, donnent des ordres, font des promesses, présentent des excuses... Ils ont parfois l'intention de signifier réellement ce qu'ils disent.*
>
> *Pour l'auteur, parler une langue, c'est adopter un comportement, accomplir des actes de langage conformément à des règles complexes. Si le langage est un comportement, il doit être absorbé par le biais d'une théorie des actes de langage, qui rejoint ici une théorie de l'action.*
>
> *Sous cet angle, cette recherche est complémentaire de celle de Chomsky, dans la mesure où celui-ci écarte de sa description de la langue le contexte extra-linguistique et même la fonction de communication, essentielle pour Searle. A travers ces conventions extra-linguistiques qui gouvernent l'usage des expressions dans des contextes donnés, on peut distinguer les relations entre ce qui est dit, ce que cela veut dire, ce que veut dire celui qui parle, etc. Utiliser le langage et parler c'est donc s'engager, assumer des obligations. Ignorer cet emploi « engagé » des mots, c'est ignorer le langage lui-même.*

Cependant, nous utiliserons ici la notion d'actes de dialogue ((Bunt, 1996; Bunt, 1994)) qui est plus large et plus adaptée au contexte[3]. Bunt distingue trois aspects de l'acte de dialogue : la forme de l'énoncé, sa fonction communicative et son contenu sémantique. Par exemple, l'énoncé « Est-ce qu'il pleut ? », la fonction communicative « question fermée » et la proposition « il pleut » ont pour effet de rajouter l'énoncé « Est-ce qu'il pleut ? » au contexte linguistique et d'ajouter dans les croyances de l'auditeur que le locuteur veut savoir si la proposition « il pleut » est vraie.

Les actes seront représentés par des modalités. Nous utiliserons par exemple celles du tableau 2.2.

Acte	Exemple	Signification
!	$!_x p$	L'agent x affirme p.
?	$?_x p$	L'agent x a posé la question p.
!?	$!?_x p$	L'agent x affirme son ignorance sur p.
$	$\$_x$	L'agent x n'a plus rien à dire (fermeture du dialogue).

TAB. 2.2: Exemples d'actes de dialogue

On notera $\mathcal{L}$ le langage ainsi obtenu.

Nous utiliserons en plus des connecteurs de la logique classique, des connecteurs particuliers pour représenter les règles des jeux de dialogue. Une règle générale aura la forme suivante : $LHS \Longrightarrow RHS|COND\#F$

LHS : Partie gauche de la règle

RHS : Partie droite de la règle

$COND$: Condition d'application

F : Force de la règle

La condition $COND$ permet d'exprimer des conditions ne pouvant pas s'écrire en logique modale. Cela sera surtout utile lors de l'implémentation. La force F code « l'importance[4] » de la règle, elle a été introduite pour orienter certains choix, nous verrons plus loin à quoi elle correspond.

On notera $\mathcal{L}'$ le langage obtenu à partir de $\mathcal{L}$ et de $\Longrightarrow$.

2.3 Mémorisation des connaissances

Pour pouvoir implanter un système suffisamment souple, à la différence de Beun, on n'utilisera qu'un seul espace de stockage pour tous les états mentaux de l'agent. Beun propose de séparer les états mentaux en trois catégories ($B_x p$, $B_x \neg B_y q$, $MB_{x,y} r$). Ces trois catégories d'états mentaux permettent déjà de simuler quelques dialogues mais cette structuration est assez restrictive. On pourra toujours simuler un tel partitionnement des états mentaux en filtrant ceux-ci selon leur modalité.

Pour présenter la suite du modèle, nous avons besoin d'une logique modale avec un mécanisme de mise à jour de la base de connaissances capable de maintenir un certain degré de consistance (locale) en utilisant un ensemble de règles d'inférence. La consistance locale est suffisante et permet d'éviter l'omniscience[5] de l'agent. Nous supposerons que nous disposons de tels outils que nous verrons plus tard en détail.

[3]N. Maudet, 1998, travaux en cours

[4]Cette « importance » peut être une force ou une probabilité par exemple

[5]Voir page 35

Chapitre 3

Structure du dialogue

Il est nécessaire de structurer le dialogue pour pouvoir en tirer des informations et surtout pouvoir continuer à dialoguer de manière cohérente. Pour cela, on va partir des éléments atomiques du dialogue, les coups, pour définir les actes de langage, les jeux et finalement le dialogue.

3.1 Les coups

Les coups correspondent au contenu littéral des échanges entre les agents. Dans la pratique, ils seraient le résultat d'une analyse syntaxique superficielle[1] des énoncés échangés par deux personnes. Le coup est l'acte de surface contenu dans un énoncé. Un coup est une formule de la logique modale préfixée par une modalité de coup (!, ?, !?, ...).

3.2 Les actes

La notion de coup est très superficielle. Dans un dialogue réel, le coup ne signifie pas toujours ce que l'auditeur désire réellement signifier en jouant ce coup. C'est le cas pour une question indirecte, pour une question qui aurait la valeur d'une affirmation. Les actes de langage résultent donc d'un processus cognitif qui utilise bien entendu les coups, mais aussi le contexte cognitif de l'agent. Pour rendre compte de ce phénomène, nous introduirons des règles[2] destinées à faire cette analyse. Cependant, les coups et les actes seront représentés en logique de la même manière et dans la suite, nous ne parlerons que de coups. On peut en effet assimiler les actes à des coups au niveau du codage.

3.3 Les jeux de dialogue

Jusqu'à présent, les entités présentées étaient atomiques, uniquement représentées par une formule de la logique modale $\mathcal{L}$. Les coups sont les éléments constitutifs du dialogue, mais on ne sait toujours pas comment les produire. On définit pour cela les jeux de dialogue. Un jeu de dialogue est un ensemble de règles qui définissent le comportement de l'agent en fonction de son contexte cognitif. Lorsque l'état mental d'un agent correspond à l'état requis par un jeu[3], l'agent peut entrer dans ce jeu et son comportement est dicté par les règles qui définissent le jeu.

La description des jeux de dialogue que l'on va voir ici est générale. Nous verrons plus loin comment définir des jeux intentionnels ou conventionnels.

---

[1] Voir Annexe « Analyse du langage naturel », page 106

[2] Voir la description des jeux de dialogue et des règles de mise à jour, page 16

[3] Par exemple, lorsqu'un agent a « l'intention » de savoir quelque chose, il peut rentrer dans un jeu de question (dont le but est d'obtenir une réponse)

La première version du simulateur proposait pour représenter le dialogue une suite de coups. Cette suite était linéaire, sans aucune hiérarchie. En fait, la seule structure qui apparaissait était la structure de pile dans la succession des questions et de leurs réponses, il y avait un emboîtement des questions, et les agents tentaient de répondre en priorité aux questions les plus récentes. Ce que l'on veut obtenir ici est une formalisation plus stricte de ce type de phénomènes. On considérera donc que le dialogue peut se décomposer en plusieurs jeux (éventuellement de types et de buts différents) dont la structure n'est plus linéaire, chaque jeu pouvant aussi se décomposer en sous-jeux nécessaires à son succès ((Maudet and Evrard, 1998)). Par exemple, dans le cadre d'un dialogue coopératif, pour résoudre un problème, on peut avoir d'abord besoin de résoudre d'autres problèmes dont les solutions, une fois mises en commun, permettent de répondre à la question initiale. On retrouve cette structure dans les problèmes de planification par exemple.

Il faut maintenant pouvoir caractériser un jeu. On vient de voir que la structuration du dialogue en jeux permet de spécifier le but de chaque étape du dialogue. Cela permet aussi de découper un dialogue en plusieurs jeux de types et/ou de buts différents. Pour caractériser cela, il faut pouvoir répondre aux questions suivantes :

- Comment entre-t-on dans un jeu ?
- Pourquoi reste-t-on dans un jeu ?
- Quand sort-on d'un jeu ?
- Quelles sont les règles propres au jeu et comment les utiliser ?

Il apparaît donc qu'un jeu est caractérisé par un ensemble de conditions qui décrivent l'entrée et la sortie du jeu et par un ensemble de règles qui dictent le comportement de l'agent dans le jeu. Dans la pratique, un jeu sera identifié par un terme non nécessairement complètement instancié. On pourra donc emboîter plusieurs instances du même modèle de jeu, mais on se restreindra à l'unicité d'une instance à un instant donné. En effet, si l'on est déjà dans un jeu, il est inutile de le réactiver. Il n'est cependant pas exclu d'avoir dans l'arborescence des jeux représentant le dialogue plusieurs occurrences d'un jeu instanciées de la même manière, à condition qu'aucune ne soit ancêtre d'une autre. C'est-à-dire que l'on peut rentrer dans un jeu duquel on est déjà sorti.

Les jeux de dialogue ont donc principalement deux fonctions :

- Décrire le passage d'un jeu à l'autre
- Décrire le comportement de l'agent.

Les jeux seront donc définis par deux types d'information :

- Des conditions de changement de jeu
- Des règles manipulant les coups et les états mentaux.

Nous verrons dans la suite que les conditions de changements d'états sont composées de conditions d'entrée, d'invariants et de conditions de sortie. Les règles qui définissent les jeux sont de quatre types : règles de perception, règles de mise à jour, règles de génération et règles d'interdiction.

3.3.1 Exemple de dialogue

Avant de définir précisément les jeux de dialogue, considérons un petit exemple montrant la structuration du dialogue en jeux. Nous prendrons deux agents avec un comportement coopératif, l'un deux posant une question. Le dialogue obtenu est présenté dans le tableau 3.1.

Dans l'état initial (1), l'agent y veut savoir si r. Il pose donc la question (2). L'agent x reconnaît donc l'intention de y de savoir si r (3) et a donc l'intention de savoir si q (qu'il connaît comme étant une justification de r), il pose la question q (4). L'agent y ne peut pas répondre (6). L'agent x a alors l'intention de savoir si p (7) et pose la question (8). A partir de là, les agents peuvent répondre aux questions posées et le dialogue se termine. Pour ce dialogue, nous avons besoin de trois jeux : un jeu de base et des jeux de question et réponse. La structure du dialogue est présentée dans le tableau 3.1.

	Agent x		Coups	Agent y	
	Jeux	Etats mentaux		Etats mentaux	Jeux
1	Jeu de base	$B_x(p \to q)$ $B_x(q \to r)$		$B_y p$ $W_y KIF_y r$	Jeu de base Question(r)
2			$?_y r$		
3	Reponse(r) Question(q)	$B_x W_y KIF_y r$ $W_x KIF_x q$			
4			$?_x q$		
5				$B_y W_x KIF_x q$	Reponse(q)
6			$!?_y q$		
7	Question(q)	$W_x KIF_x p$			
8			$?_x p$		
9				$B_y W_x KIF_x p$	Reponse(p)
10			$!_y p$		
11		$B_x p$ $B_x q$			~~Reponse(p)~~
12	~~Question(p)~~		$!_x q$		
13	~~Question(q)~~	$B_x r$		$B_y q$	
14			$!_x r$		~~Reponse(q)~~
15	~~Reponse(r)~~			$B_y r$	~~Question(r)~~

TAB. 3.1: Exemple de dialogue

3.3.2 Conditions de changement de jeu

Le changement de jeu est caractérisé par trois types de conditions. Sachant qu'un jeu correspond à un comportement déterminé par une certaine configuration du contexte cognitif, une modification des états mentaux peut provoquer la sortie du jeu courant ou l'entrée dans un autre jeu. On voit donc apparaître des conditions d'entrée et de sortie. Si l'on regarde de plus près, les conditions de sortie sont de deux types. La sortie d'un jeu est due soit à la réalisation d'un but local, soit à un échec. Pour décrire cela, on rajoute des invariants propres au jeu.

Remarquons tout d'abord qu'une telle définition des jeux suppose que l'on peut emboîter les jeux, pour résoudre des sous-problèmes par exemple (il n'est pas nécessaire de sortir d'un jeu pour rentrer dans un autre). Le dialogue global est donc caractérisé par une arborescence de jeux.

Conditions d'entrée

Les conditions d'entrée sont des conditions portant sur l'état mental de l'agent. Elles doivent être vérifiées pour que l'agent puisse rentrer dans le jeu. Ces conditions permettent de ne pas utiliser des jeux inadaptés au dialogue à un instant donné, dans un contexte particulier. Par exemple, si l'un des deux agents a posé une question, l'autre agent rentrera dans un jeu de réponse dont les conditions d'entrée sont vérifiées. Ces conditions sont testées uniquement avant de rentrer dans le jeu. En cours de jeu, la falsification de ces conditions n'intervient pas dans les changements éventuels de jeu.

Invariants

Une fois l'agent rentré dans le jeu, il faut pouvoir exprimer ses motivations à rester dans ce jeu. C'est le rôle des invariants. Ces conditions doivent rester vraies tout le long du jeu. La violation d'un invariant provoque, contrairement aux conditions d'entrée, la sortie du jeu. Dans ce cas, le jeu a échoué et l'agent retourne au jeu englobant.

Conditions de sortie

Les conditions de sortie caractérisent le succès du jeu. A leur satisfaction, l'agent sort du jeu et retourne dans le jeu englobant.

Dans tous les cas, quand on sort d'un jeu, on retourne dans le jeu immédiatement supérieur. Le passage d'un jeu à un autre nécessite d'abord de sortir du premier jeu, puis de rentrer dans le second.

La manière de passer d'un jeu à un autre n'est pas toujours simple. Dans sa thèse (Awada, 1993) présente un moteur de planification. L'un des problèmes rencontrés est l'activation de plans, ce qui est tout à fait similaire à l'activation d'un jeu. Il propose des précontraintes qui doivent être vérifiées pour activer le plan, et des préconditions que le planificateur doit essayer de satisfaire. Ces notions se retrouvent dans les jeux de dialogues présentés ici. Pour rentrer dans un jeu, il faut que les conditions d'entrée soient satisfaites, et que les invariants (après activation du jeu) soient vérifiés. Appliquer des règles de mise à jour des états mentaux pour satisfaire les invariants revient en quelques sortes à essayer de satisfaire des préconditions.

3.3.3 Règles

Les règles qui définissent un jeu doivent décrire deux actions essentielles : perception et génération des coups. Rappelons que nous ne traiterons ici que les coups et non les actes de langage, le coup étant en quelque sorte « l'acte de langage de surface ». Cela simplifie le traitement des coups. La prise en compte des actes de langage proprement dits se fera au niveau des états mentaux. Nous n'avons donc pas besoin de considérer les actes de langage, mais plutôt des modifications des états mentaux qui dépendent des coups et des états mentaux eux-mêmes.

Le rôle d'un jeu de dialogue est de définir le comportement de l'agent selon le contexte dans lequel il se trouve. Un jeu de dialogue doit donc assurer deux tâches : l'interprétation et la génération. Pour cela, on va utiliser plusieurs ensembles de règles qui vont décrire ce comportement. Tout d'abord, le précédent coup a été produit dans le but d'agir sur l'auditeur et même dans certains cas pour le faire réagir afin d'obtenir une information que l'on veut connaître, ou pour obtenir de lui quelque chose que l'on ne peut ou ne veut faire soi-même. La première étape est donc évidente, le dernier coup produit (ou les derniers coups produits) doit être analysé pour modifier l'état mental de l'agent. Ensuite, il faut pouvoir produire de nouvelles connaissances. A partir de là, on peut commencer la génération proprement dite du coup. On peut découper cette dernière phase en deux étapes. Tout d'abord, on produit des coups candidats à la génération, puis on filtre ces coups. On obtient finalement un ensemble de coups. Nous verrons plus tard comment choisir parmi ces coups.

Un jeu est caractérisé par quatre types de règles. Ces règles doivent permettre de modifier les états mentaux et de générer des coups en fonction des états mentaux et des coups précédemment joués. Pour cela, on a besoin de règles qui intègrent les coups aux états mentaux (règles de perception), de règles qui décrivent le fonctionnement interne de l'agent (règles de mise à jour des états mentaux), de règles de génération et d'interdiction de coups.

Dans la suite, $\mathcal{C}$ désigne l'ensemble des coups et $\mathcal{EM}$ celui des états mentaux.

Règles de perception : $\mathcal{C} \longrightarrow \mathcal{EM}$

Les règles de perception mettent en relation les coups joués avec les états mentaux de l'agent. C'est la seule « entrée » que possède l'agent. Elles associent à un coup ou une séquence de coups une formule de $\mathcal{L}$. Lors de la reconnaissance d'un coup associé à une règle de perception, les états mentaux de la partie droite de la règle sont utilisés pour la mise à jour de la base de connaissances courante. Ces règles sont appliquées après la génération d'un coup.

Exemple :

$$!_y P \Longrightarrow B_x !_y P \quad \text{(mémorisation d'un coup)}$$

Règles de mise à jour des états mentaux : $\mathcal{EM} \longrightarrow \mathcal{EM}$

Les règles de mise à jour décrivent le fonctionnement interne de l'agent. A partir d'une expression portant sur les états mentaux courants, la règle produit une combinaison d'états mentaux dont le comportement est le même que pour les règles de perception.

Exemples :

$B_x ?_y Q \Longrightarrow B_x W_y KIF_y Q$	(interprétation des coups)
$B_x B_y P \wedge \neg B_x \neg P \Longrightarrow B_x P$	(appropriation de croyances)
$W_x KIF_x Q \wedge B_x (P \rightarrow Q) \Longrightarrow W_x KIF_x P$	(décomposition de problèmes en sous-problèmes)

Règles de génération : $\mathcal{EM} \longrightarrow \mathcal{C}$

Les règles de génération servent à produire des coups en fonction des états mentaux de l'agent. La partie gauche d'une règle de génération est une expression portant sur les états mentaux courants de l'agent. La partie droite est le coup généré. Ces règles ne sont en fait qu'une partie des règles de génération proposées par Beun. L'autre partie correspond aux règles d'interdiction.

Il se peut qu'aucune règle de génération ne puisse être appliquée. Dans ce cas, on considère que l'agent « ne sait pas » quoi dire (c'est différent du coup qui permet à l'agent de terminer le dialogue). C'est un blanc dans la conversation, un « coup vide ».

Exemples :

$B_x W_y KIF_y Q \wedge B_x Q \Longrightarrow !_x Q$	(réponse à une question)
$W_x KIF_x Q \wedge \neg B_x \neg KIF_y Q \Longrightarrow ?_x Q$	(question)

Règles d'interdiction : $\mathcal{EM} \longrightarrow \mathcal{C}$

Les règles d'interdiction servent à filtrer les coups générés. Les règles de générations produisent un ensemble de coups candidats à la génération. Les règles d'interdiction permettent d'éliminer certains de ces coups. Elles correspondent par exemple à des maximes ((Grice, 1975))[4].

Exemple :

$$\neg B_x P \vee B_x \neg P \Longrightarrow \neg !_x P \quad \text{(sincérité)}$$

Beun regroupe ces deux types de règles (génération et interdiction). Cela n'est pas limitatif, mais il semble qu'énoncées ainsi, les règles sont plus simples, et on retrouve un comportement assez naturel, en effet, dans certains cas, une proposition peut avoir à la fois des raisons d'être énoncée et des raisons de ne pas l'être, il y a des règles de nature générative (donner la réponse à une question) et d'autres de nature prohibitive (ne pas se répéter, ne pas dire ce que l'on croit faux). Cette distinction permettra aussi de traiter le problème du clash[5] et du flout[6], en séparant les causes qui provoquent la génération de celles qui provoquent l'interdiction de certains coups. Si la « force de génération » est supérieure à celle d'interdiction, l'agent va jouer un coup. Cette situation peut correspondre à un clash (impossibilité de satisfaire à la fois les règles de génération et les règles d'interdiction correspondantes) ou à un flout (une règle de génération a une force supérieure aux autres, même en étant incompatible avec les règles d'interdiction correspondantes).

[4]Voir Annexe « Maximes de Grice », page 78

[5]Clash : Violation de maximes de conversation lorsqu'il est impossible de les satisfaire toutes simultanément. Par exemple, les maximes de quantité et de qualités sont incompatibles quand l'agent doit répondre à une question et qu'il ne possède qu'une information incomplète ou imprécise (« A quelle heure part le train ? Vers 10 heures. » Ici l'agent respecte la maxime de qualité mais transgresse celle de quantité).

[6]Flout : Violation de maximes de conversation sans qu'il y ait nécessairement clash. Par exemple lorsqu'un agent transgresse la maxime de quantité pour ne pas dévoiler des détails qu'il préfère garder secrets. (« Quel est votre salaire ? C'est un salaire moyen. »).

Définition d'un jeu

Pour résumer, un jeu est défini par :

$$G = \langle C_{in}, C_{inv}, C_{out}, R_p, R_u, R_g, R_i \rangle$$

C_{in}, C_{inv}, C_{out} sont des formules de $\mathcal{L}$ et représentent respectivement les conditions d'entrée, les invariants et les conditions de sorties.

R_p, R_u, R_g, R_i sont des sous-ensembles de $\mathcal{L}'$ et représentent respectivement les règles de perception, de mise à jour, de génération et d'interdiction.

Toutes ces conditions et ces règles sont nécessaires pour la définition d'un jeu. Etant donné qu'un agent peut jouer des jeux différents selon le contexte cognitif dans lequel il se trouve, il faut pouvoir exprimer les changements de jeux (condition d'entrée, invariant et condition de sortie). Les règles de perceptions sont nécessaires car elles constituent la seule « entrée » que possède l'agent. Elles pourraient être définies au niveau de l'agent mais elles seraient alors indépendantes du jeu. Les règles de mise à jour des états mentaux sont de même nature que les règles d'inférence propres aux agents. Il faut cependant les définir dans les jeux puisqu'elles sont caractéristiques d'un comportement particulier (celui défini par le jeu), par exemple l'appropriation de croyances ne sera pas la même dans un jeu coopératif et dans un jeu non coopératif. Il en est de même pour les règles de génération et d'interdiction. La génération de coup est le résultat d'un comportement dépendant du contexte, donc du jeu dans lequel l'agent se trouve. Par exemple, le coup !? qui marque l'ignorance d'un agent sera utilisé dans un jeu de réponse, mais pas dans un jeu de question.

La définition précédente ne fait pas apparaître la stratégie utilisée par l'agent pour choisir parmi tous les coups générables. Les règles de génération et d'interdiction produisent un ensemble de coups qui n'est pas nécessairement réduit à un singleton. Lorsque plusieurs coups sont générables, le choix se fait sur la force des coups (différence entre les forces de génération et d'interdiction). Nous verrons dans les extensions proposées comment utiliser des heuristiques plus fines pour réaliser ce choix.

3.3.4 Hiérarchie des jeux

Hiérarchie statique

Les jeux de dialogue ne sont pas tous indépendants. Certains jeux sont des extensions d'autres jeux. On définit donc une relation d'héritage des jeux. Cela permet de définir un jeu à partir d'un autre. La relation d'héritage vérifie la propriété suivante :

Soit $(G_j)_{j=1\ldots n} = (\langle C_{in,j}, C_{inv,j}, C_{out,j}, R_{p,j}, R_{u,j}, R_{g,j}, R_{i,j} \rangle)_{j=1\ldots n}$ une famille de n jeux. Si le jeu $G = \langle C_{in}, C_{inv}, C_{out}, R_p, R_u, R_g, R_i \rangle$ hérite de la famille de jeux , on a la relation suivante :

$$\forall k \in \{p, u, g, i\} \quad \bigcup_{j=1}^{n} R_{k,j} \subset R_k$$

Donc, lorsqu'un jeu hérite d'un autre, ses conditions de changement de jeux ne sont pas modifiées mais les règles qui le définissent sont rajoutées à son ensemble de règles. L'importation de règles n'est donc que la réunion ensembliste de règles. N. Maudet propose la redéfinition de règles au moment de l'importation. Cela est tout de même rendu possible en utilisant des poids supérieurs pour les règles devant en redéfinir d'autres.

Ce type de hiérarchie est dit statique car les relations entre les jeux sont définies au moment de la définition des jeux et restent figées pendant tout le déroulement du dialogue. Ce n'est qu'un moyen pratique de définir aisément une famille de jeux ayant des caractéristiques communes.

Hiérarchie dynamique

Tels qu'ils ont été définis, les jeux peuvent être imbriqués pendant le dialogue (la structure même du dialogue est calquée sur l'imbrication des jeux). Les jeux définis précédemment de manière statique sont empilés pendant le dialogue. La pile des jeux représente l'état du dialogue à un instant donné. Dans un empilement donné, on applique en priorité les règles du jeu du sommet de la pile. Tant qu'aucun coup n'est générable, on recommence le processus avec les jeux moins imbriqués (donc stockés à une profondeur plus grande dans la pile). Cette structure est appelée hiérarchie dynamique car elle dépend de l'état du dialogue et évolue au fur et à mesure de l'élaboration du dialogue.

3.4 Le dialogue

On peut définir le dialogue de deux manières différentes. Selon une première définition, plus commune, le dialogue est une succession d'échanges de coups. Cette définition n'est pas suffisante. En effet, on a vu que les jeux définissent une structure arborescente. Comme l'on ne peut pas appeler « dialogue » toute séquence de coups (tout comme une phrase n'est pas une suite quelconque de mots, ou un discours n'est pas une suite quelconque de phrases), il faut trouver une définition qui tienne compte de la structure produite par les jeux. On dira donc qu'un dialogue est une séquence de coups ordonnés qui sont les feuilles d'un arbre produit par les jeux. Les jeux sont les nœuds de l'arbre. Tout jeu est le fils du jeu dans lequel l'agent se trouvait avant d'y rentrer. Remarquons que cette structure permet de considérer les coups comme des jeux élémentaires.

Un dialogue est donc caractérisé par les coups qui le constituent mais aussi par la manière dont les coups ont été générés, c'est-à-dire par l'arbre des jeux.

La représentation du dialogue par une arborescence de jeux est à mettre en parallèle avec la représentation de la forme syntaxique d'une phrase.

Chapitre 4

Formalisation des agents

Nous avons vu comment définir les jeux de dialogue. Ces jeux correspondent à des comportements qui vont affecter les agents pendant le dialogue. Il reste donc à formaliser les agents et à voir comment un agent peut utiliser un jeu.

4.1 Définition d'un agent

Un agent est une entité douée de capacités cognitives (moteur d'inférence) et connaissant un ensemble de jeux. Chaque agent possède de plus une base de connaissances et une représentation mentale de l'autre agent.

Un agent est donc défini par $\langle KB, G, A_y, d\rangle$, où : $KB \subset \mathcal{L}$, $G \subset GAMES$, $GAMES$ est l'ensemble des jeux prédéfinis, A_y est le nom de l'agent qui sert de représentation mentale de l'autre agent, $d \in \mathbb{N}$, d est la profondeur limite du moteur d'inférence[1].

On définit de la même manière que pour les jeux une relation d'héritage (statique) entre agents. Donc si un agent $A = \langle KB, G, A_y, d\rangle$ hérite d'une famille d'agents $(A_i)_i = (\langle KB_i, G_i, A_{y,i}, d_i\rangle)_i$, on a la propriété suivante :

$$\begin{cases} \bigcup_i KB_i \subset KB \\ \bigcup_i G_i \subset G \end{cases}$$

Ici la réunion des bases de connaissances est en fait la construction incrémentale de la base avec éventuellement un mécanisme de maintien de la consistance.

4.1.1 Base de connaissances

La base de connaissances KB est constituée de formules de $\mathcal{L}$. Cette base contient les connaissances de l'agent (sous la forme $B_x p$) mais aussi tous les états mentaux qui caractérisent l'agent à un instant donné (les intentions par exemple). C'est sur cette base de connaissances que s'appliquent les règles définies dans les jeux ainsi que le moteur d'inférence de l'agent.

4.1.2 Fonctionnement des jeux chez un agent

Après avoir décrit les jeux et les agents, on va montrer comment un agent peut utiliser les jeux pour dialoguer. Nous avons vu que l'agent possède une base de connaissances sur laquelle il peut travailler de manière autonome au moyen du moteur d'inférence. La notion de jeu va permettre d'orienter le processus cognitif de l'agent dans le sens du dialogue. Pour cela, l'agent va utiliser les règles définies dans le jeu. Nous avons déjà décrit chaque type de règle (perception, mise à jour, génération, interdiction). Après avoir rappelé brièvement le fonctionnement des jeux, nous

[1]Limiter la profondeur de fonctionnement du moteur d'inférence permet d'éviter l'omniscience, voir page 35

verrons comment résoudre le problème des choix face aux alternatives qui se présentent en phase de génération et de changement de jeu.

La première fonction d'un jeu est de générer un coup dans le contexte courant. La génération se fait en trois étapes :

1. Perception
2. Génération
3. Interdiction

Les règles de mise à jour des états mentaux sont utilisées à chaque apparition ou disparition d'un état mental.

Le processus de génération d'un coup est résumé dans la figure 4.1.

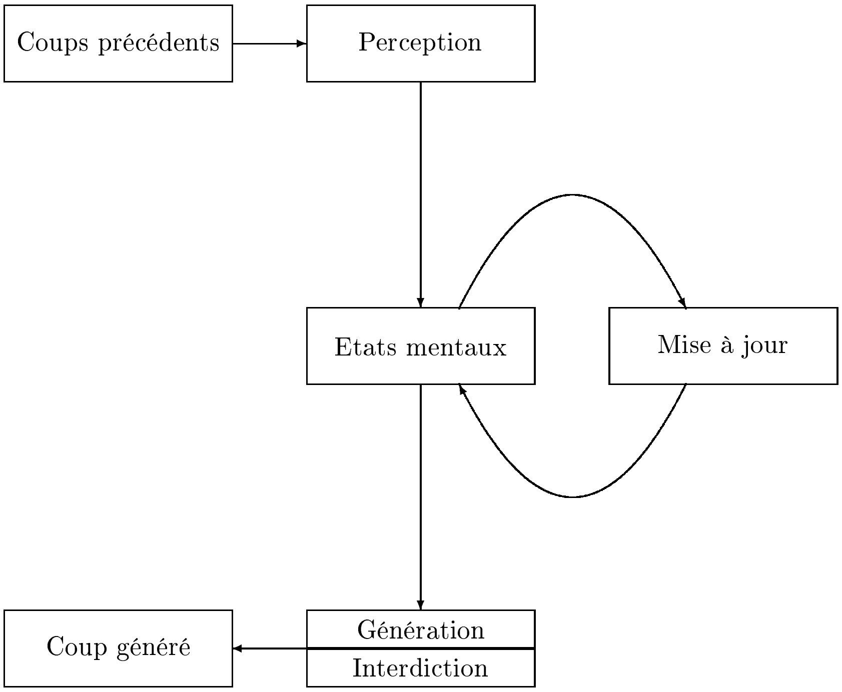

Fig. 4.1: Processus de génération d'un coup

Il reste cependant un problème au niveau de la génération du coup. En effet, plusieurs règles de génération peuvent être appliquables et après filtrage par les règles d'interdiction, il peut encore rester plusieurs coups générables. Dans ce cas, il faut pouvoir choisir le coup à générer. Il faudrait pouvoir choisir le coup le plus pertinent, celui dont les règles de génération sont les plus importantes ou celui dont les états mentaux déclencheurs sont les plus "forts". On associe donc à chaque état mental et à chaque règle une force. La force d'un état mental peut être le nombre d'utilisations de cet état, une probabilité, ... On présentera plus loin deux implémentations utilisant deux forces différentes. Le calcul de la force d'une formule respecte les règles suivantes[2] :

$$\begin{cases} f(A \wedge B) = \min(f(A), f(B)) \\ f(A \vee B) \leq \max(f(A), f(B)) \end{cases}$$

L'utilisation d'une force associée aux états mentaux et aux règles permet de donner des forces de génération (associée à la règle de génération utilisée) et d'interdiction (associée à la règle d'interdiction utilisée). Les coups générables sont alors affectés d'une combinaison des forces de génération

[2]Voir cours de DEA (Raisonnement Plausible et Incertitude), Logiques possibilistes et probabilistes. (Cayrol and Prade, 1998)

et d'interdiction et on génère finalement le coup de force maximale. Les forces de génération et d'interdiction sont associées aux règles de génération et d'interdiction correspondantes, elles font donc partie de la définition du jeu, ce qui assure de la part des agents un certain respect des règles. Les forces qui proviennent de l'agent sont celles qui correspondent à leurs états mentaux.

4.1.3 Représentation mentale d'un autre agent

Le dialogue entre deux agents n'est possible que si les deux agents se connaissent. Si l'un des deux agents n'a aucune représentation de l'agent avec lequel il parle, le dialogue est inconcevable. Dans un dialogue réel, il y a toujours chez les deux agents une image, même réduite, de l'autre. Prenons le cas extrême de deux passants qui ne se connaissent pas. Ceux-ci se croisent dans la rue, l'un demande l'heure à l'autre. Avant même le début du dialogue, l'agent qui pose la question avait établi un contact visuel et était conscient de l'existence de l'autre agent, ce qui était nécessaire pour ce dialogue. Ceci peut paraître trivial mais il est nécessaire de le formaliser pour pouvoir simuler informatiquement le dialogue. Ce « premier contact » est simulé de manière implicite dans la déclaration des deux agents que l'on va faire dialoguer. Mais cette connaissance reste très rudimentaire. Dans de nombreux cas, une représentation plus complexe est nécessaire. Par exemple, dans une relation entre un professeur et un élève, le professeur a besoin de savoir comment raisonne l'élève afin de pouvoir l'aider. Il faut donc absolument que les agents possèdent une image mentale de l'agent avec lequel ils vont dialoguer. Nous l'avons vu, cela permet de savoir avec qui dialoguer. Mais ce n'est pas le seul intérêt. En effet, pour que le dialogue ne se limite pas à un échange de coups générés de manière purement réactive, un agent doit pouvoir tenir compte dans le processus de génération de l'état mental de l'autre agent. Il est évident qu'une telle représentation est nécessaire pour faire de la reconnaissance de plan par exemple. Pour illustrer ceci, considérons l'exemple suivant :

Une personne veut aller à Nancy, elle demande : « Connaissez-vous l'heure du prochain train pour Paris ? » On peut imaginer quatre réponses possibles : « Oui » : On répond à la question contenue dans le coup (l'acte de surface). En fait l'employé SNCF sait qu'il a affaire à un client qui veut connaître l'heure de départ du prochain train. La réponse minimale qu'il attend est donc un horaire : « 8h47 » Si l'employé sait que le client veut aller à Nancy, il pourra répondre : « 8h47. Mais le train de 9h00 arrive plus tôt à Nancy » : L'employé sait que le client veut aller à Nancy et suppose que s'il veut partir le plus tôt possible, c'est pour arriver le plus tôt possible. On peut pousser plus loin le raisonnement. L'employé croit que le client veut arriver à Nancy le plus tôt possible et semble vouloir passer par Paris : « 8h47. Mais le train de 9h00 arrive plus tôt à Nancy mais ne passe pas par Paris. »

On le voit bien sur cet exemple et cela se rencontre dans beaucoup de situations, deux agents doivent avoir une certaine forme de connaissance de l'autre pour pouvoir dialoguer. Pour cela, on dote l'agent d'une représentation de l'autre agent. Soit l'agent $A = \langle KB, G, A_y, d \rangle$, où $A_y = \langle KB_y, G_y, A_z, d_y \rangle$ est la description d'un agent. Cette description fait apparaître la connaissance d'un autre agent. On pourra en effet représenter pour un agent la connaissance que l'autre agent a de lui et itérer le processus indéfiniment. Cependant, au-delà de deux niveaux, cette structure devient difficile à appréhender. Au niveau de l'implémentation, il faudra limiter la longueur de cette chaîne.

Pour prendre en compte la représentation mentale de l'autre agent, il suffit alors d'appliquer les mécanismes de simulation exposés précédemment à l'agent A et à l'agent $\tilde{B}_x A_y$. L'opérateur $\tilde{B}_x$ a ici une sémantique particulière. A_y dénote un agent qui n'existe que dans les états mentaux de A. A_y est défini comme un agent classique et l'opérateur $\tilde{B}_x$ permet de passer de cette description classique à la description de l'agent dans les états mentaux de l'agent A. Pour cela, il faut dans A_y permuter les rôles (x et y) puis préfixer tous les éléments de la description par la modalité B_x. Cette transformation doit aussi être réalisée dans les jeux de A_y. Formellement, l'opérateur $\tilde{B}_x$

est défini de la manière suivante :

$$\begin{cases} \tilde{B}_x\langle KB_y, G_y, A_y, d_y\rangle = \langle \tilde{B}_x KB_y, \tilde{B}_x G_y, \tilde{B}_x A_y, d_y\rangle \\ \tilde{B}_x E = \left\{\tilde{B}_x e | e \in E\right\} \\ \tilde{B}_x(P \Longrightarrow Q) = \left(\tilde{B}_x P \Longrightarrow \tilde{B}_x Q\right) \\ \tilde{B}_x(\alpha \star \beta) = \tilde{B}_x\alpha \star \tilde{B}_x\beta, \quad \star \in \{\rightarrow, \wedge, \vee\} \\ \tilde{B}_x\neg\alpha = \neg\tilde{B}_x\alpha \\ \tilde{B}_x\alpha = \tilde{B}_x\left(S_{x,y}\alpha\right) \\ \tilde{B}_x\epsilon = \tilde{B}_x\epsilon \end{cases}$$

E est un ensemble, α et β sont des formules quelconques et ϵ une formule atomique. L'opérateur $S_{x,y}$ permute les constantes x et y dans un terme quelconque.

Avec les notations précédentes, il faut aussi produire les représentations des agents A, $\tilde{B}_x A_y$, $\tilde{B}_x\tilde{B}_y A_z$, ...

Chapitre 5

Différentes approches du dialogue

Nous avons défini la notion de jeu de manière très générale. Dans les systèmes à base d'agents, on peut considérer deux types d'agents, des agents cognitifs[1] et des agents réactifs. Ces deux types d'agents n'utilisent à priori pas les mêmes règles pour générer des coups et donc utilisent des jeux de types différents. On va donc définir une approche intentionnelle et une approche conventionnelle des jeux de dialogue.

5.1 Approche intentionnelle

L'approche intentionnelle des jeux de dialogue est basée sur la reconnaissance d'intention. Elle met en jeu un processus cognitif à base de règles de mise à jour des états mentaux. Les agents qui utilisent de tels jeux auront un comportement cognitif. Les jeux tels qu'ils ont été présentés permettent donc cette approche intentionnelle, nous le verrons dans les exemples qui suivent.

5.2 Approche conventionnelle

Nous voulons ici introduire des règles de dialogue indépendantes du contexte cognitif. Ces règles s'apparentent à un système de production de coups tenant compte uniquement des coups précédents. Un jeu défini avec ce type de règles caractérisera plutôt un comportement réactif. Par rapport aux jeux intentionnels, on a besoin de trois autres types de règles. L'agent doit être capable de reconnaître à partir des coups joués les actes conventionnels qui ont été produits. Ensuite il faut pouvoir générer puis filtrer des coups à partir des actes conventionnels reconnus. On définit donc des règles de mise à jour conventionnelle, de génération conventionnelle et d'interdiction conventionnelle dont l'expression ne contient que des coups. L'approche conventionnelle des jeux de dialogue est basée sur l'interprétation des coups déjà joués. Cela signifie que les conditions d'entrée et de sortie doivent aussi tenir compte des coups joués. Pour cela, on définit des coups d'entrée et de sortie. La définition d'un jeu conventionnel devient alors :

$$\begin{cases} G^c = \langle G^c_{int}, G^c_{conv}, w^c \rangle \\ G^c_{int} = \langle C_{in}, C_{inv}, C_{out}, R_p, R_u, R_g, R_i \rangle \\ G^c_{conv} = \langle C^c_{in}, C^c_{out}, R^c_p, R^c_u, R^c_g, R^c_i \rangle \end{cases}$$

G^c_{int} est la partie intentionnelle du jeu. C'est un jeu tel qu'il a été défini précédemment. G^c_{conv} est la partie conventionnelle du jeu est constituée de conditions conventionnelles (des coups) et de règles conventionnelles (mise à jour, génération, interdiction). w^c est le poids des règles conventionnelles par rapport aux règles intentionnelles. Cela permet de définir des jeux plus ou moins conventionnels en utilisant les deux approches.

[1]On parle ici d'agents cognitifs par opposition à un comportement plus réactif que sont les jeux à base de règles conventionnelles. On verra plus loin que ce type de jeux reste très réactif par rapport aux jeux intégrant un planificateur.

En fait, un jeu conventionnel sera traduit de la même manière qu'un jeu intentionnel, en combinant les conditions et les règles intentionnelles et conventionnelles comme cela est décrit dans la suite.

5.2.1 Changements conventionnels de jeu

On associe à un jeu conventionnel un ou plusieurs coups qui permettent d'entrer dans ce jeu. Ces coups d'entrée permettent de court-circuiter les conditions d'entrée intentionnelles. Cependant, un coup d'entrée n'est effectivement considéré comme valide que si les conditions d'entrée intentionnelles ne sont pas fausses (c'est à dire que l'on ne peut pas montrer la négation de ces conditions). Les nouvelles conditions d'entrée dans un jeu conventionnel sont donc : $C_{in} \vee \left(C_{in}^c \wedge \neg\overline{C_{in}}\right)$, où $\overline{C_{in}}$ dénote la négation de C_{in}. Ainsi, même si l'on rentre dans un jeu de manière conventionnelle, les conditions d'entrée ne seront pas fausses, même si l'on ne peut pas montrer qu'elles sont vraies[2].

Les invariants sont définis de la même manière que pour un jeu intentionnel.

Pour la sortie conventionnelle d'un jeu, on définit un ou plusieurs coups de sortie. La sortie d'un tel jeu est alors $C_{out} \vee C_{out}^c$.

5.2.2 Règles conventionnelles

Les règles conventionnelles seront elles aussi traduites comme des règles intentionnelles. En effet, les règles de mise à jour conventionnelles qui permettent d'interpréter les séquences de coups déjà joués fonctionnent comme des règles de mise à jour intentionnelles. Il en va de même pour les règles de génération et d'interdiction conventionnelles qui fonctionnent sur le même principe que les règles intentionnelles correspondantes.

[2]Dans les implémentations présentées dans la suite, $\overline{C}$ n'aura pas toujours la même signification. Dans l'implémentation avec chaînage avant et arrière, on essaiera de démontrer la négation. Dans l'implémentation probabiliste, on pourra toujours démontrer la négation, c'est la probabilité associée à la formule qui différera.

Chapitre 6

Formalisme de description

Nous allons présenter ici le formalisme utilisé pour décrire les jeux de dialogue et les agents. Ce formalisme est directement utilisable pour décrire effectivement les jeux et les agents en Prolog.

6.1 Introduction

Pour implanter un modèle de la théorie des jeux de dialogue, nous avons besoin d'un langage de description des agents, des jeux et des ensembles de règles utilisées dans les jeux. Pour simplifier l'analyse des descriptions, le langage est basé sur celui de Prolog. Les objets décrits dans le langage sont des termes syntaxiquement analysables par Prolog. Il suffira donc d'écrire l'analyseur sémantique et de produire les clauses Prolog correspondantes.

Nous ne décrivons ici que le langage. La génération de code Prolog dépend de l'implémentation retenue et sera présentée en annexe.

6.2 Présentation du langage

Le langage doit pouvoir décrire les agents et les jeux. Dans les deux cas, il s'agit de fabriquer un ensemble de couples attribut/valeur.

Dans la suite le langage sera décrit par une grammaire, les symboles terminaux sont en minuscule, les non-terminaux en majuscule.

6.2.1 Description des jeux

JEU $\longrightarrow$

game NOM = { DESCR_JEU* } .

Le nom d'un jeu est un terme Prolog contenant les variables qui caractérisent les instances des jeux de ce type.

Un jeu peut posséder les attributs suivants :

- Conditions d'entrée
- Invariants
- Conditions de sortie
- Règles de perception
- Règles de mise à jour
- Règles de génération
- Règles d'interdiction

Un jeu peut aussi importer les règles d'un jeu déjà défini. Dans ce cas, le nom utilisé est le terme complet représentant le jeu importé afin de lier les nouvelles règles à celles du jeu importé.

DESCR_JEU $\longrightarrow$

import JEU
| entrance = EXPR
| invariant = EXPR
| exit = EXPR
| perception = { REGLE* }
| update = { REGLE* }
| generation = { REGLE* }
| interdiction = { REGLE* }

REGLE $\longrightarrow$

EXPR $\Longrightarrow$ EXPR ['|' EXPR] [# EXPR]

La partie optionnelle de la règle ['|' EXPR] permet de spécifier des conditions d'application de la règle qui ne peuvent pas s'exprimer uniquement avec des états mentaux (par exemple l'extraction d'une sous formule). La partie optionnelle de la règle [# EXPR] permet de modifier le poids de la règle (de génération ou d'interdiction). La valeur par défaut est 1.

Règles de perception

Les règles de perception permettent de traiter les coups joués. Elles sont donc de la forme : Coup $\Longrightarrow$ EXPR Le coup s'unifie avec le dernier coup joué et l'expression EXPR est une expression portant sur les états mentaux. Pour pourvoir travailler sur les séquences de coups, il faut mémoriser la date de chaque coup. Ainsi, un coup est mémorisé par le couple Coup :Date.

Règles de mise à jour

Ces règles gèrent les modifications internes aux états mentaux. Les expressions utilisées ne peuvent donc pas faire apparaître de coups (sauf préfixés par une modalité de croyance par exemple). De plus, la partie droite de la règle doit être une expression sur les états mentaux.

Règles de génération

Ces règles produisent des coups en fonction des états mentaux. La partie gauche est donc une expression quelconque portant sur les états mentaux. La partie droite doit être un coup.

Règles d'interdiction

Ces règles sont symétriques aux règles de génération. La seule différence est que la partie droite est la négation d'un coup. Cette négation signifie que le coup ne peut pas être joué.

Importation de jeux

Lorsque l'on importe un jeu, les conditions du jeu importé ne sont pas prises en compte. Seuls les ensembles de règles importés sont rajoutés à ceux du jeu en cours de définition.

6.2.2 Description des agents

AGENT $\longrightarrow$

agent NOM = { DESCR_AG* }

Un agent est caractérisé par :

- Capacité mémoire[1]

[1] Certaines de ces caractéristiques ne sont pas implémentées (Capacité mémoire, redondances).

- Profondeur maximale des chaînages avant et arrière
- Stockage ou non des redondances
- Base de connaissances initiale
- Règles d'inférence[2]
- Ensemble de jeux
- Jeu initial

DESCR_AG ⟶

memory = INT
| depth = INT
| redondances = (yes|no)
| kb = BASE_CONNAISSANCES
| ir = REGLES_D_INFERENCE
| games = LISTE_DE_JEUX
| start = JEU_INITIAL
| import NOM

Un agent est caractérisé par deux types d'attributs : des valeurs atomiques (taille de la mémoire, profondeur, ...) et des ensembles (base de connaissances, jeux). Ces deux types d'attributs ont des comportements différents lors de l'importation de la description d'un agent dans un autre. Les valeurs atomiques importées ne sont conservées que si elles ne sont pas modifiées dans la description courante. Pour les ensembles, on fait la réunion des ensembles correspondant aux attributs de même nom de l'agent courant et de l'agent importé. L'importation n'est pas limitée à un seul agent. Les agents sont importés dans l'ordre d'écriture des clauses import. Cela signifie que les valeurs atomiques des derniers agents importés écrasent celles des premiers.

6.2.3 Exécution du dialogue

La clause « AGENT1 and AGENT2 are speaking » permet de désigner les agents qui vont dialoguer. Ce sont les agents qui seront compilés en Prolog.

[2] L'implémentation actuelle regroupe la base de connaissances et les règles d'inférence dans le même attribut kb. L'implémentation probabiliste nécessite la séparation de ces deux attributs.

Chapitre 7

Simulation du dialogue

Nous allons voir ici quelques exemples de jeux et donner leur sémantique. La notion de jeux de dialogue n'a pas d'intérêt si les jeux ne sont pas utilisés ensembles. Un jeu seul ne permet pas de produire une structure de dialogue riche. En plus de la définition des jeux, il faut donc penser à la manière dont ils vont être utilisés.

Il y a deux manières de définir des jeux. La première, présentée par (Maudet and Evrard, 1998), se rapproche plus de la théorie des jeux. Un jeu est un ensemble de règles appliquées par les deux agents, les rôles des deux agents dans le jeu pouvant être différents. Ils doivent donc s'accorder sur l'entrée et la sortie du jeu. La deuxième définit le comportement d'un agent. Chaque agent peut être dans son propre jeu. Il serait en fait plus judicieux de parler de rôles plutôt que de jeux. Les agents sont toutefois dans un jeu commun dès le début (le jeu de base). Un jeu serait alors un couple de rôles.

7.1 Hiérarchie des jeux

Nous allons considérer qu'un agent peut être dans deux états différents au cours du dialogue. Le premier état correspond à une situation dans laquelle l'agent n'a pas de but explicite, il doit simplement participer au dialogue, observer un comportement coopératif (ou au moins être capable de rentrer dans une phase de dialogue plus active si la situation se présente). Cet état correspond à un jeu de base dans lequel les agents sont en permanence. Ils sont dans ce jeu au début du dialogue et ce jeu reste au fond de la pile des jeux durant tout le dialogue. On ne peut pas parler de but dans cet état puisque le jeu de base n'est que la représentation de l'activité minimale des agents. Il faut ensuite définir des jeux spécifiques à des objectifs particuliers du dialogue.

Un autre but implicite, comme cela est suggéré par (Maudet and Evrard, 1998), est la maximisation de l'échange d'information. Un tel jeu sera ici défini comme une extension du jeu de base.

La hiérarchie des jeux que nous décrivons ici est statique.

7.1.1 Jeu de base

Le jeu de base définit le comportement minimal de l'agent en attendant de rentrer dans un jeu spécifique doté d'un objectif particulier. Ce jeu doit donc comporter l'axiomatique minimale relative aux actes de langages connus des agents. Cette axiomatique est commune à tous les jeux qui hériteront de ce jeu de base. C'est aussi le seul jeu qui comporte une règle de perception. En effet, pour simplifier l'analyse des coups, on n'utilisera qu'une seule règle de perception qui aura pour effet de mémoriser le coup joué dans l'espace des croyances de l'agent. La véritable analyse sera faite par les règles de mise à jour. Le jeu de base comporte aussi des règles de mise à jour des états mentaux qui définissent une certaine rationalité de l'agent et des règles d'interdiction pour coder quelques maximes de Grice.

Conditions de changement de jeu

Les jeux de base (simple ou avec maximisation de l'échange d'information) ne possèdent pas de conditions d'entrée, d'invariant et de conditions de sortie. En fait, ce que l'on souhaite c'est d'être dans le jeu de base durant tout le dialogue. Pour cela, les conditions d'entrée et l'invariant sont « *true* » et la condition de sortie est « *false* ».

Règle de perception

$$C \implies B_x C | C \neq \$_z$$

Cette règle est très générale. C est une variable qui peut s'unifier avec n'importe quel coup joué. Cette règle permet à l'agent x de mémoriser le coup dans ses croyances. La condition $C \neq \$_z$ évite la mémorisation des coups de clôture du dialogue. Ces coups gêneraient en effet la détection de séquence de coups.

Règles de mise à jour

$$\begin{array}{rcl} B_x ?_y Q & \implies & B_x W_y KIF_y Q \\ B_x !_y P & \implies & B_x B_y P \\ B_x !?_y P & \implies & B_x \neg KIF_y P \end{array}$$

Ces règles interprètent des coups en terme de croyances de l'auditeur sur les états mentaux du locuteur.

Règle de génération

Bien que le jeu de base ne soit pas destiné à produire des coups, il est commode de placer ici la génération de la clôture du dialogue.

$$true \implies \$_x$$

Avec cette règle, l'agent peut proposer la fermeture du dialogue à tout instant. Afin d'éliminer les dialogues vides, cette règle est associée à des règles d'interdiction qui forcent l'agent à rester dans le jeu de base tant qu'il a des intentions ou pense que l'autre en a.

Règles d'interdiction

$$W_x I \vee B_x W_y J \implies \neg \$_x$$

Cette règle est déjà en quelque sorte une condition de la maximisation de l'échange d'information. Il faut cependant la définir dans le jeu de base, pour contrôler la clôture du dialogue.

Les règles d'interdiction du jeu de base permettent aussi de simuler quelques maximes de Grice[1]. En voici quelques unes :

$$\neg B_x P \implies \neg !_x P$$

Ne pas énoncer une proposition pour laquelle on manque de preuve.

$$\neg B_x \neg P \implies \neg !_x P$$

Ne pas énoncer une proposition que l'on croit fausse.

[1] Voir Annexe « Maximes de Grice »

7.1.2 Jeu de base avec maximisation de l'échange d'information

Pour implanter des jeux coopératifs, il faut pouvoir échanger un maximum d'information, afin d'obtenir une base de connaissances contenant la réunion des bases de connaissances initiales des deux agents. Cela est bien sûr impossible si cette réunion est inconsistante. Dans ce cas la maximisation de l'échange se fera en adoptant un comportement coopératif.

Conditions de changement de jeu

Ce jeu étant aussi un jeu de base, il ne possède pas de conditions d'entrée, d'invariant et de conditions de sortie.

Règles de mise à jour

$$B_x B_y P \wedge \neg B_x \neg P \quad \Longrightarrow \quad B_x P$$

Cette règle exprime le transfert des connaissances de la représentation d'un agent vers les croyances de l'agent. On veut un comportement coopératif, il n'y a donc pas de raison de refuser une information si l'on n'a aucune preuve du contraire.

$$W_x KIF_x Q \wedge B_x(P \rightarrow Q) \wedge \neg KIF_x P \quad \Longrightarrow \quad W_x KIF_x P$$

Pour résoudre un problème, l'agent peut s'intéresser aux sous problèmes permettant la résolution complète.

7.1.3 Jeu de question

Le jeu de question permet principalement de générer l'acte de questionner. Ce jeu est activé si l'agent désire savoir quelque chose. Ce jeu hérite du jeu de base.

Condition d'entrée

$$W_x KIF_x Q$$

Le jeu de question est activé lorsque l'agent possède l'état mental correspondant à une intention particulière : celle de savoir si une formule Q est vraie ou non.

Invariant

$$W_x KIF_x Q$$

L'échec de l'invariant doit caractériser l'échec du jeu de question. Il est en effet sans intérêt de rester dans un tel jeu dès lors que l'agent ne veut plus savoir ou connaît la réponse.

Condition de sortie

$$KIF_x Q$$

Le succès du jeu de question est caractérisé par l'ensemble des états dans lesquels l'agent a obtenu par un moyen quelconque une réponse.

Règles de mise à jour

$$KIF_xQ \implies \neg W_xKIF_xQ$$

Dans ce jeu, il est nécessaire de détruire la condition d'entrée avant de sortir[2]. Donc quand la réponse à la question a été trouvée, on nie les conditions d'entrée.

Règles de génération

$$W_xKIF_xQ \implies ?_xQ$$

7.1.4 Jeu de réponse

Le jeu de réponse permet de définir un rôle symétrique à celui du jeu de question. En fait, dans la vision des jeux de (Maudet and Evrard, 1998), le jeu question/réponse serait composé des deux rôles question et réponse.

Condition d'entrée

$$B_xW_yKIF_yQ$$

Le jeu de réponse est activé lorsque l'agent possède une croyance sur un état mental de l'autre agent correspondant à une intention particulière : celle de savoir si une formule Q est vraie ou non.

Invariant

$$B_xW_yKIF_yQ$$

L'échec de l'invariant doit caractériser l'échec du jeu de réponse. Il est en effet sans intérêt de rester dans un tel jeu dès lors que l'autre agent ne veut plus savoir ou connaît la réponse.

Condition de sortie

$$B_xKIF_yQ$$

Le succès du jeu de question est caractérisé par l'ensemble des états dans lesquels l'autre agent a obtenu par un moyen quelconque une réponse.

Règles de génération

$$\begin{aligned} B_xQ &\implies\ !_xQ \\ B_x\neg Q &\implies\ !_x\neg Q \\ \neg KIF_xQ &\implies\ !?_xQ \end{aligned}$$

Ces règles définissent l'ensemble de coups jouables dans un jeu de réponse, en fonction des états mentaux de l'agent.

[2]Cela est fréquent en planification. L'application d'un opérateur détruit souvent les conditions qui ont autorisé l'application de cet opérateur.

7.1.5 Jeu de résolution de conflit

Dans certains cas, la réunion des bases de connaissances des deux agents peut être inconsistance. Si l'inconsistance est détectée, pour être coopératif, il est nécessaire de résoudre le conflit.

Condition d'entrée

$$B_x B_y P \wedge B_x \neg P$$

Un conflit ne peut être détecté que dans la base de connaissances de l'agent et dans la représentation qu'il se fait de l'autre agent. Cela n'est pas une inconsistance logique, mais une supposition de l'inconsistance sur la réunion des croyances.

Invariant

$$B_x B_y P \wedge B_x \neg P$$

Ici encore, l'invariant est identique à la condition d'entrée. En effet, le jeu de résolution de conflit doit se poursuivre tant que le conflit n'a pas disparu.

Condition de sortie

$$B_x B_y \neg P \vee B_x P$$

Le jeu de résolution de conflit réussit si l'un des deux agents parvient à « convaincre » l'autre. Cependant, cette condition peut sembler trop forte. En effet, le conflit peut disparaître sans pour autant que les agents s'entendent sur la valeur de vérité d'une proposition. Dans ce cas, l'un des agents (ou les deux) conclue à l'ignorance totale. La condition de sortie peut donc être affaiblie en :

$$\neg KIF_x B_y P \vee \neg KIF_x P$$

Règles de mise à jour

$$\begin{array}{rcl} B_x B_y \neg P & \Longrightarrow & \neg B_x B_y P \\ B_x P & \Longrightarrow & \neg B_x \neg P \end{array}$$

Ces règles permettent de détruire les conditions d'entrée, une fois que le conflit est résolu.

Règles de génération

$$\begin{array}{rcl} B_x P & \Longrightarrow & !_x P \\ B_x(R \rightarrow P) & \Longrightarrow & !_x(R \rightarrow P) \\ B_x(R \rightarrow P) & \Longrightarrow & !_x R \end{array}$$

Les règles de génération du jeu de résolution de conflit permettent à l'agent dénoncer soit la proposition telle qu'elle est dans les croyances, soit de donner une justification de cette croyance. Une justification peut être une connaissance sous forme d'implication, ou la partie gauche d'une implication.

7.2 Hiérarchie des agents

La notion de hiérarchie pour les agents est moins forte (elle est ici uniquement statique). Elle permet simplement d'importer des descriptions d'agent. Nous allons illustrer cela en créant un agent virtuel de base qui sera raffiné en un agent plus évolué.

7.2.1 Agent de base

L'agent de base doit comporter un ensemble minimal de règles d'inférence afin d'assurer une certaine rationalité. L'agent de base est le premier agent créé, il ne peut donc avoir aucune représentation mentale d'un autre agent (il pourra cependant servir de représentation d'un agent pour un autre agent défini dans la suite).

Règles d'inférence

$$\frac{B_x(P \rightarrow Q) \wedge B_xP}{B_xQ} \quad \text{Axiome (K)}$$

$$\frac{B_xP \vee B_x\neg P}{KIF_xP} \quad \text{Définition de la modalité } KIF$$

$$\frac{B_xP}{\neg B_x\neg P}$$

Base de connaissances

Pour montrer les mécanismes d'importation et de représentation de l'autre agent, nous allons utiliser par exemple la base de connaissances suivante : $\{B_xq, B_x(p \rightarrow q)\}$

Jeux

L'agent de base est initialement dans le jeu de base et possède les jeux « question » et « réponse ».

7.2.2 Agent évolué

L'agent évolué est un agent de base qui possède comme représentation de l'autre agent, la description de l'agent de base. L'agent évolué importe donc la description de l'agent de base et est complété par la description de l'agent fictif qui représente sa connaissance de l'autre ($\tilde{B}_x agent\_de\_base$). En utilisant le formalisme décrit précédemment, l'agent est défini de la manière suivante :

```
Agent agent_evolue =
{
import agent_de_base,
accointance = agent_de_base
}.
```

L'agent obtenu est le suivant :

Règles d'inférence

$$\frac{B_x(P \to Q) \wedge B_x P}{B_x Q} \qquad \frac{B_x P \vee B_x \neg P}{KIF_x P} \qquad \text{Règles importées}$$

$$\frac{B_x B_y(P \to Q) \wedge B_x B_y P}{B_x B_y Q} \qquad \frac{B_x B_y P \vee B_x B_y \neg P}{B_x KIF_y P} \qquad \text{Règles de l'accointance}$$

Base de connaissances

De la même manière, après application de l'opérateur $\tilde{B}_x$, la base de connaissances devient : $\{B_x q, B_x(p \to q), B_x B_y q, B_x B_y(p \to q)\}$.

Jeux

L'agent évolué hérite de l'agent de base, il possède donc les jeux définis dans l'agent de base $\{question, reponse\}$, mais aussi les jeux définis dans l'agent représentant l'accointance calculée par application de l'opérateur $\tilde{B}_x$. Ces jeux sont $\left\{\tilde{B}_x question, \tilde{B}_x reponse\right\}$. L'agent évolué possède deux piles de jeux : une pour la représentation de l'autre agent et une pour lui-même. Ces piles sont initialisées avec respectivement les jeux $\tilde{B}_x jeu\_de\_base$ et jeu_de_base.

Exemple de jeu représentant l'autre agent

L'agent évolué possède une représentation de la base de connaissances et des jeux de l'autre agent. Regardons de plus près la manière dont est traduit un jeu. Par exemple, le jeu de base produit est celui-ci :

- Conditions de changement de jeu :

Conditions d'entrée	$B_x W_y KIF_y Q$
Invariant	$B_x W_y KIF_y Q$
Conditions de sortie	$B_x KIF_y Q$

- Règles du jeu :

Règle de perception	$C \Longrightarrow B_x B_y C \mid C \neq \$_x$
Règles de mise à jour	$\left\{\begin{array}{ccc} B_x B_y ?_x Q & \Longrightarrow & B_x B_y W_x KIF_x Q \\ B_x B_y !_x P & \Longrightarrow & B_x B_y B_x P \\ B_x B_y !?_x P & \Longrightarrow & B_x B_y \neg KIF_x P \\ B_x KIF_y Q & \Longrightarrow & B_x \neg W_y KIF_y Q \end{array}\right.$

Le mécanisme utilisé pour construire ces représentations est décrit plus en détails au chapitre 3, page 12.

Chapitre 8

Choix effectués dans le modèle

Après avoir présenté en détail un modèle pour les jeux de dialogue, nous allons résumer les choix qui ont été effectués dans ce modèle.

8.1 Pas de structure partagée

Dans les modèles de dialogues, deux tendances s'affrontent. La plus ancienne préconise l'utilisation de structures partagées pour permettre le dialogue[1]. Il semble plus intéressant de ne partager aucune structure entre les deux agents afin de pouvoir modéliser un agent en tant qu'entité autonome. Cette deuxième approche se retrouve par exemple dans l'extension de la SDRT au dialogue par (Asher and Lascaride, 1995). L'intérêt de partager des données était d'avoir une représentation de l'autre agent et de pouvoir l'utiliser. Ceci étant indispensable[2] on a rajouté une représentation de l'autre agent directement dans les croyances de l'agent. Cela est en fait plus proche de la réalité. On ne peut pas partager des états mentaux, on peut simplement avoir des croyances sur les états mentaux de l'autre agent.

8.2 Eviter l'omniscience

Dans une application à base de système expert, si l'objectif est de fournir un diagnostic le plus précis et le plus juste possible, l'agent doit être capable de connaître toutes les conséquences logiques de ses connaissances. Cependant, dans notre modèle l'omniscience est à éviter ((Maudet and Evrard, 1998; MacKenzie, 1979)). Un moyen intéressant pour éviter l'omniscience est l'utilisation de coups logiques ((Maudet and Evrard, 1998)) pour contrôler les inférences de l'agent. Dans un souci de simplicité, la solution retenue ici est la limitation en profondeur du moteur d'inférence.

8.3 Hiérarchisation des jeux

Afin de structurer le dialogue, les jeux qui ont été définis précédemment sont soumis à une certaine hiérarchie. Un jeu unique ne permettrait pas la production d'un dialogue bien structuré. On distinguera deux types de hiérarchie : statique et dynamique.

8.3.1 Hiérarchie statique des jeux

La hiérarchie statique des jeux de dialogue peut être vue grossièrement comme une notion d'héritage. Elle intervient au niveau de la déclaration des jeux et se résume en l'importation

[1] Le « black board » utilise un espace de stockage accessible à tous les agents. Voir cours de DEA (Systèmes Multi Agents), (Gleizes and Fade, 1998)

[2] Voir l'exemple du train page 21

de règles de perception, de mise à jour, de génération et d'interdiction. Cela permet de définir simplement des extensions de jeux existants.

8.3.2 Hiérarchie dynamique des jeux

La hiérarchie dynamique est l'organisation de la pile des jeux durant le dialogue. Cette pile code l'imbrication des jeux au moment du dialogue. On retrouvera par exemple un jeu de réponse imbriqué dans un jeu de question, lui même imbriqué dans le jeu de base. Cette imbrication permet de sélectionner en priorité les règles du jeu stocké au sommet de la pile, les jeux du fond de la pile étant les moins prioritaires. Cela est à rapprocher des logiques stratifiées[3].

8.4 Jeu initial

Le comportement de l'agent étant déterminé par le jeu dans lequel il se trouve, il faut prévoir un jeu initial. Ce jeu initial (l'un des jeux de base décrit précédemment) est stocké au fond de la pile des jeux et y reste durant tout le dialogue. Si la pile était initialement vide, l'agent ne posséderait aucune règle de mise à jour des états mentaux et ne pourrait donc pas prendre part au dialogue. Le rôle du jeu initial est donc de fournir cet ensemble minimal de règles capables d'initier le dialogue.

8.5 Jeux de dialogue et théorie des jeux

En théorie des jeux, le jeu définit un ensemble de règles que les joueurs appliquent pour jouer leurs coups. Cela signifie que le jeu est une structure commune aux deux agents. Si l'on veut respecter le principe de séparation des données des deux agents, il faut que ceux-ci puissent se mettre d'accord sur l'entrée et la sortie d'un jeu. Ici on définira plutôt le jeu comme un ensemble de règles propre à un agent, ce qui ne représente que la moitié d'un jeu en théorie des jeux. Un jeu serait en fait un « rôle ». A un instant donné, un couple de rôle constituerait un jeu commun aux deux agents, chaque agent étant responsable du « rôle » qu'il occupe. La structure constituée des deux rôles est inaccessible aux agents. Ils peuvent seulement avoir des croyances sur les jeux qui sont joués pendant le dialogue.

8.6 Paramètres explicites et implicites

Dans ce modèle, on a essayé d'expliciter un maximum de paramètres. Il reste cependant quelques paramètres implicites.

8.6.1 Paramètres explicites

Règles d'inférence

Les règles d'inférence sont totalement décrites dans la description de l'agent. On peut donc utiliser n'importe quel système de règles d'inférence. En fait, les règles de mise à jour fonctionnent aussi comme des règles d'inférence et sont traduites de la même manière. Il a donc été très facile d'expliciter les règles d'inférence. Le moteur d'inférence qu'il a fallu coder se résume donc en l'instanciation des règles.

Caractéristiques de l'accointance

La représentation d'un agent dans les états mentaux peut être explicitement spécifiée dans la description de l'agent. Cependant cette représentation est limitée à un seul agent (le dialogue ne

[3]Voir cours de DEA « Raisonnement Plausible et Incertitude », (Cayrol and Prade, 1998)

peut s'effectuer qu'entre deux agents), mais reste facilement généralisable à condition de pouvoir aussi expliciter le nom (une référence) de l'autre agent.

La description d'une accointance se fait de la même manière que celle d'un agent. On décrit un agent fictif qui sera utilisé via la clause « import » de la description d'un agent réel. Une accointance peut aussi avoir un modèle de l'agent avec lequel elle va dialoguer. Cela signifie qu'un agent peut avoir un modèle du modèle que l'autre agent se fait de lui-même. Ce type de représentation est codé par des croyances imbriquées. Bien que le modèle ne fixe aucune limite dans le niveau d'imbrication des croyances ($B_x B_y B_x \dots p$), il semble qu'au-delà de deux niveaux, l'imbrication soit difficile à appréhender ((Carletta and Mellish, 1995)).

Limitation de l'omniscience en profondeur

Nous avons choisi d'éviter l'omniscience. Pour cela, nous limitons la profondeur du moteur d'inférence des agents. La profondeur peut être spécifiée et différente pour chaque agent. Nous verrons plus loin de problème de la limitation de l'omniscience « en largeur ».

But des jeux spécifiques

Le but des jeux peut être assimilé à leur condition de sortie. Donc dans une certaine mesure, le but d'un jeu particulier (différent d'un jeu de base) est paramétrable[4]. Cela doit en fait être nuancé, comme on le verra dans les paramètres implicites (But des jeux).

8.6.2 Paramètres implicites

But des jeux

Bien que pouvant être en partie explicité, le but d'un jeu est en général d'en sortir (c'est à dire d'en satisfaire les conditions de sortie). C'est encore différent pour les jeux initiaux (les jeux de bases présentés précédemment). Ces derniers fournissent des règles permettant de décrire le fonctionnement minimal de l'agent avant de réellement dialoguer. Ils sont là par exemple pour mettre les agents dans un certain état de coopération.

Nom de l'accointance

Nous avons vu que la description de l'accointance est totalement explicite. Son nom (une référence à l'interlocuteur) reste cependant implicite. On le précise au moment de la déclaration des agents destinés à dialoguer (clause « are speaking »), mais ensuite les agents n'énoncent pas leurs actes pour un agent particulier. En effet, les coups sont uniquement indicés par le nom du locuteur et non de l'auditeur visé (les agents parlent « dans le vide », mais savent de qui proviennent les coups). Les coups sont diffusés et peuvent être traités en fonction de l'émetteur. Cela n'est pas gênant tant qu'il n'y a que deux agents. Au-delà, il sera nécessaire d'expliciter les noms des locuteurs et des auditeurs.

Mécanisme de maintien de la cohérence

Le mécanisme de maintien de la cohérence est fixé par l'implémentation. En aucun cas l'agent ne peut agir sur le maintien de la cohérence. Un moyen d'expliciter ce mécanisme serait d'utiliser la notion de coups logiques[5]afin d'orienter la révision des bases de connaissances des agents selon des règles à préciser.

[4]Le but du jeu de question est bien d'obtenir une réponse : KIF_xQ

[5]Voir (Maudet and Evrard, 1998). Un coup logique est une modification de la base de connaissances activée par des règles explicites propres à l'agent. Il est généré de la même manière qu'un coup classique. La seule différence est que la porté d'un tel coup est interne à l'agent qui garde la main dans le dialogue en cours.

Limitation de l'omniscience « en largeur »

Nous avons vu qu'il est possible de limiter la profondeur du système d'inférence. Il serait intéressant de pouvoir aussi en limiter la largeur, c'est à dire de pouvoir appliquer une heuristique pour limiter les inférences à effectuer. Ici aussi, la notion de coups logiques serait un moyen élégant de contourner le problème, en explicitant les inférences que l'agent va ou ne va pas réaliser. Remarquons aussi que cela permet de limiter la profondeur.

Stratégie d'utilisation des jeux

L'entrée dans un jeu, sa sortie et l'utilisation de ses règles ne peuvent pas être décidées par l'agent. En fait les jeux en cours ne sont pas mémorisés dans la base de connaissances de l'agent. L'agent « subit » les règles du jeu dans lequel il se trouve.

Mode de fonctionnement des règles

Nous verrons dans les extensions que l'on peut imaginer plusieurs modes de fonctionnement des règles dans un jeu. Dans le modèle actuel, une règle n'est déclenchée que si ses prémisses sont vérifiées dans la base de connaissances de l'agent et l'agent n'a aucun contrôle sur leur déclenchement.

Chapitre 9

Exemples de dialogue

Dans les exemples suivants, les formules logiques sont écrites en utilisant la syntaxe du programme Prolog. Les modalités sont représentées par les opérateurs infixes, le premier argument étant le nom de l'agent. L'opérateur unaire « - » représente la négation. Les majuscules sont des variables.

9.1 Premier exemple

Nous allons voir un premier exemple qui met en évidence l'emboîtement des jeux. Il s'agit de résoudre un problème simple. Le premier agent est caractérisé par $\{B_{a_1}p, W_{a_1}KIF_{a_1}r\}$ et le second par $\{B_{a_2}(p \to q), B_{a_2}(q \to r)\}$. Le but est donc de démontrer r.

Cet exemple utilise les jeux de base, question et réponse décrits précédemment.

```
/************************************************************/
/*                                                          */
/* Exemple de jeux                                          */
/*                                                          */
/************************************************************/

game jeu_de_base =
{
    perception =
    {
        C => x b (C:t) | C\=($ _)
    },

    update =
    {
        x b (y ? Q:_) => x b y w y kif Q,
        x b (y ! P:_) => x b y b P,
        x b (y ! P:_) and x w x kif P => x b P,

        x b P => -(x b -P),

        x b y b P and -(x b -P) => x b P
    },

    generation =
    {
        -(x w _) and -(x b y w _) => $ x # 0.01
```

```
    },

    interdiction =
    {
        x b (C:T) => -C | T<t # 100
    }
}.

game question(Q) =
{
    import jeu_de_base,

    entrance = x w x kif Q,
    invariant = x w x kif Q or -(x b y b x kif Q),
    exit = x kif Q and x b y b x kif Q,

    generation =
    {
        x w x kif Q => x ? Q # 10,
        x b Q => x ! Q # 2,
        x b -Q => x ! -Q # 2
    },

    update =
    {
        x kif Q => -(x w x kif Q),
        x b (P->Q) and -(x kif P) => x w x kif P
    }
}.

game reponse(Q) =
{
    import jeu_de_base,

    entrance = x b y w y kif Q,
    invariant = x b y w y kif Q,
    exit = x b y kif Q,

    generation =
    {
        x b Q => x ! Q # 10,
        x b (P->Q) => x ! (P->Q) # 5,
        x b (P->Q) and x b P => x ! P # 3,
        -(x b Q) => x '!?' Q
    },

    update =
    {
        x b (P->Q) and -(x kif P) => x w x kif P,
        x b y kif Q => -(x b y w y kif Q),

        x b (x ! P:_) and x b y w y kif P => x b y b P

    }
```

```
}.

agent agent_de_base(PROF) =
{

    start = jeu_de_base,

    games = { question(_), reponse(_) },

    depth = PROF,

    kb =
    {
        nonvar(Q) | x b (P->Q) and x b P then x b Q,
        nonvar(P) | x b (P->Q) and x b -Q then x b -P,

        nonvar(P) | (x b P) or (x b -P) then x kif P
    }
}.

agent a1 =
{
    import agent_de_base(3),

    accointance = agent_de_base(3),

    kb =
    {
        x w x kif r,
        x b p
    }
}.

agent a2 =
{
    import agent_de_base(3),

    accointance = agent_de_base(3),

    kb=
    {
        x b (p->q),
        x b (q->r)
    }
}.

a1 and a2 are speaking.
ß
```

Les états mentaux marqués en gras sont ceux qui nous intéressent ici. A chaque étape, on ne reprend pas la totalité des bases de connaissances, mais uniquement les états qui changent.

Remarquons tout d'abord qu'après chaque coup, les agents croient en la réussite du coup (*).

La première ligne du tableau contient les bases de connaissances des agents. Dans cet état, l'agent a_1 rentre dans le jeu *question*(r) (1). L'agent a_2 perçoit donc que a_1 est dans un jeu

$question(r)$ (2) (3), a_2 rentre alors dans $reponse(r)$ (4). De même, a_1, en utilisant sa représentation de a_2 pense que a_2 rentre dans un jeu de $reponse(r)$ (5). L'un des coups permis dans cette situation est par exemple une justification de la réponse (6).

La simulation de a_1 dans les croyances de a_2 permet à a_2 de supposer que a_1 accepte cette justification (7) et rentre dans un jeu de $question(q)$ (8), il rentre donc dans le jeu de $reponse(q)$ (9). C'est effectivement ce qu'il ce passe pour a_1 (10) qui pose la question (11). a_2 qui était déjà dans le jeu $reponse(q)$ donne une justification (12) et refait les mêmes hypothèses que précédemment sur a_1 (13). Mais cette fois, a_1 réagit différemment puisqu'il croit p. Après s'être approprié $p \rightarrow q$ (14), il croit q et r (15) et ne se pose plus de question (16). a_1 donne la réponse q (17). A partir de là les deux agents ont toutes les réponses qu'ils attendaient (18) et le dialogue se termine.

a1	Coups	a2
a1 b p - (a1 b-p) **a1 w a1 kif r (1)** a1 kif p		**a2 b (q->r)** - (a2 b- (q->r)) **a2 b (p->q)** - (a2 b- (p->q)) a2 kif (p->q) a2 kif (q->r)
a1 rentre dans le jeu question(r) (1)	**a1 ?r (1)**	
a1 b a2 b (a1 ?r :0) (*) **a1 b a2 b a1 w a1 kif r (*)** - (a1 b a2 b- (a1 w a1 kif r)) a1 b a2 kif a1 w a1 kif r - (a1 b a2 b- (a1 ?r :0)) a1 b a2 kif (a1 ?r :0) a1 b (a1 ?r :0) a1 b a1 w a1 kif r - (a1 b- (a1 w a1 kif r)) - (a1 b- (a1 ?r :0))		a2 b a1 b (a1 ?r :0) - (a2 b a1 b- (a1 ?r :0)) a2 b a1 kif (a1 ?r :0) **a2 b (a1 ?r :0) (*)** **a2 b a1 w a1 kif r (*) (2)** - (a2 b- (a1 w a1 kif r)) - (a2 b- (a1 ?r :0))
agent_de_base(3) rentre dans le jeu a1 b reponse(r) (5)	**a2 ! (q->r) (6)**	**agent_de_base(3) rentre dans le jeu a2 b question(r) (3)** **a2 rentre dans le jeu reponse(r) (4)**
a1 b a2 b (a2 ! (q->r) :1) - (a1 b a2 b- (a2 ! (q->r) :1)) a1 b a2 kif (a2 ! (q->r) :1) **a1 b (a2 ! (q->r) :1) (*)** **a1 b a2 b (q->r) (*)** - (a1 b- (a2 b (q->r))) **a1 b (q->r) (10)** **a1 w a1 kif q (10)** - (a1 b- (q->r))		**a2 b a1 b (a2 ! (q->r) :1) (*)** **a2 b a1 b a2 b (q->r) (*)** - (a2 b a1 b- (a2 b (q->r))) **a2 b a1 b (q->r) (7)** **a2 b a1 w a1 kif q (7)** - (a2 b a1 b- (q->r)) a2 b a1 kif (q->r) a2 b a1 kif a2 b (q->r) - (a2 b a1 b- (a2 ! (q->r) :1))

...

...

a1 b a2 kif (q->r) - (a1 b- (a2 kif (q->r))) - (a1 b- (a2 ! (q->r) :1))		a2 b a1 kif (a2 ! (q->r) :1) a2 b (a2 ! (q->r) :1) a2 b a2 b (q->r) - (a2 b- (a2 b (q->r))) a2 b a2 kif (q->r) - (a2 b- (a2 kif (q->r))) - (a2 b- (a2 ! (q->r) :1))
a1 rentre dans le jeu question(q) (10)	**a1 ?q (11)**	**agent_de_base(3) rentre dans le jeu a2 b question(q) (8)** **a2 rentre dans le jeu reponse(q) (9)**
a1 b a2 b (a1 ?q :2) (*) **a1 b a2 b a1 w a1 kif q (*)** - (a1 b a2 b- (a1 w a1 kif q)) a1 b a2 kif a1 w a1 kif q - (a1 b a2 b- (a1 ?q :2)) a1 b a2 kif (a1 ?q :2) a1 b (a1 ?q :2) a1 b a1 w a1 kif q - (a1 b- (a1 w a1 kif q)) - (a1 b- (a1 ?q :2))		a2 b a1 b (a1 ?q :2) - (a2 b a1 b- (a1 ?q :2)) a2 b a1 kif (a1 ?q :2) **a2 b (a1 ?q :2) (*)** - (a2 b- (a1 ?q :2))
agent_de_base(3) rentre dans le jeu a1 b reponse(q)	**a2 ! (p->q) (12)**	
a1 b a2 b (a2 ! (p->q) :3) - (a1 b a2 b- (a2 ! (p->q) :3)) a1 b a2 kif (a2 ! (p->q) :3) **a1 b (a2 ! (p->q) :3) (*)** **a1 b a2 b (p->q) (*)** - (a1 b- (a2 b (p->q))) **a1 b (p->q) (14)** - (a1 b- (p->q))		**a2 b a1 b (a2 ! (p->q) :3) (*)** **a2 b a1 b a2 b (p->q) (*)** **- (a2 b a1 b- (a2 b (p->q)))** **a2 b a1 b (p->q) (13)** **a2 b a1 w a1 kif p (13)** - (a2 b a1 b- (p->q)) a2 b a1 kif (p->q) a2 b a1 kif a2 b (p->q)

...

. . .

a1 b q (15) - (a1 b-q) **a1 b r (15)** - (a1 b-r) **- (a1 w a1 kif r) (16)** **- (a1 w a1 kif q) (16)** a1 b a2 kif (p->q) - (a1 b- (a2 kif (p->q))) - (a1 b- (a2 ! (p->q) :3))		- (a2 b a1 b- (a2 ! (p->q) :3)) a2 b a1 kif (a2 ! (p->q) :3) a2 b (a2 ! (p->q) :3) a2 b a2 b (p->q) - (a2 b- (a2 b (p->q))) a2 b a2 kif (p->q) - (a2 b- (a2 kif (p->q))) - (a2 b- (a2 ! (p->q) :3))
	a1 !q (17)	**agent_de_base(3) rentre dans le jeu a2 b question(p)** **a2 rentre dans le jeu reponse(p)**
a1 b a2 b (a1 !q :4) (*) **a1 b a2 b a1 b q (*)** - (a1 b a2 b- (a1 b q)) **a1 b a2 b q (18)** - (a1 b a2 b-q) **a1 b a2 b r (18)** - (a1 b a2 b-r) a1 b a2 kif r a1 b a2 kif q a1 b a2 kif a1 b q - (a1 b a2 b- (a1 !q :4)) a1 b a2 kif (a1 !q :4) a1 b (a1 !q :4) a1 b a1 b q - (a1 b- (a1 b q)) a1 b a1 kif q - (a1 b- (a1 kif q)) - (a1 b- (a1 !q :4))		a2 b a1 b (a1 !q :4) - (a2 b a1 b- (a1 !q :4)) a2 b a1 kif (a1 !q :4) **a2 b (a1 !q :4) (*)** **a2 b a1 b q (*)** - (a2 b- (a1 b q)) **a2 b q (18)** - (a2 b-q) **a2 b r (18)** - (a2 b-r) **a2 b a1 b r (18)** **- (a2 b- (a1 b r))** a2 b a1 kif r - (a2 b- (a1 kif r)) **- (a2 b a1 w a1 kif r) (18)** a2 b a1 kif q - (a2 b- (a1 kif q)) **- (a2 b a1 w a1 kif q) (18)** - (a2 b- (a1 !q :4))
	a2' ! ?'p	

. . .

	...	
a1 b a2 b (a2'!?'p :5) - (a1 b a2 b- (a2'!?'p :5)) a1 b a2 kif (a2'!?'p :5) a1 b (a2'!?'p :5) - (a1 b- (a2'!?'p :5))		a2 b a1 b (a2'!?'p :5) - (a2 b a1 b- (a2'!?'p :5)) a2 b a1 kif (a2'!?'p :5) a2 b (a2'!?'p :5) - (a2 b- (a2'!?'p :5))
	$a1	
	$a2	

9.2 Interrogation d'un passant par un policier

L'exemple suivant permet de simuler le dialogue entre un policier et un passant :

Policier	:	Quel est votre nom ?	$?_x q$
Passant	:	Cela ne vous regarde pas !	$!_y \neg A_x KIF_x q$
Policier	:	Je suis policier !	$!_x p(x)$
Passant	:	Je m'appelle Dupond.	$!_y q$

Dans cet exemple le dialogue est instancié de la manière suivante :

x	$=$	Policier
y	$=$	Passant
q	$=$	Un renseignement
$A_z p$	$=$	« L'agent z est autorisé à p »
$p(z)$	$=$	« L'agent z est policier »

Ici, le jeu réponse est différent. Pour obtenir une réponse, il faut être autorisé à poser la question. Le jeu réponse n'est donc activable qu'à cette condition. Dans le cas où l'autorisation n'est pas prouvée, c'est un jeu mefiance qui sera activé. Dans ce jeu, l'agent ne fait qu'exprimer sa méfiance mais ne donne aucune information. Nous avons aussi besoin d'un jeu conflit qui est activé en cas de conflit entre les croyances d'un agent et des croyances qu'il a de l'autre agent.

```
/**********************************************************/
/*                                                        */
/* Exemple de jeux                                        */
/*                                                        */
/**********************************************************/

game jeu_de_base =
{
    perception =
    {
        C => x b (C:t) | C\=($ _)
    },

    update =
    {
        x b (y ? Q:_) => x b y w y kif Q,
        x b (y ! P:_) => x b y b P,
        x b (y ! P:_) and x w x kif P => x b P,

        x b P => -(x b -P),

        x b (y ! P:_) => x b y b x a x kif P
    },

    generation =
    {
        -(x w _) and -(x b y w _) => $ x # 0.01
    },

    interdiction =
    {
        x b (C:T) => -C | T<t # 100
    }
}.

game question(Q) =
```

```
{
    import jeu_de_base,

    entrance = x w x kif Q,
    invariant = x w x kif Q,
    exit = x kif Q,

    generation =
    {
        x w x kif Q => x ? Q # 10
    },

    update =
    {
        x kif Q => -(x w x kif Q)
    }
}.

game reponse(Q) =
{
    import jeu_de_base,

    entrance = x b y w y kif Q and x b y a y kif Q,
    invariant = x b y w y kif Q,
    exit = x b y kif Q,

    generation =
    {
        x b Q => x ! Q # 10,
        -(x b Q) => x '!?' Q
    },

    update =
    {
        x b (P->Q) and -(x kif P) => x w x kif P,
        x b y kif Q => -(x b y w y kif Q),

        x b (x ! P:_) and x b y w y kif P => x b y b P

    }
}.

game conflit(Q) =
{
    import jeu_de_base,

    entrance = x b y b -Q and x b Q,
    invariant = x b y b -Q and x b Q,
    exit = x b y b Q or x b -Q,

    generation =
    {
        x b (P->Q) => x ! P # 100,
        x b (P->Q) => x ! (P->Q),
```

```
        x b Q => x ! Q
    },

    update =
    {
        x b y b Q => -(x b y b -Q),
        x b -Q => -(x b Q)
    }
}.

game mefiance(Q) =
{
    import jeu_de_base,

    entrance = x b y w Q and -(x b y a Q),
    invariant = x b y w Q and -(x b y a Q),
    exit = x b y a Q or -(x b y w Q),

    update =
    {
        x b (y ! P:_) and x b (P -> (y a Q)) => x b P,
        -(x b y a Q) => x w x kif y a Q
    },

    generation =
    {
        -(x b y a Q) => x ! -(y a Q)
    }
}.

agent agent_de_base(PROF) =
{

    start = jeu_de_base,

    games = { question(_), reponse(_) },

    depth = PROF,

    kb =
    {
        nonvar(Q) | x b (P->Q) and x b P then x b Q,
        nonvar(P) | x b (P->Q) and x b -Q then x b -P,

        x b (p(X)->(X a Q)),

        nonvar(P) | (x b P) or (x b -P) then x kif P
    }
}.

agent agent_evolue(PROF) =
{
    import agent_de_base(PROF),
```

```
    kb =
    {
        x b p(a1)
    }
}.

agent a1 =
{
    import agent_de_base(1),

    accointance = agent_de_base(1),

    games = { conflit(_) },

    kb =
    {
        x w x kif q,
        x b p(x)
    }
}.

agent a2 =
{
    import agent_de_base(1),

    games = { mefiance(_) },

    kb=
    {
        x b q
    }
}.

a1 and a2 are speaking.
```

ß

Les états mentaux marqués en gras sont ceux qui nous intéressent ici. A chaque étape, on ne reprend pas la totalité des bases de connaissances, mais uniquement les états qui changent.

Remarquons tout d'abord qu'après chaque coup, les agents croient en la réussite du coup (*).

La première ligne du tableau contient les bases de connaissances des agents. Dans cet état, l'agent a_1 rentre dans le jeu $question(q)$ (1). L'agent a_2 perçoit bien la question mais ne sait pas si a_1 est autorisé à savoir si q donc il ne peut que rentrer dans un jeu de $mefiance(KIF_{a_1}q)$ (2). L'agent a_1 croit alors que a_2 ne sait pas qu'il est policier (3) et doit donc jouer un jeu $correction(A_{a_1}KIF_{a_1}q)$ (4). Un moyen de justifier cela, est de dire qu'il est policier (5). Donc a_2 croit que a_1 est policier, donc autorisé (6). Il sort du jeu méfiance (7) et rentre cette fois dans le jeu $reponse(q)$ (8) (9).

a1	**Coups**	**a2**
a1 b A b (p(B)->B a C) - (a1 b- (A b (p(B)->B a C))) a1 b A kif (p(B)->B a C) - (a1 b- (A kif (p(B)->B a C))) **a1 b p(a1)** - (a1 b-p(a1)) **a1 w a1 kif q (1)** **a1 b (p(A)->A a B)** - (a1 b- (p(A)->A a B)) **a1 b a1 a A** - (a1 b- (a1 a A)) a1 kif (p(A)->A a B) a1 kif a1 a A - (a1 b- (a1 a A)) a1 kif p(a1) - (a1 b-p(a1)) a1 kif A kif (p(B)->B a C) - (a1 b- (A kif (p(B)->B a C))) a1 b A kif (p(B)->B a C) a1 kif A b (p(B)->B a C) - (a1 b- (A b (p(B)->B a C)))		**a2 b q** - (a2 b-q) **a2 b (p(A)->A a B)** - (a2 b- (p(A)->A a B)) a2 kif (p(A)->A a B) - (a2 b- (p(A)->A a B)) a2 kif q
a1 rentre dans le jeu question(q) (1)	**a1 ?q**	
a1 b a2 b (a1 ?q :0) (*) **a1 b a2 b a1 w a1 kif q (*)** - (a1 b a2 b- (a1 w a1 kif q)) a1 b a2 kif a1 w a1 kif q - (a1 b a2 b- (a1 ?q :0)) a1 b a2 kif (a1 ?q :0) a1 b (a1 ?q :0) a1 b a1 w a1 kif q - (a1 b- (a1 w a1 kif q)) - (a1 b- (a1 ?q :0))		**a2 b (a1 ?q :0) (*)** **a2 b a1 w a1 kif q (*)** - (a2 b- (a1 w a1 kif q)) - (a2 b- (a1 ?q :0))
	. . .	

. . .

	a2 !- (a1 a a1 kif q)	**a2 rentre dans le jeu mefiance(a1 kif q) (2)**
a1 b a2 b (a2 !- (a1 a a1 kif q) :1) - (a1 b a2 b- (a2 !- (a1 a a1 kif q) :1)) a1 b a2 kif (a2 !- (a1 a a1 kif q) :1) **a1 b (a2 !- (a1 a a1 kif q) :1) (*)** **a1 b a2 b- (a1 a a1 kif q) (*)** - (a1 b- (a2 b- (a1 a a1 kif q))) a1 b a2 kif- (a1 a a1 kif q) - (a1 b- (a2 kif- (a1 a a1 kif q))) a1 b a2 kif a1 a a1 kif q - (a1 b- (a2 kif a1 a a1 kif q)) **a1 b a2 b-p(a1) (3)** - (a1 b- (a2 b-p(a1))) a1 b a2 kif-p(a1) - (a1 b- (a2 kif-p(a1))) a1 b a2 kif p(a1) - (a1 b- (a2 kif p(a1))) - (a1 b- (a2 !- (a1 a a1 kif q) :1)) a1 b a2 b a1 a a1 kif- (a1 a a1 kif q) - (a1 b- (a2 b a1 a a1 kif- (a1 a a1 kif q))) a1 b a2 kif a1 a a1 kif- (a1 a a1 kif q) - (a1 b- (a2 kif a1 a a1 kif- (a1 a a1 kif q)))		a2 b (a2 !- (a1 a a1 kif q) :1) a2 b a2 b- (a1 a a1 kif q) - (a2 b- (a2 b- (a1 a a1 kif q))) a2 b a2 kif- (a1 a a1 kif q) - (a2 b- (a2 kif- (a1 a a1 kif q))) a2 b a2 kif a1 a a1 kif q - (a2 b- (a2 kif a1 a a1 kif q)) - (a2 b- (a2 !- (a1 a a1 kif q) :1)) a2 b a2 b a2 a a2 kif- (a1 a a1 kif q) - (a2 b- (a2 b a2 a a2 kif- (a1 a a1 kif q))) a2 b a2 kif a2 a a2 kif- (a1 a a1 kif q) - (a2 b- (a2 kif a2 a a2 kif- (a1 a a1 kif q)))
a1 rentre dans le jeu correction(a1 a a1 kif q) (4)	**a1 !p(a1) (5)**	
a1 b a2 b (a1 !p(a1) :2) (*) **a1 b a2 b a1 b p(a1) (*)** - (a1 b a2 b- (a1 b p(a1))) a1 b a2 kif a1 b p(a1) a1 b a2 b a1 kif p(a1) - (a1 b a2 b- (a1 kif p(a1))) a1 b a2 kif a1 kif p(a1)		**a2 b (a1 !p(a1) :2) (*)** **a2 b p(a1) (*)** - (a2 b-p(a1)) **a2 b a1 a A (6)** - (a2 b- (a1 a A)) **a2 b a1 b p(a1)** - (a2 b- (a1 b p(a1)))

. . .

...

- (a1 b a2 b- (a1 !p(a1) :2)) a1 b a2 b a1 b a2 a a2 kif p(a1) - (a1 b a2 b- (a1 b a2 a a2 kif p(a1))) a1 b a2 kif a1 b a2 a a2 kif p(a1) a1 b a2 b a1 kif a2 a a2 kif p(a1) - (a1 b a2 b- (a1 kif a2 a a2 kif p(a1))) a1 b a2 kif a1 kif a2 a a2 kif p(a1) a1 b a2 kif (a1 !p(a1) :2) a1 b (a1 !p(a1) :2) a1 b a1 b p(a1) - (a1 b- (a1 b p(a1))) a1 b a1 kif p(a1) - (a1 b- (a1 kif p(a1))) a1 b a1 b a1 a A - (a1 b a1 b- (a1 a a1 kif q)) - (a1 b- (a1 b a1 a A)) a1 b a1 kif a1 a A - (a1 b- (a1 kif a1 a A)) - (a1 b- (a1 !p(a1) :2))		a2 b a1 kif p(a1) - (a2 b- (a1 kif p(a1))) - (a2 b- (a1 !p(a1) :2)) a2 b a1 b a2 a a2 kif p(a1) - (a2 b- (a1 b a2 a a2 kif p(a1))) a2 b a1 kif a2 a a2 kif p(a1) - (a2 b- (a1 kif a2 a a2 kif p(a1)))
a1 rentre dans le jeu correction(p(a1))	**a2 !q (9)**	**a2 sort du jeu mefiance(a1 kif q) (7)** **a2 rentre dans le jeu reponse(q) (8)**
a1 b a2 b (a2 !q :3) - (a1 b a2 b- (a2 !q :3)) a1 b a2 kif (a2 !q :3) **a1 b (a2 !q :3) (*)** **a1 b a2 b q (*)** - (a1 b- (a2 b q)) a1 b a2 kif q - (a1 b- (a2 kif q)) **a1 b q** - (a1 b-q)		**a2 b (a2 !q :3) (*)** **a2 b a1 b q (*)** - (a2 b- (a1 b q)) a2 b a1 kif q - (a2 b a1 w a1 kif q) - (a2 b- (a1 kif q)) a2 b a2 b q - (a2 b- (a2 b q)) a2 b a2 kif q - (a2 b a2 w a2 kif q)

...

	...	
- (a1 w a1 kif q) - (a1 b- (a2 !q :3)) a1 b a2 b a1 a a1 kif q - (a1 b- (a2 b a1 a a1 kif q)) - (a1 b a2 b- (a1 a a1 kif q))		- (a2 b- (a2 kif q)) - (a2 b- (a2 !q :3)) a2 b a2 b a2 a a2 kif q - (a2 b- (a2 b a2 a a2 kif q)) a2 b a2 kif a2 a a2 kif q - (a2 b- (a2 kif a2 a a2 kif q))
agent_de_base(1) rentre dans le jeu a1 b reponse(q)	**$a1**	**a2 sort du jeu reponse(q)**
	$a2	

Chapitre 10

Extensions

Ici on ne fait qu'évoquer les extensions, certaines sont présentées en détail en annexe.

10.1 Mode de fonctionnement des règles

Dans le modèle présenté, le mode de fonctionnement des règles est classique. La partie gauche d'une règle doit être démontrable par l'agent pour pouvoir être activée. Lorsqu'une règle est activée, la partie droite est introduite dans la base de connaissances de l'agent. On peut imaginer deux autres types de fonctionnement pour ces règles :

- déclenchement sur prémisses partiellement vérifiées,
- déclenchement par satisfaction des conditions non vérifiées.

Dans la suite, m_c, m_{ppv} et m_{scnv} désignent respectivement les modes de déclenchement classique, sur prémisses partiellement vérifiées et par satisfaction des conditions non vérifiées.

Le déclenchement sur prémisses partiellement vérifiées permet de déclencher une règle non déclenchable par le mécanisme classique. Lorsqu'une partie des conditions déclenchantes n'est pas satisfaite, elle est supposée vraie si l'on n'a pas la preuve du contraire. La règle est alors déclenchée.

Le déclenchement par satisfaction des conditions non vérifiées se rapproche de la notion de planification. Comme précédemment, lorsque la partie gauche d'une règle n'est que partiellement vérifiée, on va quand même essayer de déclencher la règle. Dans ce mode de fonctionnement, l'agent aura pour but de satisfaire les conditions non encore satisfaites avant de pouvoir déclencher la règle.

Il est intéressant de remarquer que les règles de type m_c peuvent simuler le fonctionnement des règles m_{ppv} et m_{scvn}. Prenons par exemple une règle classique $A \wedge B \Longrightarrow C$. Pour faire fonctionner cette règle dans le mode m_{ppv}, il suffit de la traduire par les deux règles classiques suivantes :

$$\left\{ \begin{array}{l} A \wedge B \Longrightarrow C \\ A \Longrightarrow S_x B \wedge C \\ B \Longrightarrow S_x A \wedge C \end{array} \right.$$

L'expression modale $S_x A$ signifie que l'agent x suppose la formule A vraie. Cette réécriture fait apparaître les connaissances qui sont supposées pour pouvoir activer la règle. De même pour utiliser cette règle avec le mode m_{scnv}, il faut faire apparaître les buts qui expriment la satisfaction des conditions non encore vérifiées. La règle est donc traduite de la manière suivante :

$$\left\{ \begin{array}{l} A \wedge B \Longrightarrow C \\ A \Longrightarrow W_x B \\ B \Longrightarrow W_x A \end{array} \right.$$

Pour que cette traduction marche correctement, il faut que l'apparition de buts force l'agent à les satisfaire (par application d'autres règles ou changement de jeu par exemple).

10.2 Système de maintien de la cohérence en logique probabiliste

Le problème du maintien de la cohérence dans la base de connaissances de l'agent se pose rapidement. Dans le cadre de cette étude, les agents doivent être capables de tenir un raisonnement non monotone tout en conservant une certaine consistance dans la base (consistance faible ou locale). Le PTMS (Probabilistic Truth Maintenance System) permet de construire de manière incrémentale une base de connaissances. Les formules probabilistes utilisées par le système sont des formules associées à une probabilité. On dira que la base K infère la formule ϕ avec une probabilité p si $\langle \phi, p_0 \rangle \in K$, $p_0 \geq p$. Ce système respecte les trois axiomes principaux de la mise à jour révisionniste d'une base de connaissances. Soit K' la base de connaissances obtenue après mise à jour de la base K par la formule probabiliste ϕ. Alors K' infère ϕ, la « distance » entre K et K' est minimale et K' reste localement consistante « autour » de ϕ. La distance entre deux bases de connaissances est dans ce cas une distance de Hamming (on compte les états dont les probabilités sont différentes).

10.3 Explicitation du fonctionnement des jeux et des agents

Nous avons vu que le fonctionnement des jeux et des agents n'est pas complètement explicite. Il faut permettre à l'agent de contrôler ses inférences. En étendant la notion de coups logiques à celle de jeux logiques (voir plus loin) et en rendant « conscients » les agents des jeux qu'ils utilisent[1] les agents pourront utiliser les jeux pour dialoguer plutôt que d'en subir inconsciemment les règles.

10.4 Jeux logiques

Si l'on reprend la notion de coups logiques pour contrôler les inférences, les changements de jeu (on peut considérer que le changement de jeu est un coup logique), on voit apparaître une notion proche du jeu de dialogue mais destinée à la production d'états mentaux (coup logique) et non de coup de dialogue.

Par exemple, $B_x P \Longrightarrow !_x P$ ou $B_x P \Longrightarrow W_x !_x P$ feraient partie d'un jeu de dialogue et $B_x B_y P \wedge \neg B_x \neg P \Longrightarrow B_x P$ d'un jeu logique.

10.5 Planification du dialogue

Le dernier exemple de dialogue montre que le dialogue généré n'est pas forcément le plus efficace (le plus court). Si les agents pouvaient planifier leurs coups, ils devraient pouvoir se rendre compte plus tôt que certains choix risquent d'être plus coûteux dans la suite du dialogue.

La planification est donc un outil complémentaire aux jeux de dialogue. La planification doit permettre de choisir les bons jeux et les jeux doivent utiliser la planification pour fonctionner efficacement.

Un exemple de planificateur est donné plus loin.

10.6 Langage naturel

Pour interfacer un agent artificiel tel qu'il a été décrit ici avec un utilisateur humain, il faudra envisager d'échanger les coups en langage naturel et non formel. Pour cela, il existe des grammaires logiques qui permettent de passer directement d'un énoncé textuel à une représentation sémantique dans un langage formel (la logique des prédicats du premier ordre par exemple). On

[1] Dans le modèle actuel, les agents subissent les changements de jeu et leurs règles.

trouvera dans les annexes des implémentations de grammaires lexicales fonctionnelles et catégorielles (Lambek/Gentzen). Un autre modèle à la mode pour l'analyse du langage naturel est la SDRT (voir (Asher, 1993)) (Segmented Discourse Representation Theory). Par rapport aux grammaires formelles, ce modèle présente l'avantage de tenir compte de la connaissance du monde pendant la phase d'analyse du texte.

Chapitre 11

Probabilistic Truth Maintenance System

Ce chapitre présente une extension du JMS (Justification Maintenance System) basée sur une logique probabiliste : le PTMS (Probabilistic Truth Maintenance System). Le système est capable de tenir un raisonnement non monotone et de maintenir la consistance (localement) dans une base de connaissances.

11.1 Introduction

Dans les systèmes experts utilisant une logique monotone, un système de règles d'inférence classique est suffisant. Dès que l'on veut introduire une logique non monotone et le maintien de la cohérence, le problème se complique. Des systèmes représentant les connaissances dans un réseau (JMS, TMS, ...) sont alors très utiles. Ils sont cependant limités à la logique propositionnelle. Il est possible d'étendre de tels systèmes en généralisant les structures de données du système. Maintenant, les justifications ne sont plus des clauses de Horn, mais des instances de règle d'inférence. L'utilisation du système reste inchangée. On peut asserter et rétracter les données de manière incrémentale.

11.2 Caractéristiques du PTMS

Le PTMS est un système de maintien de la cohérence dans une base de connaissances probabiliste. La logique probabiliste utilise des formules probabilistes $\Phi = \langle \varphi, p \rangle$, où φ est une formule et $p \in [0, 1]$. Les formules probabilistes vérifient la propriété suivante :

$$\langle \varphi, p \rangle \Rightarrow \forall q \, (q \leq p \Rightarrow \langle \varphi, q \rangle)$$

Cette propriété permet de donner un sens à $\langle \varphi, p \rangle$: « La formule φ est vraie avec une probabilité au moins égale à p ».

11.3 Probabilités

L'utilisation de probabilités permet de modéliser un système non monotone. Chaque élément de la base de connaissances est stocké avec un réel compris entre 0 et 1. $\frac{1}{2}$ représente l'ignorance totale à propos de la valeur de vérité d'une formule. 1 signifie que la formule est certainement vraie et 0 qu'elle est sûrement fausse. Une valeur entre 0 et 1 permet de coder une nuance intermédiaire.

La non monotonie et le maintien de la cohérence sont obtenus par la propagation de probabilités dans le réseau représentant la base de connaissances. Pour définir le calcul de la probabilité $P(A)$ d'une formule complexe A, nous allons utiliser les relations suivantes :

$$\begin{aligned} P(A \wedge B) &= \min(P(A), P(B)) \\ P(A \vee B) &= \max(P(A), P(B)) \\ P(\neg A) &= 1 - P(A) \end{aligned}$$

11.4 Représentation des données

Dans le réseau représentant la base de connaissances, les nœuds sont des termes et les arcs des instances de règles d'inférence.

11.4.1 Règles d'inférence

Une règle d'inférence $\left(\frac{LHS}{RHS}\right)_{COND,P}$ est un quadruplet $\langle LHS, RHS, COND, P\rangle$, où LHS et RHS sont les parties gauche et droites de la règle, $COND$ est une condition nécessaire à l'applicabilité de la règle et P la probabilité de la règle.

11.4.2 Instances de règles d'inférence

Une règle d'inférence instanciée est un cas particulier d'une règle d'inférence qui s'applique à un sous ensemble de la base de connaissance. Une instance de règle d'inférence $\left(\frac{LHS}{RHS}\right)_P$ est représentée par le triplet $\langle LHS, RHS, P\rangle$. LHS et RHS sont les instances des parties gauche et droite de la règle d'inférence.

11.4.3 Etats mentaux

Les états mentaux sont les données stockées dans le réseau. Un état mental A_P est représenté par le quadruplet $\langle A, P, JUSTIF, CONSEQ\rangle$. A est une formule vraie avec une probabilité P. $JUSTIF$ est l'ensemble des instances de règles d'inférence qui justifient A et $CONSEQ$ l'ensemble des instances de règles d'inférence dont l'activation dépend de A.

11.4.4 Base de connaissances

Une base de connaissances est un couple $\langle IRS, MSS\rangle$ où IRS est un ensemble de règles d'inférence et MSS un ensemble d'états mentaux. Le réseau est totalement décrit dans MSS puisque chaque état mental contient les règles d'inférence qu'il utilise.

11.5 Algorithme

Pour implémenter le PTMS, nous avons besoin :

- de créer une base vide
- de construire incrémentallement l'ensemble des règles d'inférence
- de construire incrémentallement l'ensemble des états mentaux en propageant les modifications de probabilités
- de lire les probabilités stockées dans le réseau (un état absent de la base aura par défaut une probabilité de $\frac{1}{2}$.

Dans la suite, on suppose que les règles d'inférence sont ajoutées au réseau avant les états mentaux.

11.5.1 Initialisation

Au départ, on a besoin d'une base vide. Une base de connaissances vide est le couple $\langle \phi, \phi\rangle$.

11.5.2 Ajout d'une règle d'inférence

Etant donné qu'il n'y a pas d'états mentaux dans la base au moment de l'ajout d'une nouvelle règle d'inférence, il suffit donc de mettre à jour l'ensemble des règles d'inférence. L'ajout de la règle d'inférence $\left(\frac{LHS}{RHS}\right)_{COND,P}$ dans la base $\langle IRS, \phi\rangle$ produit la base $\left\langle\left\{\left(\frac{LHS}{RHS}\right)_{COND,P}\right\} \cup IRS, \phi\right\rangle$.

11.5.3 Ajout d'un nouvel état mental

L'ajout d'un nouvel état mental est en fait la mise à jour du réseau existant en effectuant les propagations de probabilités nécessaires (éventuellement en instanciant de nouvelles règles d'inférence). Considérons d'abord la mise à jour sans propagation. Nous avons besoin de manipuler des termes avec leurs connecteurs logiques. La seule restriction est que les négations doivent apparaître devant des formules atomiques pour être traitées correctement. L'algorithme de mise à jour est récursif et dépend de chacun des constituants de la formule à traiter. Mettre à jour une formule avec une probabilité P signifie qu'après la mise à jour, elle sera vraie avec une probabilité au moins égale à P. Lorsqu'il y aura plusieurs mises à jour possibles, on ne réalisera que celles qui modifient le moins possible le réseau.

Conjonction

Pour mettre à jour $A_1 \wedge \ldots \wedge A_n$ avec la probabilité P, si P_i est la probabilité de A_i, on veut que $\min_i\{P_i\} \geq P$. La mise à jour minimale pour chaque A_i est donc $\max(P_i, P)$.

Disjonction

Pour mettre à jour $A_1 \vee \ldots \vee A_n$ avec la probabilité P, si P_i est la probabilité de A_i, on veut que $\max_i\{P_i\} \geq P$. Il y a deux cas. Si $\exists i(P_i \geq P)$ alors il n'y a rien à faire. Sinon il faut mettre à jour A_j avec la probabilité P pour j tel que $|P_j - P| = \min_i\{|P_i - P|\}$ (pour minimiser les modifications de la base de connaissances). Dans ce cas, on sait que $\forall i(P_i < P)$, donc la propriété précédente est équivalente à $P_j = \max P_i$.

Négation

Mettre à jour $\neg A$ avec la probabilité P signifie que A doit être faux avec au moins la probabilité P, donc que A doit être vrai avec au plus la probabilité $1-P$. Ainsi, si P_0 est la probabilité courante de A, la nouvelle probabilité sera $\min(P_0, 1-P)$.

Formule atomique

Mettre à jour A avec la probabilité P signifie que A doit être vrai avec au moins la probabilité P. Donc, si P_0 est la probabilité courante de A, la nouvelle probabilité sera $\max(P_0, P)$.

11.5.4 Propagation des probabilités

Lorsqu'un état mental est mis à jour avec une probabilité différente, il faut propager cette modification dans le réseau pour maintenir la cohérence. La propagation des probabilités se fait sur les instances de règles d'inférence, en avant ou en arrière.

Propagation en avant

Soit l'état mental A_P et l'instance de règle d'inférence $\left(\frac{A}{B}\right)_Q$. Le réseau doit être mis à jour avec $B_{\min(P,Q)}$.

Propagation en arrière

Soit l'état mental B_P et l'instance de règle d'inférence $\left(\frac{A}{B}\right)_Q$. Le réseau doit être mis à jour avec $A_{\max(P,1-Q)}$. En effet, ceci est équivalent à une propagation en avant de $\neg B_{1-P}$ suivant la règle $\left(\frac{\neg B}{\neg A}\right)_Q$ qui conduit à la mise à jour $\neg A_{\min(1-P,Q)}$, c'est à dire $A_{\max(P,1-Q)}$.

11.6 Implémentation

Une première implémentation du PTMS est proposée en annexe.

Chapitre 12

Planification dans le dialogue

12.1 Introduction

Nous allons voir comment utiliser la planification dans le dialogue. La planification doit permettre de produire des dialogues plus courts et plus efficaces. L'exemple du policier et du passant (page 47) montre que la seule notion de jeux de dialogue n'est pas suffisante. En effet, il aurait suffit au policier de se présenter avant de poser une question pour éviter le jeu de résolution de conflit. Cela vient du fait que dans les jeux on ne tient compte que du passé, et non des évolutions futures des agents et du dialogue qu'ils engendrent. Par évolutions futures, il faut bien sûr comprendre prévisions des évolutions futures en fonction des états mentaux et des buts. Le planificateur devra donc être capable de produire un plan tenant compte des éventuels plans que l'interlocuteur pourrait générer. De la même manière que pour les jeux, on aura donc une représentation des plans de l'autre agent. De plus comme le plan est élaboré à partir de suppositions, il doit pouvoir être révisable. On peut aussi imaginer un plan non déterministe, plusieurs actions restant possibles à un instant donné.

12.2 Description du planificateur

Le plan que l'on veut pouvoir produire doit être non déterministe (plusieurs actions possibles à un instant donné), non complètement construit (la planification de certaines séquences peut être reportée à plus tard, au moment où elles vont être exécutées par exemple). Un plan ne sera donc pas linéaire, mais pourra comporter des branches correspondant à des choix encore non déterminés au moment de la génération de plan (ce ne sont pas des plans en parallèle comme dans NOAH). Pour construire un plan on a besoin de deux types d'objets :

- des problèmes à résoudre,
- des actions élémentaires (coups) ou non (jeux).

Dans la suite, on utilisera la relation $\longmapsto$ pour désigner un moteur d'inférence utilisant les états mentaux de l'agent et les règles de mise à jour.

12.2.1 Problèmes

Un problème est un couple $\langle I, G\rangle$, où I est l'état initial et G le but à atteindre. Le but de l'agent constitue le problème initial que le planificateur va traiter. Pendant la planification, un problème peut faire office de plan. Cela permet de conserver des morceaux de plan non encore calculés (impossibilité de le faire, manque de temps, ...). Lorsque l'agent essayera d'exécuter un problème non résolu, il aura le choix entre tenter de planifier le problème (l'état mental ayant changé, il se peut que le problème ait maintenant une solution) et rentrer dans un jeu permettant de résoudre le problème. Le jeu étant ici vu comme une action non élémentaire du plan.

On définit aussi un problème avec conditions et contraintes $\langle I, Contr, Cond \rangle$. Ce problème n'est soluble que si $\langle I, Cond \rangle$ l'est et si après exécution du plan obtenu, les conditions $Cond$ sont vérifiées.

Un problème est dit trivial si $I \longmapsto Contr \wedge Cond$.

12.2.2 Actions

Une action (élémentaire ou non) est un triplet $\langle Contr, Cond, Update \rangle$. $Contr$ sont les contraintes devant être satisfaites pour exécuter l'action. $Cond$ sont des conditions que l'on doit satisfaire (après planification si nécessaire). Lorsque les contraintes et les conditions sont vérifiées, on peut exécuter l'action. Après l'exécution de l'action, les mises à jour $Update$ sont rajoutées aux états mentaux. Une action peut être élémentaire, elle est associée à un coup, ou non élémentaire, elle est alors associée à un jeu. L'exécution d'un coup est simplement sa génération, l'exécution d'un jeu est la pratique de ce jeu tant qu'il peut être joué par l'agent.

12.2.3 Jeux

Les jeux gardent la même définition que précédemment. On rajoute seulement la possibilité de générer de nouveaux plans à l'intérieur du jeu, en fonction des buts locaux au jeu. On considère le jeu comme une action non élémentaire. En effet, nous avions vu que les jeux et les coups ont un statut similaire. Ils ne peuvent être utilisés que sous certaines conditions et leur exécution provoque des modifications d'états mentaux.

12.3 Génération de plan

Au début du dialogue, un premier plan est généré. Ensuite, lorsqu'un agent est confronté à un problème, soit il rentre dans un jeu, soit il tente de planifier une solution à ce problème. Il se peut aussi que dans l'état courant, aucune alternative prévue par le plan ne soit applicable. Dans ce cas, il faut replanifier le dialogue.

Soit Γ l'opérateur qui calcule l'ensemble des actions permettant d'atteindre un but. Cet opérateur est défini par : $\Gamma(\langle I, G \rangle) = \{\langle Contr, Cond, Update \rangle \in Actions | Update \longmapsto G\}$. En partant d'un problème initial et en appliquant cet opérateur à chaque problème rencontré, on calcule en chaînage arrière un plan permettant de relier l'état initial au but final.

Le calcul de plan se déroule en deux étapes :

- Déterminer $\langle I, G \rangle$ le problème à résoudre
- On remplace ce problème par l'alternative entre les opérateurs sélectionnés par Γ : $\forall \langle Contr_i, Cond_i, Update_i \rangle \in \Gamma(\langle I, G \rangle)$, on crée le problème $\langle I, Contr_i, Cond_i \rangle$ puis on ajoute dans le plan l'action Act_i associée (s'il y en a une).
- Le problème $\langle I, G \rangle$ devient $\bigcup_i [\langle I, Contr_i, Cond_i \rangle, Act_i]$. Ici la réunion ensembliste de plans est interprétée comme un choix non déterministe de plans possibles
- On recommence la planification pour les problèmes $\langle I, Contr_i, Cond_i \rangle$, sachant que les problèmes triviaux sont éliminés du plan.

Cette méthode de construction semble ne construire que des alternatives de plan à chaque étape. En fait, lorsque $\Gamma(\langle I, G \rangle)$ est réduit à un singleton, il n'y a pas d'alternative et on construit une séquence de plans.

12.4 Exécution d'un plan

Le plan généré est donc dans le cas général un ensemble d'alternatives possibles. Si l'on suppose le problème du choix de l'alternative résolu, l'exécution d'une action du plan consiste à générer le coup associé à l'action choisie et à supprimer les autres alternatives. Il y a deux types d'actions.

Les actions élémentaires associées à des coups sont exécutées en générant le coups. Les actions non élémentaires correspondent à des plans non décomposés (problèmes non résolus). Pour les exécuter, il faut soit tenter de les replanifier, en espérant que les changements d'états mentaux précédents permettent maintenant de trouver un plan convenable, soit rentrer dans un jeu dont les conditions sont cohérentes avec le problème.

Au moment de l'exécution du plan, il faut vérifier que les contraintes et les conditions associées à chaque action du plan sont bien satisfaites. Le non respect de ces conditions peut provenir d'une représentation erronée de l'interlocuteur, donc d'une mauvaise prédiction de son comportement. Dans ce cas, il faut soit replanifier le but du plan précédent en considérant l'état courant comme état initial, soit retourner à un fonctionnement à base de jeux.

Remarque : Lorsque l'on supprime les alternatives non choisies, il faudrait supprimer seulement celles qui sont différentes. Le plan étant calculé en chaînage arrière, il se peut que deux branches différentes commencent par le même coup. Un plan $\{(x,y) \vee (x,z)\}$ devrait être remplacé par $\{x,(y \vee z)\}$ (on peut aussi exécuter le plan à la manière d'un automate non déterministe). Ainsi, on n'élimine pas trop d'alternatives et on repousse d'autant une éventuelle replanification.

12.5 Exemple

Reprenons l'exemple du policier et considérons la planification effectuée par le policier. Les actions et leurs coups associés sont décrits dans le tableau 12.1.

Action	Coup associé	Définition
$Demander(Q)$	$?_x Q$	$Contr = B_x B_y A_x KIF_x Q$ $Cond = \neg KIF_x Q$ $Update = B_x B_y W_x KIF_x Q$
$Dire(P)$	$!_x P$	$Contr = \phi$ $Cond = B_x P$ $Update = B_x B_y B_x P$
$Attendre(Q)$	ϕ	$Contr = \neg KIF_x Q \wedge B_x B_y W_x KIF_x Q$ $Cond = \phi$ $Update = KIF_x Q$

TAB. 12.1: Définition de quelques actions

Le but du policier (x) est $KIF_x q$. La seule action possible pour atteindre ce but est $Attendre(q)$, dont il reste à planifier les contraintes. Le tableau 12.2 montre comment on obtient le plan final, en chaînage arrière.

$[\langle I, KIF_x q\rangle]$	On applique $Attendre(q)$
$[\langle I, \neg KIF_x q \wedge B_x B_y W_x KIF_x q\rangle, Attendre(q)]$	Le problème se simplifie car $I \longmapsto \neg KIF_x q$
$[\langle I, B_x B_y W_x KIF_x q\rangle, Attendre(q)]$	On applique $Demander(q)$
$[\langle I, B_x B_y A_x KIF_x q\rangle, Demander(q), Attendre(q)]$	On a aussi $I \longmapsto B_x A_x KIF_x q$ et $!_x P \wedge B_y(P \rightarrow A_x KIF_x q) \longmapsto B_y A_x KIF_x q$, on peut donc appliquer $Dire(p)$
$[Dire(p), Demander(q), Attendre(q)]$	

TAB. 12.2: Plan construit à partir des actions du tableau 12.1

Le raisonnement ci-dessus s'explique de la manière suivante :

- Pour savoir « q », il faut attendre l'information.
- Pour attendre une information, il faut en avoir fait la demande.
- Pour pouvoir demander quelque chose, il faut convaincre de la légitimité de la demande.

– Pour cela, il faut justifier cette autorisation, l'agent x informe que « p ».

Du coté de l'agent y, il n'y a pas de but. Donc y est initialement dans le jeu de base, sans aucun plan.

12.6 Conclusion

L'utilisation d'un planificateur pour la génération de coups dans le dialogue permet de produire des dialogues plus courts et donc plus efficaces (à condition que la planification ne soit pas trop coûteuse par rapport au fonctionnement des jeux seuls). De plus, si l'agent possède une représentation de l'agent avec lequel il veut dialoguer, il va pouvoir simuler la planification effectuée par celui-ci pour en tenir compte dans son plan. Cela semble cependant poser quelques problèmes. En effet, le planificateur fonctionnant en chaînage arrière, on ne peut pas simuler l'exécution du plan de l'autre agent depuis le début du plan, puisque celui ci n'est pas encore calculé, l'entrelacement des deux plans est difficile à déterminer. D'un autre côté, un planificateur en chaînage avant peut être plus coûteux en temps puisqu'il fonctionne en aveugle.

Chapitre 13

Conclusion

Nous avons essayé au cours de ce travail de définir un modèle de jeux de dialogue basé sur les définitions déjà existantes et de l'implémenter en Prolog. L'approche que nous avons utilisée est essentiellement intentionnelle. Elle utilise une représentation des agents par leurs états mentaux (qui constituent le contexte cognitif de l'agent) et par leur système inférentiel. Le comportement des agents dans le dialogue est défini par la notion de jeu de dialogue. Nous nous sommes inspirés ici de la théorie des jeux pour définir un jeu de dialogue : les actes de dialogue sont les coups et les règles de mise à jour, de génération et d'interdiction définissent les coups possibles. A partir de là, nous avons rajouté la possibilité d'emboîter les jeux pendant le dialogue pour représenter la structure d'un dialogue, par exemple l'emboîtement des questions et des réponses. Finalement, avec ce modèle, nous avons pu réaliser quelques simulations de dialogue.

Pour réaliser un tel modèle nous avons été amenés à faire quelques choix. Les plus importants portent sur la représentation des agents (représentation des états mentaux et moteur inférentiel associé). Pour modéliser des agents artificiels les plus « naturels » possibles, nous n'avons pas utilisé de structure commune aux deux agents, mais des représentations de l'autre (des croyances sur les états mentaux de l'autre agent). De plus, l'omniscience doit être évitée, surtout pour la représentation de l'autre, cela est essentiel si l'on veut plus tard interfacer un agent artificiel avec un agent humain, celui-ci étant forcément limité. L'un des objectifs principaux a été d'expliciter au maximum les caractéristiques et le fonctionnement du modèle. Nous avons pu expliciter le système inférentiel (règles d'inférence redéfinissables), la représentation mentale de l'autre, les jeux et les règles définissant les jeux... Mais il manque encore a préciser le maintien (local) de la cohérence (le PTMS est une première approche mais reste implicite au niveau de l'agent) de même qu'une heuristique (une stratégie) de choix des coups générables contrôlée par l'agent. L'agent devrait pouvoir aussi être capable d'utiliser les jeux et leurs règles de manière « consciente » et non d'en subir les conséquences. Un moyen d'augmenter le degré d'explicitation des paramètres du modèle sera l'utilisation de jeux logiques, les coups logiques permettant de contrôler plus finement les inférences, les applications de règles et le maintien de la cohérence.

La prochaine étape dans l'évolution de ce modèle est l'intégration de la planification et de la reconnaissance de plan qui, comme on l'a vu, permettent d'être plus efficace au niveau du dialogue.

Les jeux de dialogue, associés aux actes de dialogue, fournissent donc un cadre assez souple pour modéliser le dialogue, mais aussi les agents du dialogue. Une application intéressante de ce modèle serait la définition d'un langage de programmation basé sur les jeux logiques et de dialogue et sur les actes de dialogue, avec une architecture client/serveur intelligents, intégrant des primitives de planification.

Deuxième partie

Extension et révision du modèle

Chapitre 14

Introduction

Dans la première partie de ce rapport nous avons présenté un modèle de dialogue. Ce modèle va être ici complété pour tenir compte des coups logiques (voir (Maudet and Evrard, 1998)) et donc aussi des jeux logiques (voir page 56). De plus les coups et les jeux vont être définis de façon à simplifier leur utilisation dans un cadre de planification.

Chapitre 15

Représentation des connaissances

Le modèle précédent n'est pas complètement satisfaisant au niveau de la représentation des connaissances. Nous allons conserver ici une logique modale, mais aussi redéfinir l'attribution des forces aux états mentaux et introduire un datage des connaissances.

15.1 Force des états mentaux

Dans le modèle précédant, une force était associée aux états mentaux. Il semble plus judicieux d'associer cette force aux modalités. Cela permet d'introduire des nuances dans les croyances par exemple. Une agent peut croire fortement qu'un autre pourrait avoir des raisons de croire quelque chose ($B_x^1 B_y^{0,6} p$) ou croire plus faiblement qu'une autre agent est convaincu de quelque chose ($B_x^{0,6} B_y^1 p$).

La force de l'état mental proprement dite sera alors la force associée à la modalité principale de la formule représentant l'état mental.

15.2 Datage des états mentaux

Certains actes de langage ne peuvent être bien pris en compte que si les états mentaux sont datés. Nous avons en effet besoin d'une logique temporelle pour traiter la promesse par exemple. Avant d'effectuer une promesse, l'auditeur n'est pas tenu de réaliser l'action promise. L'obligation est un effet de la promesse. Ce changement d'état est mieux décrit si l'on peut tenir compte du temps dans la logique. Le modèle précédent ne permettait de coder les états mentaux qu'au présent. Si l'on date les états mentaux, on va aussi pouvoir tenir compte de croyances sur le passe, et donc détecter des changements d'états mentaux. Cependant, comme pour l'attribution des forces, il faut ici attribuer une date à chaque modalité de la formule représentant l'état mental.

Pour que cela fonctionne, il faut aussi rajouter des règles permettant de faire évoluer les états mentaux en fonction du temps. On considérera que le temps est discrêt, chaque instant étant représenté par un entier. Considérons par exemple la régle d'inférence suivante :

$$\frac{M_x^{t,p}\varphi}{M_x^{t+1,p}\varphi}$$

Cette règle simule l'évolution d'un état mental (d'un type M quelconque) dans le temps. Une telle règle peut être associée à une force afin de conforter une croyance, ou de simuler des troubles de la mémoire.

Pour résumer, un état mental sera codé comme le tableau 15.1 l'indique.

Dans la suite, les forces et les dates ne seront pas systématiquement précisées pour ne pas surcharger l'écriture des coups et des jeux.

Codage d'un état mental :	
– de type M, – de contenu propositionnel φ, – à l'instant t, – de force p, – chez l'agent x.	$M_x^{t,p}\varphi$

TAB. 15.1: Codage des états mentaux

15.3 Règles d'inférence

Jusqu'à présent, les règles d'inférence possédaient une force qui permettait de multiplier les forces des états mentaux. Cela peut poser un problème. Par exemple, une règle qui multiplie par 2 une incertitude totale (probabilite de $\frac{1}{2}$) va produire une certitude. En fait, c'est la distance à $\frac{1}{2}$ qu'il faut multiplier (puis borner par 0 et 1 si l'on utilise effectivement des probabilités). Si un état a une probabilité p, la probabilité p' obtenue après application d'un coefficient k vérifie la propriété : $|\frac{1}{2} - p'| = k|\frac{1}{2} - p|$, en restant bornée par 0 et 1. On a donc : $p' = \min\left(1, \max\left(0, \frac{1}{2} + k(p - \frac{1}{2})\right)\right)$

Les algorithmes permettant la propagation des probabilités deviennent donc :

$$\begin{array}{llll} A_\alpha & \left(\frac{A}{B}\right)_\gamma \quad B_\beta & \Longrightarrow & B_{\max\left(\beta,\min\left(1,\max\left(0,\frac{1}{2}+\gamma(\alpha-\frac{1}{2})\right)\right)\right)} \\ & & & = B_{\max\left(\beta,\min\left(1,\frac{1}{2}+\gamma(\alpha-\frac{1}{2})\right)\right)} \\ & & & \\ (\neg B)_{1-\beta} & \left(\frac{\neg B}{\neg A}\right)_\gamma \quad A_\alpha & \Longrightarrow & (\neg A)_{\max\left(1-\alpha,\min\left(1,\max\left(0,\frac{1}{2}+\gamma(1-\beta-\frac{1}{2})\right)\right)\right)} \\ & & & = A_{\min\left(\alpha,\max\left(0,\frac{1}{2}+\gamma(\beta-\frac{1}{2})\right)\right)} \end{array}$$

Nous avons utilisé ici une logique probabiliste. Cependant, il n'est pas nécessaire de reprendre le PTMS (voir page 58) puisque les jeux logiques vont permettre d'appliquer les règles d'inférence et donc de maintenir (localement) la cohérence.

Chapitre 16

Actes de dialogue

Nous allons ici critiquer la précédente représentation des actes de dialogue et en proposer une nouvelle permettant la prise en compte du temps, de la planification, des implicatures ...

16.1 Inconvénients du précédent modèle

Les actes de dialogue, tels qu'ils étaient définis jusqu'à présent, ne permettaient de communiquer que la famille d'acte, un contenu propositionnel et la date de l'acte. On ne tenait pas compte de la force de l'acte. Donc, non seulement pour être cohérent avec la nouvelle représentation des états mentaux, mais aussi pour modéliser des nuances à l'intérieur d'une même classe d'actes de dialogue, un acte de dialogue contiendra son type, un contenu propositionnel, une date et une force.

Le modèle précédent ne permettait pas non plus de planifier simplement les coups dans le dialogue. En effet, les conditions et les effets des coups étaient codés dans les règles des jeux. Nous allons ici associer ces conditions et ces effets à l'acte.

16.2 Représentation des actes de dialogue

Finalement, un acte de dialogue sera représenté par :

Expression en logique modale

Conditions Equivalent des règles de génération

Interdictions Equivalent des règles d'interdiction

Effets sur l'auditeur Effets produits par l'acte sur l'auditeur

Effets sur le locuteur Effets produits par l'acte sur le locuteur

De plus, l'expression en logique modale de l'acte est la même que celle des états mentaux. Un acte A, de contenu propositionnel φ, énoncé par l'agent x à la date t avec la force p s'écrira : $A_x^{t,p}\varphi$

Par exemple l'acte de dialogue $?_x^{t,p}q$ sera caractérisé par :

$$\begin{array}{lcl} Condition & = & W_x^{t,p}KIF_x^{t,p'}q \\ Interdiction & = & \neg KIF_x^{t,p'}q \\ Effect/y & = & B_y^{t,1}W_x^{t,p}KIF_x^{t,p}q \\ Effect/x & = & B_x^{t,1}B_y^{t,1}W_x^{t,p}KIF_x^{t,p}q \end{array}$$

Le tableau 16.1 résume le codage d'un acte de dialogue.

16.2.1 Remarque

Il semblerait que les actes de dialogue aient une structure assez particulière. L'exemple précédent semble montrer que les effets d'un acte se déduisent des conditions qui ont rendu la génération

$$A_x^{t,p}\varphi \; = \; \boxed{\begin{array}{|lcl|}\hline Condition & = & \ldots \\ Interdiction & = & \ldots \\ \hline Effect/x & = & \ldots \\ Effect/y & = & \ldots \\ \hline \end{array}}$$

TAB. 16.1: Codage des actes de dialogue

possible. Les effets seraient alors des croyances sur les conditions. Il semble en effet assez naturel de croire que les conditions d'un acte sont remplies lorsque celui-ci est joué. On aurait donc $Effets/y = \tilde{B}_y Conditions$ et $Effets/x = \tilde{B}_x Effets/y$.

Cela fonctionne avec certains actes (!, ?, !?), mais pas avec la promesse. Si l'acte $\&_x p$ (x promet p) est produit, c'est que x s'engage à ce que p devienne vrai ($W_x p$), alors qu'avant la promesse, il n'en avait pas l'intention ($\neg W_x p$) ou l'obligation. On ne peut pas avoir chez y à la fois $B_y \neg W_x p$ et $B_y W_x p$. On ne peut donc pas calculer systématiquement les effets à partir des conditions. Nous conserverons donc les effets dans la définition des actes.

Chapitre 17

Jeux logiques et jeux de dialogue

17.1 Introduction

Nous avons évoqué dans la première partie de ce rapport la notion de jeux logiques. Les jeux logiques permettent de contrôler les inférences, ou plus généralement les coups logiques. Par opposition, les jeux de dialogue permettront le contrôle des coups de dialogue. Nous avons déjà défini les coups de dialogue (les actes de dialogue). Il reste donc à définir les coups logiques, puis les jeux logiques. Nous verrons ensuite ce qu'il reste aux jeux de dialogue.

17.2 Jeux logiques

17.2.1 Coups logiques

Si l'on se réfère au modèle précédent, les coups logiques peuvent être des applications de règles d'inférence ou de mise à jour. Cependant, les règles de mise à jour fonctionnent de la même manière que les règles d'inférence. On ne va donc pas les distinguer ici, et ne considérer que des règles d'inférence. La principale différence entre les coups logiques et les coups de dialogue est la portée du coup. Le coup de dialogue a des effets chez l'auditeur et le locuteur. Le coup logique étant interne à l'agent qui le joue, il n'a d'effets que chez l'agent. A ceci près, la définition d'un coup logique ressemble à celle d'un acte de dialogue. Par exemple, on pourra représenter le modus ponens par le coup logique suivant :

$$\frac{B_x(P \rightarrow Q) \wedge B_x P}{B_x Q} = \boxed{\begin{array}{lcl} Condition & = & W_x KIF_x Q \\ Interdiction & = & KIF_x Q \end{array}}$$

Les effets (ici $B_x Q$) sont entièrement contenus dans l'expression du coup, il est donc inutile de spécifier un champ $effets$.

17.2.2 Jeux logiques

Les jeux logiques doivent contrôler les coups logiques définis précédemment. Tout comme les jeux de dialogue, ils doivent être activés en fonction du contexte. On va donc conserver dans la définition des jeux logiques la condition d'entrée, l'invariant et la condition de sortie. Les jeux logiques contiennent de plus les coups logiques qu'ils vont pouvoir générer. Par exemple, un jeux logique de « réflexion » peut s'écrire :

$$reflection(Q) = \begin{array}{|l|} \hline \begin{array}{lcl} Entrance & = & W_x KIF_x Q \\ Invariant & = & W_x KIF_x Q \\ Exit & = & KIF_x Q \end{array} \\ \hline \frac{B_x(P \rightarrow Q) \wedge B_x P}{B_x Q} = \boxed{\begin{array}{lcl} Condition & = & W_x KIF_x Q \\ Interdiction & = & KIF_x Q \end{array}} \\ \hline \end{array}$$

La génération d'un coup logique dépend du jeu logique dans lequel il se trouve. Il doit être généré avec la force $P(Condition \wedge \neg Interdiction \wedge Entrance \wedge Invariant \wedge \neg Exit)$ afin de tenir compte du contexte qui a activé le jeu logique.

Pour résumer, les coups logiques et les jeux logiques se codent comme indiqué dans les tableaux 17.1 et 17.2.

$$\frac{A}{B} = \boxed{\begin{array}{lcl} Condition & = & \ldots \\ Interdiction & = & \ldots \end{array}}$$

TAB. 17.1: Codage des coups logiques

$$JeuLogique = \boxed{\begin{array}{lcl} Entrance & = & \ldots \\ Invariant & = & \ldots \\ Exit & = & \ldots \\ \hline Move & = & \ldots \\ \ldots & = & \end{array}}$$

TAB. 17.2: Codage des jeux logiques

17.3 Jeux de dialogue

Les jeux de dialogue sont définis de manière analogue aux jeux logiques, la seule différence étant la nature des coups. Les coups de dialogue ont déja été décrits (voir page 72). Par exemple, un jeu de réponse peut être représenté par :

$$reponse(Q) = \boxed{\begin{array}{lcl} Entrance & = & B_x W_y KIF_y Q \\ Invariant & = & B_x W_y KIF_y Q \\ Exit & = & B_x KIF_y Q \\ \hline !_x Q & = & \boxed{\begin{array}{lcl} Condition & = & B_x Q \\ Interdiction & = & B_x \neg Q \\ \hline Effect/x & = & B_x B_y B_x Q \\ Effect/y & = & B_y B_x Q \end{array}} \end{array}}$$

La génération d'un coup de dialogue dépend du jeu de dialogue dans lequel il se trouve. Donc, comme pour les coups logiques, il doit être généré avec la force $P(Condition \wedge \neg Interdiction \wedge Entrance \wedge Invariant \wedge \neg Exit)$ afin de tenir compte du contexte qui a activé le jeu de dialogue.

La structure des jeux de dialogue est résumée dans le tableau 17.3.

$$JeuDeDialogue = \boxed{\begin{array}{lcl} Entrance & = & \ldots \\ Invariant & = & \ldots \\ Exit & = & \ldots \\ \hline Move & = & \ldots \\ \ldots & = & \end{array}}$$

TAB. 17.3: Codage des jeux de dialogue

Troisième partie

Annexes

Annexe A

Maximes de conversation

A.1 Maximes de conversation de Grice

En linguistique, les maximes de conversation (Grice, 1975) permettent d'orienter une conversation. Ces maximes sont les suivantes :

A.1.1 Maximes de quantité

N1 Make your contribution as informative as is required (for the current purposes of the exchange).

N2 Do not make your contribution more informative than is required.

Les maximes N1 et N2 expriment des conditions sur la quantité d'information à échanger. Un comportement coopératif est basé sur un échange à la fois suffisant (N1) et minimal (N2).

A.1.2 Maximes de qualité

Q1 Do not say what you believe to be false.

Q2 Do not say that for which you lack adequate evidence.

Les maximes Q1 et Q2 expriment des conditions sur la qualité de l'information à échanger. Si l'on considère toujours un comportement coopératif, l'échange ne doit pas comporter d'information fausse (Q1), ou que l'on ne peut pas justifier (Q2).

A.1.3 Maxime de relation

R1 Be relevant.

La maxime R1 exprime la pertinence de la contribution par rapport au contexte de la conversation.

A.1.4 Maximes de manière

M1 Avoid obscurity of expression.

M2 Avoid ambiguity.

M3 Be brief (avoid unnecessary prolixity).

M4 Be orderly

Les maximes de manière décrivent de quelle manière la contribution doit être faite. L'expression doit être claire (M1) et non ambiguë (M2). Pour cela, il faut être bref (M3) et méthodique (M4).

A.2 Principes de planification du langage de Tauli

Des travaux similaires à ceux de (Grice, 1975) ont été menés par (Tauli, 1968). Pour une comparaison plus détaillée de ces deux travaux, on se reportera à (Traunmuller, 1991). On retrouve chez Tauli des principes qu'il a énoncés ainsi :

A.2.1 Principes de clarté

C1 The expression must convey to the listener all the meanings intended by the speaker.

C3 The expression must contain redundancy.

C4 The greater the possibility for semantic confusion the greater must be the difference in expression.

Pour qu'une contribution soit la plus claire possible, elle doit contenir toute l'information que le locuteur a l'intention de communiquer (C1), éventuellement contenir des redondances (C2) et lever les ambiguïtés lorsque cela est nécessaire (C3).

A.2.2 Principes d'économie

E1 The expression must not convey more meaning than necessary.

E2 The expression must be the shortest possible.

E3 The more frequent the expression the shorter it must be.

Pour qu'une contribution soit la plus économique possible, il faut qu'elle comporte l'information minimale requise (E1), qu'elle soit la plus courte possible (E2) et plus une expression est fréquente, plus elle doit être courte (E3).

A.2.3 Comparaisons avec les maximes de Grice

Les principes C1 et E1 décrivent la qualité et la quantité d'information échangée. Elles sont à rapprocher des maximes de quantité et de qualité de Grice.

Les principes C3, C4, E2, E3 expriment la manière dont l'information doit être échangée. Elles sont à rapprocher des maximes de manière de Grice.

Annexe B

Probabilistic Truth Maintenance System

Le PTMS a été décrit dans le rapport (page 11). Le programme qui suit est une implémentation du PTMS en Prolog.

```
/****************************************************************************/
/*                                                                          */
/* Probabilistic Truth Maintenance System                                   */
/*                                                                          */
/****************************************************************************/

/* Le reseau est stocke dans la BD de Prolog (soit ptms cette cle)
 * Dans la suite : key-name = format designe un objet de type 'name'
 * enregistre avec la cle 'key' au format 'format'
 *
 * Liste des PTMS en cours d'utilisation :
 *  recorded_ptms-KEY = KEY
 *
 * Reseau des connaissances :
 *
 *  ptms-PTMS   = ptms(IRK,IIRK,MSK,GAME)
 *
 *  ptms_ir-IR  = ir(GAME,LHS,RHS,NLHS,NRHS,COND,P)
 *  ptms_iir-IIR= iir(REF,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)
 *  ptms_ms-MS  = ms(REF,A,P,JUSTIF,CONSEQ)
 *  PREMISSES   = [REF]
 *  CONCL       = [REF]
 *  JUSTIF      = [REF]
 *  CONSEQ      = [REF]
 *
 * Donnees intermediaires :
 *  state_to_propagate  : file des etats a propager (DEPTH,STATE,PROBA)
 */

:-module(ptms,
    [
        new_ptms/1,         % new_ptms(PTMS)
        delete_ptms/1,      % delete_ptms(PTMS)
        print_ptms/1,       % print_ptms(PTMS)
```

```
        copy_ptms/2,        % copy_ptms(PTMS1,PTMS2)
        get_ptms_game/2,    % get_ptms_game(PTMS,GAME)
        set_ptms_game/2,    % set_ptms_game(PTMS,GAME)
        new_ir/3,           % new_ir(PTMS,IR,GAME)
        ask/3,              % ask(PTMS,A,P)
        store/4             % store(PTMS,DEPTH,A,P)
    ]
).

:-use_module(cd-tools).

/****************************************************************************/
/*                                                                          */
/* Definition des connecteurs logiques                                      */
/*                                                                          */
/****************************************************************************/

:-op(
    [
        970,    xfx,    #,
        960,    xfy,    |,
        950,    xfx,    then,
        500,    yfx,    or,
        400,    yfx,    and,
        200,    fx,     -
    ]
).

/****************************************************************************/
/*                                                                          */
/* Initialisation du module                                                 */
/*                                                                          */
/****************************************************************************/

init:-
    empty_db([recorded_ptms,state_to_propagate,rule_to_connect,state_to_connect,
state_to_eventually_connect,connection_to_propagate]),
    forall(recorded(recorded_ptms,PTMS),delete_ptms(PTMS)).

/****************************************************************************/
/*                                                                          */
/* Creation d'un nouveau PTMS                                               */
/*                                                                          */
/****************************************************************************/

new_ptms(PTMS):-
    (   nonvar(PTMS)
    ;   gensym(ptms,PTMS)
    ),
    !,
    delete_ptms(PTMS),
    concat(PTMS,'_ir',IRK),
    concat(PTMS,'_iir',IIRK),
    concat(PTMS,'_ms',MSK),
```

```
    recordz(PTMS,ptms(IRK,IIRK,MSK,no_game_at_initialisation)),
    recordz(recorded_ptms,PTMS).

/*****************************************************************************/
/*                                                                           */
/* Destruction d'un PTMS                                                     */
/*                                                                           */
/*****************************************************************************/

delete_ptms(PTMS):-
    (   recorded(recorded_ptms,PTMS,REF1)
    ->  erase(REF1)
    ;   true
    ),
    recorded(PTMS,ptms(IRK,IIRK,MSK,_),REF2),
    !,
    empty_db([IRK,IIRK,MSK]),
    erase(REF2).
delete_ptms(_).

/*****************************************************************************/
/*                                                                           */
/* Affichage du reseau d'un PTMS                                             */
/*                                                                           */
/*****************************************************************************/

print_ptms(PTMS):-
    recorded(PTMS,ptms(IRK,IIRK,MSK,CUR_GAME)),
    !,
    writef("Jeu courant : %t\n",[CUR_GAME]),
    (   nl,
        recorded(IRK,ir(GAME,LHS,RHS,_,_,COND,P)),
        numbervars((GAME,LHS,RHS,COND),0,_),
        writef("%t : %t then %t\t| %t\t# %t\n",[GAME,LHS,RHS,COND,P]),
        fail
    ;   nl,
        recorded(IIRK,iir(REF,GAME,LHS,RHS,_,_,P,_,_)),
        numbervars((GAME,LHS,RHS),0,_),
        writef("%t\t%t : %t then %t\t\t# %t\n",[REF,GAME,LHS,RHS,P]),
        fail
    ;   nl,
        recorded(MSK,ms(REF,A,P,_,_)),
        numbervars(A,0,_),
        writef("%t\t%t\t# %t\n",[REF,A,P]),
        fail
    ;   nl
    ).

/*****************************************************************************/
/*                                                                           */
/* Duplication d'un PTMS                                                     */
/*                                                                           */
/*****************************************************************************/
```

```
copy_ptms(PTMS1,PTMS2):-
    nonvar(PTMS1),
    new_ptms(PTMS2),
    get_ptms_game(PTMS1,GAME),
    set_ptms_game(PTMS2,GAME),
    recorded(PTMS1,ptms(IRK1,IIRK1,MSK1,_)),
    recorded(PTMS2,ptms(IRK2,IIRK2,MSK2,_)),
    copy(IRK1,IRK2),
    copy(IIRK1,IIRK2),
    copy(MSK1,MSK2).

copy(K1,K2):-
    forall(recorded(K1,X),recordz(K2,X)).

/******************************************************************************/
/*                                                                            */
/* Lecture du jeu courant d'un PTMS                                           */
/*                                                                            */
/******************************************************************************/

get_ptms_game(PTMS,GAME):-
    recorded(PTMS,ptms(_,_,_,GAME)).

/******************************************************************************/
/*                                                                            */
/* Changement du jeu courant d'un PTMS                                        */
/*                                                                            */
/******************************************************************************/

set_ptms_game(PTMS,GAME):-
    recorded(PTMS,ptms(IRK,IIRK,MSK,_),REF),
    erase(REF),
    recordz(PTMS,ptms(IRK,IIRK,MSK,GAME)).

/******************************************************************************/
/*                                                                            */
/* Ajout d'une nouvelle regle d'inference                                     */
/*                                                                            */
/******************************************************************************/

new_ir(PTMS,(LHS then RHS|COND#P),GAME):-!,new_ir0(PTMS,LHS,RHS,COND,P,GAME).
new_ir(PTMS,(LHS then RHS|COND),GAME):-!,new_ir0(PTMS,LHS,RHS,COND,1,GAME).
new_ir(PTMS,(LHS then RHS#P),GAME):-!,new_ir0(PTMS,LHS,RHS,true,P,GAME).
new_ir(PTMS,(LHS then RHS),GAME):-!,new_ir0(PTMS,LHS,RHS,true,1,GAME).

new_ir0(PTMS,LHS,RHS,COND,P,GAME):-
    recorded(PTMS,ptms(IRK,_,_,_)),
    neg(LHS,NLHS),
    neg(RHS,NRHS),
    recordz(IRK,ir(GAME,LHS,RHS,NLHS,NRHS,COND,P)).

neg(P,NP):-simpl(-P,NP).

/******************************************************************************/
```

```
/*                                                                          */
/* Interrogation d'un PTMS                                                  */
/*                                                                          */
/****************************************************************************/

ask(PTMS,A,P):-
    recorded(PTMS,ptms(_,_,MSK,_)),
    simpl(A,A0),
%   findall(P0,ask0(MSK,A0,P0),LP),
%   max(LP,P).
    ask0(MSK,A0,P).

ask0(MSK,A,P):-
    var(A),
    !,
    recorded(MSK,ms(_,A,P,_,_)).
%   fail.

ask0(_,true,1):-!.
ask0(_,false,0):-!.

ask0(MSK,A and B,P):-
    !,
    ask0(MSK,A,PA),
    ask0(MSK,B,PB),
    P is min(PA,PB).

ask0(MSK,A or B,P):-
    !,
    ask0(MSK,A,PA),
    ask0(MSK,B,PB),
    P is max(PA,PB).

ask0(MSK,-A,P):-
    !,
    ask0(MSK,A,PA),
    P is 1-PA.

ask0(MSK,A,P):-
    (   recorded(MSK,ms(_,A,P,_,_))
    *-> true
    ;   P=0.5
    ).

simpl(V,V):- var(V), !.
simpl(-V,-V):- var(V), !.
simpl(-(-X),U):- !, simpl(X,U).
simpl(-(X and Y),U or V):- !, simpl(-X,U), simpl(-Y,V).
simpl(-(X or Y),U and V):- !, simpl(-X,U), simpl(-Y,V).
simpl(T,TS):-
    functor(T,F,N),
    !,
    functor(TS,F,N),
    simpl_args(N,T,TS).
```

```
simpl(T,T).

simpl_args(0,_,_):-!.
simpl_args(N,T,TS):-
    arg(N,T,A),
    simpl(A,AS),
    arg(N,TS,AS),
    N1 is N-1,
    simpl_args(N1,T,TS).

/****************************************************************************/
/*                                                                          */
/* Mise a jour d'un PTMS                                                    */
/*                                                                          */
/****************************************************************************/

store(PTMS,DEPTH,A,P):-
    simpl(A,A0),
    store0(PTMS,DEPTH,A0,P),
    !,
%   update(PTMS,DEPTH,A,P).
    propagate(PTMS).
store(_,_,_,_).

store1(PTMS,DEPTH,A,P):-
    simpl(A,A0),
    store0(PTMS,DEPTH,A0,P).

/* La mise a jour se fait de maniere a modifier le moins possible la
 * base de connaissances.
 */

/* store0 echoue s'il n'y a pas de modif a faire ! */

store0(_,_,V,_):-
    var(V),
    !,
    fail.

/* Conjonction : (A and B)(P)
 * KB={A(Pa), B(Pb)}
 * On veut : min{Pa',Pb'}>=P
 * donc Pi'=max(Pi,P)
 */

store0(K,D,A and B,P):-
    !,
    ask(K,A,PA),
    ask(K,B,PB),
    (   P>PA
    ->  store0(K,D,A,P),
        (   P>PB
        ->  store0(K,D,B,P)
        ;   true
```

```
        )
    ;   P>PB
    ->  store0(K,D,B,P)
    ).


/* Disjonction : (A or B)(P)
 * KB={A(Pa), B(Pb)}
 * On veut : max{Pa', Pb'}>=P
 * Donc si Pa<P et Pb<P, on maj l'etat le plus proche de P
 */

store0(K,D,A or B,P):-
    !,
    ask(K,A,PA),
    (   PA<P
    ->  ask(K,B,PB),
        (   PB<P
        ->  (   PA<PB           % PB plus proche de P que PA
            ->  store0(K,D,B,P)
            ;   store0(K,D,A,P)
            )
        )
    ).

/* Negation : (-A)(P)
 * KB={A(Pa)}
 * (-A)(P) : "A est faux avec une proba >= P
 * donc A est vrai avec une proba =< 1-P
 * donc Pa'=min{Pa,1-P}
 */

store0(_,_,-V,_):-
    var(V),
    !,
    fail.

%store0(K,D,-(A and B),P):-!, store0(K,D,-A or -B,P).
%store0(K,D,-(A or B),P):-!, store0(K,D,-A and -B,P).
%store0(K,D,-(-A),P):-!, store0(K,D,A,P).

store0(K,D,-A,P):-
    !,
    ask(K,A,PA),
    NPA is 1-P,
    (   NPA<PA
    ->  recordz(state_to_propagate,(D,A,NPA))
    ).

/* Fait positif : A(P)
 * KB={A(Pa)}
 * A(P) : "A est vrai avec une proba >= P
 * donc Pa'=max{Pa,P}
 */
```

```
store0(K,D,A,P):-
    ask(K,A,PA),
    (   P>PA
    ->  recordz(state_to_propagate,(D,A,P))
    ).

/*****************************************************************************/
/*                                                                           */
/* Propagation des modifications dans le reseau                              */
/*                                                                           */
/*****************************************************************************/

propagate(PTMS):-
    recorded(state_to_propagate,(D,A,P),REF),
    erase(REF),
    update(PTMS,D,A,P),
    (   D > 0
    ->  D1 is D-1,
        recorded(PTMS,ptms(_,IIRK,MSK,GAME)),
        recorded(MSK,ms(_,A,_,JUSTIF,CONSEQ)),
        !,
        (   member(REF_IIR,CONSEQ),
            recorded(IIRK,iir(REF_IIR,GAME,LHS,RHS,_,_,P_IIR,_,_)),
            ask(PTMS,LHS,P_LHS),
            P_RHS is min(P_LHS,P_IIR),
            store1(PTMS,D1,RHS,P_RHS),
            fail
        ;   member(REF_IIR,JUSTIF),
            recorded(IIRK,iir(REF_IIR,GAME,_,_,NLHS,NRHS,P_IIR,_,_)),
            ask(PTMS,NRHS,P_NRHS),
            P_NLHS is max(P_NRHS,1-P_IIR),
            store1(PTMS,D1,NLHS,P_NLHS),
            fail
        ;   true
        )
    ),
    propagate(PTMS).
propagate(_).   % Pas d'etat a propager

/*****************************************************************************/
/*                                                                           */
/* Mise a jour des etats propages                                            */
/*                                                                           */
/*****************************************************************************/

update(PTMS,D,A,P):-
    recorded(PTMS,ptms(_,_,MSK,_)),
    (   recorded(MSK,ms(REFA,A,_,JUSTIF,CONSEQ),REF)
    ->  erase(REF),
        recordz(MSK,ms(REFA,A,P,JUSTIF,CONSEQ))
    ;   gen_ref(REFA),
        recordz(MSK,ms(REFA,A,P,[],[])),
        connect(PTMS,REFA,D)
```

```
    ).

/****************************************************************************/
/*                                                                          */
/* Connexion d'un etat au PTMS                                              */
/*                                                                          */
/****************************************************************************/

connect(_,_,0):-!.
connect(PTMS,REFA,DEPTH):-
    recorded(PTMS,ptms(_,IIRK,MSK,_)),
    (   instanciated_inference_rule(PTMS,IIR),
        (   rule_uses(IIR,REFA),
            new_rule(IIRK,IIR)
        ->  IIR=iir(REF_IIR,_,_,_,_,_,_,_,_),
            gen_ref(REF_IIR),
            recordz(rule_to_connect,IIR),
            (   recorded(state_to_eventually_connect,ST0,REF),
                erase(REF),
                ST=ST0,  %simpl(ST0,ST),
                recordz(state_to_connect,ST),
                fail
            ;   true
            )
        ;   empty_db(state_to_eventually_connect)
        ),
        fail
    ;   recorded(state_to_connect,(NEW_REF,A,P),REF),
        erase(REF),
        recordz(MSK,ms(NEW_REF,A,P,[],[])),
        DEPTH1 is DEPTH-1,
        recordz(connection_to_propagate,(DEPTH1,NEW_REF)),
        fail
    ;   recorded(rule_to_connect,IIR,REF),
        erase(REF),
        connect_rule(PTMS,IIR),
        fail
    ;   recorded(connection_to_propagate,(D,R),REF)
    ->  erase(REF),
        connect(PTMS,R,D)
    ;   true
    ).

connect_rule(PTMS,iir(REF_IIR,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)):-
    recorded(PTMS,ptms(_,IIRK,MSK,_)),
    recordz(IIRK,iir(REF_IIR,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)),
    (   member(REF_MS,PREMISSES),
        recorded(MSK,ms(REF_MS,A,PA,JUSTIF,CONSEQ),REF),
        erase(REF),
        recordz(MSK,ms(REF_MS,A,PA,JUSTIF,[REF_IIR|CONSEQ])),
        fail
    ;   member(REF_MS,CONCL),
        recorded(MSK,ms(REF_MS,A,PA,JUSTIF,CONSEQ),REF),
        erase(REF),
```

```
        recordz(MSK,ms(REF_MS,A,PA,[REF_IIR|JUSTIF],CONSEQ)),
        fail
    ;   true
    ).

rule_uses(iir(_,_,_,_,_,_,_,PREMISSES,_),REFA):-
    memberchk(REFA,PREMISSES),
    !.
rule_uses(iir(_,_,_,_,_,_,_,_,CONCL),REFA):-
    memberchk(REFA,CONCL).

new_rule(IIRK,iir(_,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)):-
    \+recorded(IIRK,iir(_,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)),
    \+recorded(rule_to_connect,iir(_,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)).

/****************************************************************************/
/*                                                                          */
/* Instanciation d'une regle d'inference                                    */
/*                                                                          */
/****************************************************************************/

/* Le backtracking fournit toutes les instances possibles
 */

instanciated_inference_rule(PTMS,
iir(_,GAME,LHS,RHS,NLHS,NRHS,P,PREMISSES,CONCL)):-
    recorded(PTMS,ptms(IRK,_,MSK,GAME)),
    recorded(IRK,ir(GAME,LHS0,RHS0,NLHS0,NRHS0,COND,P)),
    inst(LHS0,MSK,PREMISSES),
    COND,
    simpl(LHS0,LHS),
    simpl(RHS0,RHS),
    simpl(NLHS0,NLHS),
    simpl(NRHS0,NRHS),
    formula_items(RHS,MSK,CONCL).

inst(A,MSK,PA):-
    var(A),
    !,
    inst0(A,MSK,PA).

inst(A and B,MSK,PAB):-
    !,
    inst(A,MSK,PA),
    inst(B,MSK,PB),
    union(PA,PB,PAB).

inst(A or B,MSK,PAB):-
    !,
    inst(A,MSK,PA),
    inst(B,MSK,PB),
    union(PA,PB,PAB).

inst(-A,MSK,PA):-
```

```
    !,
    inst(A,MSK,PA).

inst(A,MSK,PA):-
    inst0(A,MSK,PA).

inst0(A,MSK,[REF]):- recorded(MSK,ms(REF,A,_,_,_)).
inst0(A,_,[REF]):- recorded(state_to_connect,(REF,A,_)).

formula_items(A,_,_):-
    var(A),
    !,
    fail.

formula_items(A and B,MSK,PAB):-
    !,
    formula_items(A,MSK,PA),
    formula_items(B,MSK,PB),
    union(PA,PB,PAB).

formula_items(A or B,MSK,PAB):-
    !,
    formula_items(A,MSK,PA),
    formula_items(B,MSK,PB),
    union(PA,PB,PAB).

formula_items(-A,MSK,PA):-
    !,
    formula_items(A,MSK,PA).

formula_items(A,MSK,[REFA]):-
    (   (   recorded(MSK,ms(REFA,A,_,_,_))
        ;   recorded(state_to_connect,(REFA,A,_))
        ;   recorded(state_to_eventually_connect,(REFA,A,_))
        )
    *-> true
    ;   gen_ref(REFA),
        recordz(state_to_eventually_connect,(REFA,A,0.5))
    ).
```

Annexe C

Quelques prédicats utiles en Prolog

Cette partie présente des prédicats d'intérêt général en Prolog. Tous ces prédicats sont regroupés dans le fichier cd-tools.pl à la fin de cette annexe.

C.1 Traitement des erreurs

Le système de traitement des erreurs proposé est très simple. On considérera deux types d'erreurs. Les erreurs graves (error/[1,2]) mettent fin au programme. Les erreurs moins importantes (warning/[1,2]) se contentent d'afficher un message et le programme peut se poursuivre. Les deux prédicats prennent en paramètre un message d'erreur et éventuellement une liste d'arguments (voir le prédicat writef/2).

Pour simplifier le contrôle des erreurs, on utilisera aussi le prédicat check/[2,3]. Ce prédicat vérifie d'abord qu'une certaine condition est satisfaite. Si ce n'est pas le cas, il génère une erreur.

SWI-Prolog propose un mécanisme d'exceptions. Cela peut être pratique, mais parfois on préférerait que certaines exceptions soient traitées comme des échecs de prédicats (par exemple l'absence d'un prédicat). Le prédicat noex/3 intercepte toutes les exceptions et les traduits en échec.

Exemples :

```
error(" Fichier introuvable ").
warning(" Mauvaise valeur : %t ",[X]).
chk(N<10, " Entier trop grand : %t ",[X]).
noex((p(X,Y),q(X,Y))).
```

C.2 Bases de données

Voici quelques prédicats à utiliser en complément des bases de données de Prolog.

SWI-Prolog offre 3 types de bases de données. Nous ne considérerons ici que l'espace des clauses Prolog et le mécanisme record. Dans la distribution originale de Prolog, il manque des prédicats pour la remise à zéro d'une base de données. Pour cela, on introduit deux prédicats : empty_db/1 efface toutes les entrées de la base de données enregistrées sous une clé donnée et empty_module/1 supprime toutes les clauses contenues dans un module particulier.

Exemple :

```
empty_db(ma_cle).
empty_module(mon_module).
```

Un module doit pouvoir être réinitialisable. Un module réinisialitable contient un prédicat dynamique (init/0). Pour réinitialiser tous les modules, on appelle le prédicat reinit/0. Ce predicat extrait toutes les clauses init/0 de tous les modules et les appelle.

Exemple :

```
% dans un module
init :-empty_db(cle).

% module principal
reinit.
```

La puissance de Prolog vient de l'unification et du backtrack. Cependant tous les prédicats ne sont pas défaisables. C'est le cas en particulier de assert et retract. Parfois on a besoin de supprimer des clauses qui ont été assertée dans une branche qui échoue. Les prédicats rasserta/1, rassertz/1, rretracta/1, rretractz/1 possèdent un tel comportement.

Exemple :

```
?-rassertz(foo),fail ;foo.

No
```

« foo » est asserté puis rétracté.

C.3 Interface utilisateur

Pour tester nos programmes, nous avons besoin d'une interface utilisateur simple. L'interface utilisée est textuelle.

Le prédicat menu/3 définit un menu textuel. Chaque entrée du menu est associée à une action qui sera exécutée lorsque l'entrée sera sélectionnée.

Exemple :

```
menu(" Main menu ",
    [   " f " :file :file_menu,
        " e " :edit :edit_menu,
        " q " :quit :EXIT=true
    ],
    " Enter your choice : "
).
```

Il y a aussi un prédicat pour la saisie au clavier : readln/1. Ce prédicat lit une chaine de caractères et construit un atome dont le nom est la chaine saisie.

Exemple :

```
readln(STR).
```

C.4 Tri

Pour trier des listes, on utilisera le prédicat merge_sort/3. Ce prédicat est générique et a besoin d'une relation d'ordre pour fonctionner. Il est équivalent au prédicat predsort/3 de SWI-Prolog.

Exemple :

```
cmp(X:_,Y:_):-X<Y.

?-merge_sort(cmp,[1:a,3:b,2:d],S).

L=[1:a,2:d,3:b]
```

C.5 Structures de contrôle

Prolog offre de nombreuse structures de contrôle (cut, soft-cut, repeat, fail, ...). Il manque une boucle de type « for ». Le prédicat for/4 est limité aux valeurs entières et aux incréments de 1.

Exemple :

```
for(I,1,10,
    (   foo(I),
        print(I)
    )
)
```

Une autre structure de contrôle inspirée des macros en C permet d'activer ou non certaines parties du code. On peut définir un symbole et eventuellement lui associer une valeur (define/1), puis demander si un symbole est défini ou lire sa valeur (ifdef/1). Un symbole peut aussi être redéfini (redefine/1) ou indéfini (undefine/1).

C.6 Unification

SWI-Prolog, comme beaucoup d'autres compilateurs Prolog, est basé sur une unification rapide dépourvue du test d'occurrence. Le temps d'exécution est alors bien meilleur et il est possible d'utiliser des termes cycliques. Mais parfois on a besoin d'éviter les termes cycles. Une solution simple consiste à redéfinir l'unification. Le prédicat '*=' /2 effectue une unification propre avec le contrôle d'occurrence de ces deux arguments. On définit de même une semi-unification ('=>'/2) et une semi-unification avec test d'occurrence ('*=>'/2). La semi-unification X=>Y signifie que l'on va essayer d'instancier X de manière à ce qu'il s'unifie avec Y, sans modifier Y.

Exemple :

```
?-X=f(X).
...boucle infinie...

?-X*=f(X).

No

?-f(X,y)=>f(X,Y).

No

?-f(X,Y)=>f(Z,y).

X=Z
Y=Z
Yes
```

C.7 Substitution de termes

La substitution est un problème classique en informatique. Nous l'utilisons dans notre simulateur de dialogue et dans le PTMS pour traiter les descriptions des jeux de dialogue et les formules logiques. Malgré sa simplicité, il permet de réduire le nombre de lignes de Prolog en écrivant les règles de substitution de manière simple et compacte. Les règles sont de plus associées à des conditions qui dictent l'applicabilité des règles. Le prédicat de substitution a été écrit en SWI-Prolog et devrait être facilement portable sans modification.

C.7.1 Formalisation des substitutions

L'objectif est de pouvoir réécrire un terme en un autre terme. Pour cela nous avons besoin de la description des termes avant et après la traduction. Une règle de réécriture est de la forme `LHS >>> RHS`. LHS est un sous terme du terme à traduire. RHS est la traduction de LHS. Parfois la traduction dépend des instances du terme à traduire. Par exemple, on peut avoir besoin de faire certains calculs ou de filtrer certains termes avant la traduction. Pour cela, on rajoute à ces règles une condition. Cette condition est un programme Prolog qui exprime soit une simple condition, soit un calcul de la traduction. Une règle de réécriture sera donc de la forme `LHS >>> RHS if COND`. Pour lever l'ambiguïté lorsque deux règles ou plus sont applicables, on suppose que c'est la première qui doit l'être (ce choix est courant en Prolog). Une substitution est un ensemble ordonné de règles de réécriture.

Considérons l'exemple suivant. Soit une expression arithmétique à simplifier. Cette expression peut contenir des nombres et des variables. On utilise la substitution suivante :

```
(X+Y)*Z >>> X*Z+Y*Z
Z*(X+Y) >>> (X+Y)*Z
X+(Y+Z) >>> (X+Y)+Z
1*X >>> X
X*1 >>> X
X+Y >>> Z if (number(X), number(Y), Z is X+Y)
X*Y >>> Z if (number(X), number(Y), Z is X*Y)
X+Y >>> Y+X if (number(Y), \+ number(X))
X*Y >>> Y*X if (number(Y), \+ number(X))
```

La traduction du terme `(1+x+2)*(2+y+3)` est le terme `15+3*y+5*x+y*x`.

C.7.2 Algorithme

A partir d'un terme, l'algorithme produit un autre terme. L'algorithme utilisé est le suivant :

- Appliquer la substitution à chacun des arguments du terme
- Chercher la première règle applicable à ce nouveau terme dont la condition est satisfaite
- Si une telle règle existe, l'appliquer et réappliquer la substitution au nouveau terme
- Sinon, retourner le terme courant

Lors de l'application d'une règle, deux termes se ressemblent s'ils s'unifient et si le terme provenant de la règle n'instancie pas le sous terme du terme à traduire (sinon la règle de substitution modifierait le terme original). Cela signifie que la règle `f(x,y) >>> g(u,v)` ne doit pas être appliquée à la variable non instanciée X. X doit restée non instanciée pour ne pas créer le sous terme g(u,v) dans le terme courant. Au contraire, une variable provenant d'une règle de réécriture doit pouvoir être instanciée.

C.7.3 Implémentation

Le programme utilise deux prédicats : un pour la substitution (subst/3) et un pour la reconnaissance de motifs (match/2).

Pour écrire des règles de réécriture, on a besoin des opérateurs '>>>' (règles) et '`if`' (conditions).

```
:-op(950,xfx,>>>).
:-op(949,xfx,if).
```

Substitution

Le prédicat subst(S,T,TS) réussit si TS est le résultat de l'application de la substitution S au terme T.

Lorsque la partie droite d'une règle de réécriture est une variable non instanciée, le résultat de la substitution est la variable originale et non sa copie. Dans les autres cas, on travaille sur une copie de la règle pour éviter de modifier la substitution (en la sur-instanciant).

Reconnaisance de motifs

Le prédicat match(X,Y) réussit si X s'unifie avec Y mais n'instancie aucune variable de Y.

Remarque

Le système décrit ici permet de faire des substitutions en Prolog de manière simple et compacte. Le prédicat subst/3 est très général. Cela explique sa lenteur dans le cas de substitutions simples qui ne nécessitent pas la réapplication des règles lorsqu'une règle vient d'être appliquée.

C.8 Sources

```
/*****************************************************************************/
/*                                                                           */
/* Quelques predicats utiles en Prolog                                       */
/*                                                                           */
/*****************************************************************************/

:-module('cd-tool',
    [
        error/1, error/2,
        warning/1, warning/2,
        chk/2, chk/3,
        noex/1,

        empty_db/1,
        empty_module/1,
        reinit/0,
        gen_ref/1,
        rasserta/1, rassertz/1,
        rretracta/1, rretractz/1,

        member/3,

        op/1,

        menu/3,
        readln/1,

        merge_sort/3,

        for/4,
        define/1, redefine/1, undefine/1,
        ifdef/1, ifdef/2, ifndef/1, ifndef/2, ifdef/3, ifndef/3,

        '*='/2,             % unification avec controle d'occurrence
        '=>'/2,             % semi unification
        '*=>'/2,            % semi unification avec controle d'occurrence

        subst/3,
```

```
        pwritef/2
    ]
).

/****************************************************************************/
/*                                                                          */
/* Messages d'erreur et avertissement, tests de condition                   */
/*                                                                          */
/****************************************************************************/

error(MSG):-error(MSG,[]).
error(MSG,ARGS):-
    writef("ERROR : "), writef(MSG,ARGS), nl,
    abort.

warning(MSG):-warning(MSG,[]).
warning(MSG,ARGS):-
    writef("WARNING : "), writef(MSG,ARGS), nl.

:-module_transparent chk/2, chk/3.

/* Si COND echoue, chk emet un message d'erreur
 */

chk(COND,MSG):-chk(COND,MSG,[]).
chk(COND,MSG,ARGS):-
    (   COND
    ;   error(MSG,ARGS)
    ),
    !.

/****************************************************************************/
/*                                                                          */
/* Suppression des exceptions                                               */
/*                                                                          */
/****************************************************************************/

noex(GOAL):-catch(GOAL,_,fail).

/****************************************************************************/
/*                                                                          */
/* Remise a zero d'une base de donnees                                      */
/*                                                                          */
/****************************************************************************/

empty_db([]):-!.
empty_db([K|KS]):-!,empty_db(K),empty_db(KS).
empty_db(K):-forall(recorded(K,_,REF),erase(REF)).

/****************************************************************************/
/*                                                                          */
/* Remise a zero d'un module                                                */
/*                                                                          */
```

```
/****************************************************************************/

empty_module([]):-!.
empty_module([M|MS]):-!,empty_module(M),empty_module(MS).
empty_module(M):-forall(clause(M:_,_,REF),erase(REF)).

/****************************************************************************/
/*                                                                          */
/* Menu textuel                                                             */
/*                                                                          */
/****************************************************************************/

:-module_transparent menu/3.

menu(TITLE,OPTIONS,PROMPT):-
    writef("\n%s\n\n",[TITLE]),
    forall(member(CH:OPT:_,OPTIONS),
        (   writef("%s : ",[CH]),
            (   OPT=(FORMAT,ARGS)
            ->  writef(FORMAT,ARGS)
            ;   writef(OPT)
            ),
            nl
        )
    ),
    writef("\n%s",[PROMPT]),
    repeat,
        get_single_char(C),
    memberchk([C]:_:GOAL,OPTIONS),
    writef("%s\n",[[C]]),
    (   GOAL
    ;   warning("The command failed")
    ),
    !.

/****************************************************************************/
/*                                                                          */
/* Saisie d'une ligne de texte                                              */
/*                                                                          */
/****************************************************************************/

readln(LN):-
    readln([],LN),
    !.
readln(LN0,LN):-    % LN0 : already read text
    get_single_char(CH),
    (   CH==8   ->  (   LN0=[_|LN1], writef([8,32,8])
                    ;   LN1=[]
                    ),
                    readln(LN1,LN)
    ;   CH>=32  ->  writef("%s",[[CH]]),
                    readln([CH|LN0],LN)
    ;   (   CH==13
        ;   CH==10
```

```
        )       ->  reverse(LN0,LIST),
                    name(LN,LIST)
    ;   readln(LN0,LN)
    ).

/****************************************************************************/
/*                                                                          */
/* Reinitialisation de module                                               */
/*                                                                          */
/****************************************************************************/

:-module_transparent
    reinit/0,
    reinit/1.

reinit:-
    findall(M:INIT,
        (   current_module(M),
            clause(M:init,INIT)
        ),
        INITS
    ),
    reinit(INITS).

reinit([]):-!.
reinit([INIT|INITS]):-
    (   memberchk(INIT,INITS)
    ;   INIT
    ),
    !,
    reinit(INITS).

/****************************************************************************/
/*                                                                          */
/* Generateur de references uniques                                         */
/*                                                                          */
/****************************************************************************/

init:-flag(gen_ref,_,0).

gen_ref(R):-flag(gen_ref,R,R+1).

/****************************************************************************/
/*                                                                          */
/* Tri par fusion                                                           */
/*                                                                          */
/****************************************************************************/

/* Tri la liste L selon l'ordre defini par le predicat CMP_FUNC
 * CMP_FUNC doit definir un ordre ('<' par exemple)
 */

:-module_transparent merge_sort/3, merge__go/4, merge_2/3, cmp/2.
```

```
merge_sort(CMP_FUNC,L,S):-
    asserta(('cd-tools':cmp(X,Y):-!,call(CMP_FUNC,X,Y)),REF),
    length(L,N),
    merge__go(N,L,S,[]),    % tri les N premiers elem de L
    erase(REF).

merge__go(0,L,[],L):-!.         % rien a trier
merge__go(1,[X|L],[X],L):-!.    % tri un seul element
merge__go(N,L,S,R):-            % tri N elements
    N1 is N>>1,     merge__go(N1,L,S1,R1),  % tri la premiere moitier
    N2 is N-N1,     merge__go(N2,R1,S2,R),  % puis la seconde
    merge_2(S2,S1,S).           % et fusionne les deux sous listes

merge_2([],S,S):-!.
merge_2([X|L1],[Y|L2],[X|L]):-
    'cd-tools':cmp(X,Y),
    !,
    merge_2(L1,[Y|L2],L).
merge_2(L1,[Y|L2],[Y|L]):-
    merge_2(L2,L1,L).

/******************************************************************************/
/*                                                                            */
/* boucles                                                                    */
/*                                                                            */
/******************************************************************************/

:-module_transparent for/4.

for(I,MIN,MAX,BODY):-
    VMIN is MIN,
    VMAX is MAX,
    between(VMIN,VMAX,I),
        BODY,
    fail.
for(_,_,_,_).

/******************************************************************************/
/*                                                                            */
/* assert et retract reversibles                                              */
/*                                                                            */
/******************************************************************************/

/* Le backtrack sur ces predicats supprime leurs effets
 */

:-module_transparent rasserta/1, rassertz/1, rretracta/1, rretractz/2.

rasserta(X):-
    asserta(X,REF),
    (   true
    ;   erase(REF), fail
    ).
```

```
rassertz(X):-
    assertz(X,REF),
    (   true
    ;   erase(REF), fail
    ).

rretracta(X):-
    retract(X),
    (   true
    ;   asserta(X), fail
    ).

rretractz(X):-
    retract(X),
    (   true
    ;   assertz(X), fail
    ).


/****************************************************************************/
/*                                                                          */
/* member avec extraction                                                   */
/*                                                                          */
/****************************************************************************/

member(E,[E|L],L).
member(E,[F|L1],[F|L2]):-member(E,L1,L2).


/****************************************************************************/
/*                                                                          */
/* Directives d'execution                                                   */
/*                                                                          */
/****************************************************************************/

:-module_transparent ifdef/2, ifdef/3, ifndef/2, ifndef/3.

define(A=X):-
    !,
    (   ifdef(A)
    ->  warning("%t is redefined.",[A]),
        undefine(A)
    ;   true
    ),
    recordz(A,X).
define(A):-
    define(A=true).

redefine(A):-
    undefine(A),
    define(A).

ifdef(A=X):-!, recorded(A,X).
ifdef(A):-recorded(A,_).
```

```
ifdef(A,GOAL):-ifdef(A), !, GOAL.
ifdef(_,_).
ifdef(A,GOAL,_):-ifdef(A), !, GOAL.
ifdef(_,_,GOAL):-GOAL.

ifndef(A=X):-!, \+recorded(A,X).
ifndef(A):- \+recorded(A,_).
ifndef(A,GOAL):-ifndef(A), !, GOAL.
ifndef(_,_).
ifndef(A,GOAL,_):-ifndef(A), !, GOAL.
ifndef(_,_,GOAL):-GOAL.

undefine(A=X):-!, forall(recorded(A,X,REF),erase(REF)).
undefine(A):-forall(recorded(A,_,REF),erase(REF)).

/****************************************************************************/
/*                                                                          */
/* Unification avec occur check                                             */
/*                                                                          */
/****************************************************************************/

/*  occur_check     : controle d'occurrence d'une variable dans un terme
 *                    X=f(X) echoue
 *  semi unification    : X => Y "X s'unifie a Y"
 *                        La semi unification de X vers Y ne modifie que X
 */

:-op(700,xfx,[*=,*=>,=>]).
:-op(700,xfx,has_no).

/* Unification avec controle d'occurrence
 */

X *= Y :-
    (   var(X)
    ->  (   var(Y)
        ->  X=Y
        ;   Y has_no X,
            X=Y
        )
    ;   (   var(Y)
        ->  X has_no Y,
            X=Y
        ;   functor(X,F,N),
            functor(Y,F,N),
            unify_args(N,X,Y)
        )
    ).

/* Semi-unification avec controle d'occurrence
 */

X *=> Y :-
    (   var(X)
```

```
    ->  (   var(Y)
        ->  X=Y
        ;   Y has_no X,
            X=Y
        )
    ;   nonvar(Y),
        functor(X,F,N),
        functor(Y,F,N),
        half_unify_args(N,X,Y)
    ).

/* Semi-unification rapide (sans controle d'occurrence)
 */

X => Y :-
    (   var(X)
    ->  X=Y
    ;   nonvar(Y),
        functor(X,F,N),
        functor(Y,F,N),
        quick_half_unify_args(N,X,Y)
    ).

TERM has_no VAR :-
    (   var(TERM)
    ->  TERM\==VAR
    ;   functor(TERM,_,N),
        has_no(N,TERM,VAR)
    ).

has_no(0,_,_):-!.
has_no(N,TERM,VAR):-
    arg(N,TERM,ARG),
    ARG has_no VAR,
    N1 is N-1,
    has_no(N1,TERM,VAR).

unify_args(0,_,_):-!.
unify_args(N,X,Y):-
    arg(N,X,XN),
    arg(N,Y,YN),
    XN*=YN,
    N1 is N-1,
    unify_args(N1,X,Y).

half_unify_args(0,_,_):-!.
half_unify_args(N,X,Y):-
    arg(N,X,XN),
    arg(N,Y,YN),
    XN*=>YN,
    N1 is N-1,
    half_unify_args(N1,X,Y).

quick_half_unify_args(0,_,_):-!.
```

```
quick_half_unify_args(N,X,Y):-
    arg(N,X,XN),
    arg(N,Y,YN),
    XN=>YN,
    N1 is N-1,
    quick_half_unify_args(N1,X,Y).

/******************************************************************************/
/*                                                                            */
/* Substitution de termes                                                     */
/*                                                                            */
/******************************************************************************/

:-op(950,xfx,>>>).
:-op(949,xfx,if).

:-module_transparent subst/3, subst_args/4.

subst(S,T,TS):-
    functor(T,F,N),
    functor(TS0,F,N),
    subst_args(N,S,T,TS0),
    (   member(LHS0>>>RHS0,S),
        copy_term(LHS0>>>RHS0,LHS>>>RHS),
        match(LHS,TS0),
        (   var(RHS)
        ->  TS=RHS0
        ;   RHS=(TS1 if COND)
        ->  COND,
            (   var(TS1)
            ->  RHS0=(TS0 if _),
                TS=TS0
            ;   subst(S,TS1,TS)
            )
        ;   subst(S,RHS,TS)
        )
    ;   TS=TS0
    ),
    !.
subst(_,X,X).

subst_args(0,_,_,_):-!.
subst_args(N,S,T,TS):-
    arg(N,T,A),
    arg(N,TS,AS),
    subst(S,A,AS),
    N1 is N-1,
    subst_args(N1,S,T,TS).

match(X,Y):-
    (   var(X)
    ->  X=Y
    ;   nonvar(Y),
        functor(X,F,N),
```

```
        functor(Y,F,N),
        match_args(N,X,Y)
    ).

match_args(0,_,_):-!.
match_args(N,X,Y):-
    arg(N,X,AX),
    arg(N,Y,AY),
    match(AX,AY),
    N1 is N-1,
    match_args(N1,X,Y).


/****************************************************************************/
/*                                                                          */
/* Depth                                                                    */
/*                                                                          */
/****************************************************************************/

depth(T,D):-
    functor(T,_,N),
    !,
    max_depth(N,T,0,DA),
    D is DA+1.

depth(_,0).

max_depth(0,_,D,D):-!.
max_depth(N,T,D1,D2):-
    arg(N,T,A),
    depth(A,D),
    N1 is N-1,
    (   D>D1
    ->  max_depth(N1,T,D,D2)
    ;   max_depth(N1,T,D1,D2)
    ).

/****************************************************************************/
/*                                                                          */
/* Element maximal d'une liste de reels positifs                            */
/*                                                                          */
/****************************************************************************/

max([],0):-!.
max([E|L],M):-max(L,N), M is max(E,N).

/****************************************************************************/
/*                                                                          */
/* writef + instanciation des variables                                     */
/*                                                                          */
/****************************************************************************/

pwritef(MSG,ARGS):-
    copy_term(ARGS,TMP),
```

```
    numbervars(TMP,0,_),
    writef(MSG,TMP).

/*****************************************************************************/
/*                                                                           */
/* Version simplifiee pour les appels multiples a op/3                       */
/*                                                                           */
/*****************************************************************************/

op([]):-!.
op([PRECEDENCE,TYPE,OP|OPS]):-
    op(PRECEDENCE,TYPE,OP),
    op(OPS).
```

Annexe D

Traitement du langage naturel

Les dialogues générés jusqu'à présent utilisent un langage formel. Pour une utilisation avec des agents humains, il faut utiliser le langage naturel. Le langage naturel pose principalement deux problèmes : l'analyse et la génération. Nous allons voir ici deux méthodes d'analyse du langage naturel : Grammaires lexicales fonctionnelles et Grammaires catégorielles (Lambek, Montague).

D.1 Grammaires lexicales fonctionnelles

Les grammaires lexicales fonctionnelles sont présentées en détail dans (Evrard, 1998a). Nous ne présenterons ici qu'une implémentation de ces grammaires.

D.1.1 Représentation de la structure de la phrase : les frames

Les frames seront représentés par une liste de couples(nom d'attribut, valeur d'attribut), la valeur d'un attribut pouvant être elle-même un frame (une phrase est en effet une structure hiérarchisée). Pour permettre la mise à jour des frames, la queue de la liste est une variable Prolog non instanciée, l'ajout de nouveaux attributs se faisant en instanciant cette variable. Un frame vide sera donc représenté par une variable. L'implantation ne doit pas contraindre l'ordre des attributs, cela simplifie la constitution du lexique.

Construction des frames

Les frames sont construites pendant l'analyse de la phrase par les règles de productions (voir plus loin). Les frames initiaux sont ceux des entrées du lexique, puis par unifications successives, on construit le frame correspondant à la phrase de départ. L'action essentielle sur les frames est donc l'unification, c'est-à-dire la réunion ensembliste de deux frames, à condition que les attributs de même nom aient la même valeur, ou deux valeurs unifiables. L'unification est réalisée par l'opérateur `<-/2`. `F <- G` signifie que F est unifié avec G.

L'algorithme utilisé est le suivant :

Si G est vide, on lie F et G à l'aide de l'unification de Prolog.

```
F <- G          :- var(G), !, F=G.
```

Si F est vide, il faut le créer. Il y a deux possibilités, soit G est un attribut, auquel cas on doit créer la liste, soit G est déjà un frame, il suffit alors d'unifier (au sens Prolog) F et G.

```
F <- G          :- var(F), !, (G=(_=_) -> F=[G|_] ; F=G).
```

Sinon, on est dans le cas où F est déjà construit et G est une liste d'attributs. Il faut donc parcourir G et ajouter chaque attribut de G à F.

```
F <- [A=V|G]    :- !, F <- A=V, (var(G) -> true ; F<-G).
```

Il faut maintenant traiter le cas où l'on ajoute un attribut dans F. Soit cet attribut est déjà présent, dans ce cas les valeurs associées à ces attributs doivent être unifiables, soit cet attribut est nouveau et sera ajouté dans F par la deuxième règle. On parcourt donc le frame F jusqu'à trouver l'attribut à ajouter ou arriver en fin de la liste.

```
[A=V1|_] <- A=V2    :- !, V1<-V2.
[_|F] <- A=V        :- !, F<-A=V.
```

Dans tous les autres cas, il s'agit d'unifier des valeurs d'attributs qui ne sont pas des frames. A ce niveau, c'est l'unification de Prolog est utilisée.

```
F <- F.
```

Lors de l'unification, il faut tenir compte d'un cas particulier. En effet, dans le formalisme LFG, on prend en compte les liaisons longue distance. Ces liaisons sont mémorisées dans les frames sous forme d'une liste de frames. Dans ce cas, les frames ne doivent pas être unifiés mais concaténés (en supprimant les redondances). On rajoute donc après les trois premières règles les règles suivantes.

```
[far_up=FAR_UP|_] <- far_up=UP          :- !, FAR_UP+=UP.
[far_down=FAR_DOWN|_] <- far_down=DOWN  :- !, FAR_DOWN+=DOWN.
```

Dans ces règles, l'opérateur `+=` effectue la concaténation.

Résolution des équations associées aux règles

Les équations associées aux règles de grammaire permettent l'unification de frames ou de sous frames. Il faut donc pouvoir spécifier les frames à extraire puis les unifier. On utilise la syntaxe `F:[liste d'attributs]`. Par exemple, `F:[sujet,predicat]` signifie que l'on utilise la valeur du champ predicat du frame contenu dans le champ sujet du frame `F`. On introduit alors l'opérateur `@=` qui permet de parcourir la liste des attributs et d'extraire le bon frame. Les équations s'écriront de la manière suivante :

`F:[liste]@=G :[liste]`, sachant que la liste est facultative (`F` est équivalent à `F:[]`)

D.1.2 Grammaire lexicale fonctionelle

Règles de grammaire

Chaque prédicat de l'analyseur syntaxique possède un argument qui est le frame construit au cours de l'analyse. L'implémentation choisie permet de simplifier l'écriture des règles. Ainsi, on écrit pour chaque règle de grammaire une règle Prolog de manière très concise (voir tableau D.1).

S	$\rightarrow$	NP $\uparrow Subject =\downarrow$	VP $\uparrow=\downarrow$		
S	$\rightarrow$	NP $\uparrow=\Downarrow NP$ $(\downarrow qform) = +$	AUX	NP $\uparrow Subject =\downarrow$	VP $\uparrow=\downarrow$
NP	$\rightarrow$	$Determiner$	$Noun$		
NP	$\rightarrow$	$Pronoun$			
NP	$\rightarrow$	ϕ $\uparrow=\Uparrow NP$			
VP	$\rightarrow$	$Verb$	NP $\uparrow Object =\downarrow$	NP $\uparrow Object - 2 =\downarrow$	
VP	$\rightarrow$	$Verb$	NP $\uparrow Object =\downarrow$		
VP	$\rightarrow$	$Verb$	VP $\uparrow Vcomp =\downarrow$		
VP	$\rightarrow$	$Particle$ $\uparrow ptrtype = to$	VP $\uparrow=\downarrow$		

TAB. D.1: Grammaire LFG

Lexique

On définit le lexique de la même manière. Par exemple, le verbe « promise » aura la définition suivante :

$$\begin{array}{lll} verb & promise & \uparrow predicate = promise(\uparrow subject, \uparrow object, \uparrow vcomp) \\ & & \uparrow vcompsubject = \uparrow subject \\ & & \uparrow format = (subject, -, object, vcomp) \end{array}$$

On remarque que l'on peut de manière très simple réaliser les liaisons entre le sujet de promise et le sujet de l'infinitive qui complète le verbe. On peut ainsi définir des liaisons différentes selon les verbes (pour le verbe persuade, le sujet de l'infinitive est le complément du verbe et non son sujet).

D.1.3 Les différentes étapes de l'analyse

L'analyse comporte deux étapes. La première est l'analyse syntaxique proprement dite. Elle est réalisée par les règles décrites ci-dessus. Une fois que le frame est construit, il faut encore effectuer les liaisons longue distance. Pour cela, le prédicat `make_far_links/2` parcourt le frame et construit la liste de ces liaisons. Ensuite les liaisons vers le haut et vers le bas sont unifiées deux par deux. On se reportera au code Prolog pour les détails de l'implémentation.

D.1.4 Programme

Gestion des frames et unification

```
/******************************************************************************/
/*                                                                            */
/* Christophe DELORD                                                          */
/*                                                                            */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                   */
/*                                                                            */
/******************************************************************************/

:-module(lfg,
    [
        '@='/2,                 % Operateur d'unification pour les equations
        make_far_links/2,       % Connexion des liens a longue distance
        afficher_frame/1        % Affichage du frame produit par l'analyseur
    ]
).

:-op(800,xfx,<-).       % Unification de frames
:-op(800,xfx,@=).       % Unification de sous-frames
:-op(800,xfx,+=).       % Concatenation de listes terminees par une variable

/******************************************************************************/
/*                                                                            */
/* Representation des frames                                                  */
/*                                                                            */
/******************************************************************************/

/* Les frames sont representes par des listes dont la queue est une variable.
 * Cela permet de faire les mises a jour des frames directement dans le meme
 * frame et simplifie l'ecriture des equations associees aux regles
 */
```

```
/****************************************************************************/
/*                                                                          */
/* Equations LFG                                                            */
/*                                                                          */
/****************************************************************************/

% F et G sont de la forme : frame:liste_de_champs

F @= G :-
    extract(F,F1,OP),       % OP = <- ou +=
    extract(G,G1,_),        % On extrait les deus sous-frames a unifier
    ( OP='<-'
    ->
        F1<-G1, G1<-F1      % Unification des frames
    ;
        F1+=[G1|_]          % Memorisation des liages longue distance
    ).

extract(F,F,'<-'):-         % Extraction d'une variable : '<-' = unification
    var(F),
    !.

extract(F:[far_up],F1,'+='):-   % Extraction de la liste des liasons longues
    !,                          % '+=' = concatenation (ensembliste)
    F<-far_up=F1.

extract(F:[far_down],F1,'+='):-
    !,
    F<-far_down=F1.

extract(F:[X|S],F1,OP):-        % Parcours du chemin d'acces
    !,
    F<-X=Y,                     % On cherche (ou on cree) le frame de nom X
    extract(Y:S,F1,OP).         % Puis on poursuit la recherche sur ce frame

extract(F:[],F,'<-'):-          % Chemin d'acces vide => on renvoie F
    !.

extract(F,F1,'<-'):-            % Pour le cas des predicats
    F=..[FCT|ARGS], FCT\='.', FCT\=',', FCT\=':', ARGS\=[],
    !,
    maplist(extract,ARGS,N_ARGS),   % on traite les arguments
    F1=..[FCT|N_ARGS].              % et on le reconstruit

extract(F,F,'<-').              % Dans les autres cas, rien a faire

extract(F,G):-extract(F,G,_).

/****************************************************************************/
/*                                                                          */
/* Unification                                                              */
/*                                                                          */
/****************************************************************************/
```

```
/* Cas particuliers ou les frames sont vides */

% Si G est vide, on le lie avec F
F <- G          :- var(G), !, F=G.

% Si F est vide, on le cree avec G.
% Si G n'est pas un frame, il faut creer la liste : F=[G|_]
F <- G          :- var(F), !, (G=(_=_) -> F=[G|_] ; F=G).

% On parcous le frame G, et on insere chaque couple attribut=valeur dans F
F <- [A=V|G]    :- !, F <- A=V, (var(G) -> true ; F<-G).

% Cas particulier des liasons longue distance : concatenation
[far_up=FAR_UP|_] <- far_up=UP          :- !, FAR_UP+=UP.
[far_down=FAR_DOWN|_] <- far_down=DOWN  :- !, FAR_DOWN+=DOWN.

% Unification de deux attributs deja presents : on unifie les valeurs
[A=V1|_] <- A=V2    :- !, V1<-V2.

% Tant que l'attribut n'est pas trouve, on continue sur la suite du frame
[_|F] <- A=V        :- !, F<-A=V.

F <- F.

/****************************************************************************/
/*                                                                          */
/* Ajout d'une liste de dependances a distance a la fin d'une autre         */
/*                                                                          */
/****************************************************************************/

/* ca marche pas : [X|Y]+=[X|Y] => bug
_ += L2         :- var(L2), !.
L1 += L2        :- var(L1), !, (is_list(L2) -> L1=L2; L1=[L2|_]).
[_|L1] += L2    :- L1 += L2.
*/

L1 += L2 :-             % pour rajouter L2 a la fin de L1
    filtre(L1,L2,L3),   % L3 = liste des elements de L2\L1
    cat(L1,L3).         % On rajoute L3 a L1

:-op(800,xfx,has).

% Construit la liste L3=L1\L2
% Une variable de L2 n'est dans L1 que si c'est la MEME variable
% Ceci permet d'avoir plusieurs fois la meme liaison a distance
% mais de ne la prendre en compte qu'une seule fois
filtre(_,L2,_):-var(L2),!.
filtre(L1,[X|L2],L3):-L1 has X, !, filtre(L1,L2,L3).
filtre(L1,[X|L2],[X|L3]):-filtre(L1,L2,L3).

cat(_,L3):-var(L3),!.           % L3 vide : rien a rajouter
cat(L1,L3):-var(L1),!,L1=L3.    % L1 vide, on rajoute L3
cat([_|L1],L3):-cat(L1,L3).     % Parcours L1 pour atteindre la queue
```

```
% has se comporte comme member, mais traite en plus le test d'egalite
% de deux variables SANS LES UNIFIER, ce qui est essentiel pour les
% liaisons longue distance
L has _ :- var(L), !, fail.
[X|_] has Y :- free_variables([X,Y],[X]), !.
[_|L] has Y :- L has Y.

/*****************************************************************************/
/*                                                                           */
/* Affichage d'un frame                                                      */
/*                                                                           */
/*****************************************************************************/

afficher_frame(F):-aff(F,5),nl.

aff(F,_):-var(F),!.     % frame vide, on n'affiche rien

aff([],_):-!.           % frame vide 'nettoye'

aff([X|F],I):-          % affichage d'une variable
    var(X),
    !,
    tab(I),
    writef("%t",[X]),
    aff(F,I).

aff([A=V|F],I):-        % affichage d'un couple attribut=valeur
    !,
    nl,
    tab(I),
    writef("%15l = ",[A]),
    (member(A,[predicate,far_up,far_down])
    ->
        print(V)
    ;
        I3 is I+3,
        aff(V,I3)
    ),
    aff(F,I).

aff(V,_):-writef("%t",[V]).

/*****************************************************************************/
/*                                                                           */
/* Dependances a longue distance                                             */
/*                                                                           */
/*****************************************************************************/

/* Ici, les listes des liaisons a longue distance ont ete construites.
 * Il faut maintenant apparier les liaisons vers le haut et vers le bas
 */

/* Pour faire ces liens :
 *      - On extrait les deux listes (liaisons vers le haut et vers le bas)
```

```
 *      - On unifie les elements de ces listes deux par deux
 *      - Et on nettoye le frame
 */
make_far_links(F,CF):-
    get_links(F,FAR_UP,FAR_DOWN),
    make_all_links(FAR_UP,FAR_DOWN),
    clean(F,F,CF).

/* Pour lier les variables deux par deux :
 *      - On prend un element dans chaque liste
 *      - On les unifie
 *      - On continue sur le reste de la liste
 * Cela implique que les deux listes ont la meme longueur. Sinon, la
 * phrase est malformee
 */
make_all_links(FAR_UP,FAR_DOWN):-
    my_member(ONE_FAR,FAR_UP,OTHER_FAR_UP),
    my_member(ONE_FAR,FAR_DOWN,OTHER_FAR_DOWN),
    make_all_links(OTHER_FAR_UP,OTHER_FAR_DOWN).
make_all_links([],[]).

% my_member renvoie en plus la liste privee de l'element
my_member(_,L,_):-var(L),!,fail.
my_member(E,[E|L],L).
my_member(E,[F|L],[F|M]):-my_member(E,L,M).

/* Extraction des liens
 * On parcours le frame et on memorise a chaque etape le contenu des
 * champs far_up et far_down
 */
get_links(F,_,_):-var(F), !.

get_links([far_up=UP|F],FAR_UP,FAR_DOWN):-
    !,
    get_links(F,FAR_UP,FAR_DOWN),
    FAR_UP+=UP.

get_links([far_down=DOWN|F],FAR_UP,FAR_DOWN):-
    !,
    get_links(F,FAR_UP,FAR_DOWN),
    FAR_DOWN+=DOWN.

get_links([_=V|F],FAR_UP,FAR_DOWN):-
    !,
    get_links(V,FAR_UP,FAR_DOWN),
    get_links(F,UP_F,DOWN_F),
    FAR_UP+=UP_F,
    FAR_DOWN+=DOWN_F.

get_links(_,_,_).

/* Nettoyage du frame
 *      - Suppression des champs far_up et far_down
 *      - unification des frames lies a distance
```

```
 */
clean(_,F,F):-var(F),!.

clean(F,[far_down=_|S],CF):-!,clean(F,S,CF).

clean(F,[far_up=[FAR_UP|_]|S],CF):-
    !,
    F<-FAR_UP,
    clean(F,S,CF).

clean(F,[A=V|S],[A=CV|CF]):-
    !,
    clean(V,V,CV),
    clean(F,S,CF).

clean(_,X,X).
```

Exemple de grammaire

```
/****************************************************************************/
/*                                                                          */
/* Christophe DELORD                                                        */
/*                                                                          */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                 */
/*                                                                          */
/****************************************************************************/

:-use_module(lfg).

test:-
    parse([a,girl,handed,the,baby,the,toys]),
    parse([i,want,to,kiss,the,baby]),
    parse([which,baby,did,the,girl,kiss]),
    parse([which,baby,did,the,girl,want,to,kiss]),
    parse([i,promise,the,girl,to,kiss,the,baby]),
    parse([i,persuade,the,girl,to,kiss,the,baby]),
    parse([i,want,the,girl,to,promise,the,boy,to,kiss,the,baby]),
    parse([i,want,the,girl,to,persuade,the,boy,to,kiss,the,baby]),
    parse([the,boy,want,that,the,apple,falls]),
    parse([i,want,that,the,girl,kiss,the,baby]),
    parse([i,want,the,girl,to,kiss,the,baby]),
    parse([i,want,the,girl,to,promise,me,to,kiss,the,baby]),
    parse([which,girl,did,want,me,to,promise,the,boy,to,kiss,the,baby]),
    parse([which,boy,did,the,girl,want,me,to,promise,to,kiss,the,baby]),
    parse([which,baby,did,the,girl,want,me,to,promise,the,boy,to,kiss]),
    parse([which,girl,did,want,me,to,persuade,the,boy,to,kiss,the,baby]),
    parse([which,boy,did,the,girl,want,me,to,persuade,to,kiss,the,baby]),
    parse([which,baby,did,the,girl,want,me,to,persuade,the,boy,to,kiss]),
    parse([which,toys,did,the,girl,want,me,to,handle,the,baby]).

bug2:-
    parse([which,baby,did,promise,me,to,kiss,the,girl]).
```

```
bug3:-
    parse([which,baby,did,promise,me,to,persuade,me,to,kiss,the,girl]).

bug:-parse([which,baby,did,the,girl,want,to,kiss]).

test(FRAME):-
    pS(F,[which,baby,did,the,girl,want,to,kiss],[]),
    make_far_links(F,FRAME),
    afficher_frame(FRAME).

parse(S):-
    writef("%r\n\n",['-',70]),
    print(S), nl,
    pS(F,S,[]),
%   afficher_frame(F), nl,
    make_far_links(F,FRAME),
    afficher_frame(FRAME), nl.

pS(F) -->
    pNP(S),     { F:[subject]@=S },
    pVP(V),     { F@=V }.

pS(F) -->
    pNP(NP),    { F:[far_down]@=NP, NP:[qform]@='+' },
    aux,
    pNP(S),     { F:[subject]@=S },
    pVP(VP),    { F@=VP }.

pNP(F) -->
    determiner(DET),    { F@=DET },
    noun(NOUN),         { F@=NOUN }.

pNP(F) -->
    pronoun(P),         { F@=P }.

pNP(F) -->
    empty(NP),          { F:[far_up]@=NP }.

pVP(F) -->
    verb(V),            { F@=V, V:[format]@=(subject,-) }.

pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,-,object,object2) },
    pNP(O),         { F:[object]@=O },
    pNP(O2),        { F:[object2]@=O2 }.

pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,-,object) },
    pNP(O),         { F:[object]@=O }.

pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,-,vcomp) },
    pVP(VC),        { F:[vcomp]@=VC }.
```

```
pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,-,object,vcomp) },
    pNP(OBJ),       { F:[object]@=OBJ },
    pVP(VCOMP),     { F:[vcomp]@=VCOMP }.

pVP(F) -->
    particle(P),    { P:[ptrtype]@=to },
    pVP(VP),        { F@=VP }.

pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,_,relative) },
    that,
    pS(REL),        { F:[relative]@=REL }.

pVP(F) -->
    verb(V),        { F@=V, V:[format]@=(subject,_,object,vcomp) },
    pNP(OBJ),       { F:[object]@=OBJ },
    particle(P),    { P:[ptrtype]@=to },
    pVP(VP),        { F:[vcomp]@=VP }.

empty(_) --> [].

pronoun(F)-->
    [i],    {   F:[qform]@='-',
                F:[predicate]@=i
            }.

pronoun(F)-->
    [me],   {   F:[qform]@='-',
                F:[predicate]@=i
            }.

pronoun(F)-->
    [he],   {   F:[qform]@='-',
                F:[number]@=singular,
                F:[predicate]@=he
            }.

pronoun(F)-->
    [him],  {   F:[qform]@='-',
                F:[number]@=singular,
                F:[predicate]@=he
            }.

determiner(F)-->
    [a],    {   F:[qform]@='-',
                F:[number]@=singular,
                F:[definiteness]@=indefinite
            }.

determiner(F)-->
    [the],  {   F:[qform]@='-',
                F:[definiteness]@=definite
            }.
```

```
determiner(F)-->
    [which],    {   F:[qform]@='+'
                }.

noun(F)-->
    [girl], {   F:[number]@=singular,
                F:[predicate]@=girl
            }.

noun(F)-->
    [boy],  {   F:[number]@=singular,
                F:[predicate]@=boy
            }.

noun(F)-->
    [baby], {   F:[number]@=singular,
                F:[predicate]@=baby
            }.

noun(F)-->
    [toys], {   F:[number]@=plural,
                F:[predicate]@=toy
            }.

noun(F)-->
    [apple],    {   F:[number]@=singular,
                    F:[predicate]@=apple
                }.

verb(F)-->
    [handed],   {   F:[tense]@=past,
                    F:[predicate]@=hand(F:[subject,predicate],
                                F:[object,predicate],F:[object2,predicate]),
                    F:[format]@=(subject,-,object,object2)
                }.

verb(F)-->
    [handle],   {   F:[predicate]@=hand(F:[subject,predicate],
                                F:[object,predicate],F:[object2,predicate]),
                    F:[format]@=(subject,-,object,object2)
                }.

verb(F)-->
    [kiss],     {   F:[predicate]@=kiss(F:[subject,predicate],
                                F:[object,predicate]),
                    F:[format]@=(subject,-,object)
                }.

verb(F)-->
    [falls],    {   F:[predicate]@=fall(F:[subject,predicate]),
                    F:[format]@=(subject,-)
                }.
```

```
verb(F)-->
    [eats],     {   F:[predicate]@=eats(F:[subject,predicate],
                                 F:[object,predicate]),
                    F:[format]@=(subject,-,object)
                }.

verb(F)-->
    [want],     {   F:[predicate]@=want(F:[subject,predicate],
                                 F:[vcomp,predicate]),
                    F:[vcomp,subject]@=F:[subject],
                    F:[format]@=(subject,-,vcomp)
                }.

verb(F)-->
    [want],     {   F:[predicate]@=want(F:[subject,predicate],
                                 F:[relative,predicate]),
                    F:[format]@=(subject,-,relative)
                }.

verb(F)-->
    [want],     {   F:[predicate]@=want(F:[subject,predicate],
                                 F:[vcomp,predicate]),
                    F:[vcomp,subject]@=F:[object],
                    F:[format]@=(subject,-,object,vcomp)
                }.

verb(F)-->
    [persuade], {   F:[predicate]@=persuade(F:[subject,predicate],
                                 F:[object,predicate],
                                 F:[vcomp,predicate]),
                    F:[vcomp,subject]@=F:[object],
                    F:[format]@=(subject,-,object,vcomp)
                }.

verb(F)-->
    [promise],  {   F:[predicate]@=promise(F:[subject,predicate],
                                 F:[object,predicate],
                                 F:[vcomp,predicate]),
                    F:[vcomp,subject]@=F:[subject],
                    F:[format]@=(subject,-,object,vcomp)
                }.

particle(F)-->
    [to],       {   F:[ptrtype]@=to
                }.

aux-->
    [did].

that-->
    [that].
```

D.1.5 Exemples d'analyses

```
----------------------------------------------------------------------

[a, girl, handed, the, baby, the, toys]

     subject         =
        qform           = -
        number          = singular
        definiteness    = indefinite
        predicate       = girl
     tense           = past
     predicate       = hand(girl, baby, toy)
     object          =
        predicate       = baby
        qform           = -
        definiteness    = definite
        number          = singular
     object2         =
        predicate       = toy
        qform           = -
        definiteness    = definite
        number          = plural
     format          = subject, -, object, object2

----------------------------------------------------------------------

[i, want, to, kiss, the, baby]

     subject         =
        qform           = -
        predicate       = i
     predicate       = want(i, kiss(i, baby))
     vcomp           =
        predicate       = kiss(i, baby)
        subject         =
           predicate       = i
           qform           = -
        object          =
           predicate       = baby
           qform           = -
           definiteness    = definite
           number          = singular
        format          = subject, -, object
     format          = subject, -, vcomp

----------------------------------------------------------------------

[which, baby, did, the, girl, kiss]

     subject         =
        qform           = -
        definiteness    = definite
        number          = singular
```

```
      predicate      = girl
   predicate      = kiss(girl, baby)
   object         =
      predicate      = baby
      qform          = +
      number         = singular
   format         = subject, -, object

----------------------------------------------------------------------

[which, baby, did, the, girl, want, to, kiss]

   subject        =
      qform          = -
      definiteness   = definite
      number         = singular
      predicate      = girl
   predicate      = want(girl, kiss(girl, baby))
   vcomp          =
      predicate      = kiss(girl, baby)
      subject        =
         predicate      = girl
         qform          = -
         definiteness   = definite
         number         = singular
      object         =
         predicate      = baby
         qform          = +
         number         = singular
      format         = subject, -, object
   format         = subject, -, vcomp

----------------------------------------------------------------------

[i, promise, the, girl, to, kiss, the, baby]

   subject        =
      qform          = -
      predicate      = i
   predicate      = promise(i, girl, kiss(i, baby))
   object         =
      predicate      = girl
      qform          = -
      definiteness   = definite
      number         = singular
   vcomp          =
      predicate      = kiss(i, baby)
      subject        =
         predicate      = i
         qform          = -
      object         =
         predicate      = baby
         qform          = -
         definiteness   = definite
```

```
      number          = singular
   format          = subject, -, object
format          = subject, -, object, vcomp

----------------------------------------------------------------------

[i, persuade, the, girl, to, kiss, the, baby]

     subject         =
        qform           = -
        predicate       = i
     predicate       = persuade(i, girl, kiss(girl, baby))
     object          =
        predicate       = girl
        qform           = -
        definiteness    = definite
        number          = singular
     vcomp           =
        predicate       = kiss(girl, baby)
        subject         =
           predicate       = girl
           qform           = -
           definiteness    = definite
           number          = singular
        object          =
           predicate       = baby
           qform           = -
           definiteness    = definite
           number          = singular
        format          = subject, -, object
     format          = subject, -, object, vcomp

----------------------------------------------------------------------

[i, want, the, girl, to, promise, the, boy, to, kiss, the, baby]

     subject         =
        qform           = -
        predicate       = i
     predicate       = want(i, promise(girl, boy, kiss(girl, baby)))
     vcomp           =
        predicate       = promise(girl, boy, kiss(girl, baby))
        subject         =
           qform           = -
           definiteness    = definite
           number          = singular
           predicate       = girl
        object          =
           predicate       = boy
           qform           = -
           definiteness    = definite
           number          = singular
        vcomp           =
           predicate       = kiss(girl, baby)
```

```
            subject         =
               predicate       = girl
               qform           = -
               definiteness    = definite
               number          = singular
            object          =
               predicate       = baby
               qform           = -
               definiteness    = definite
               number          = singular
            format          = subject, -, object
         format          = subject, -, object, vcomp
      object          =
         qform           = -
         definiteness    = definite
         number          = singular
         predicate       = girl
      format          = subject, -, object, vcomp

----------------------------------------------------------------------

[i, want, the, girl, to, persuade, the, boy, to, kiss, the, baby]

      subject         =
         qform           = -
         predicate       = i
      predicate       = want(i, persuade(girl, boy, kiss(boy, baby)))
      vcomp           =
         predicate       = persuade(girl, boy, kiss(boy, baby))
         subject         =
            qform           = -
            definiteness    = definite
            number          = singular
            predicate       = girl
         object          =
            predicate       = boy
            qform           = -
            definiteness    = definite
            number          = singular
         vcomp           =
            predicate       = kiss(boy, baby)
            subject         =
               predicate       = boy
               qform           = -
               definiteness    = definite
               number          = singular
            object          =
               predicate       = baby
               qform           = -
               definiteness    = definite
               number          = singular
            format          = subject, -, object
         format          = subject, -, object, vcomp
      object          =
```

```
      qform          = -
      definiteness   = definite
      number         = singular
      predicate      = girl
   format         = subject, -, object, vcomp

----------------------------------------------------------------------

[the, boy, want, that, the, apple, falls]

   subject        =
      qform          = -
      definiteness   = definite
      number         = singular
      predicate      = boy
   predicate      = want(boy, fall(apple))
   relative       =
      predicate      = fall(apple)
      subject        =
         qform          = -
         definiteness   = definite
         number         = singular
         predicate      = apple
      format         = subject, -
   format         = subject, -, relative

----------------------------------------------------------------------

[i, want, that, the, girl, kiss, the, baby]

   subject        =
      qform          = -
      predicate      = i
   predicate      = want(i, kiss(girl, baby))
   relative       =
      predicate      = kiss(girl, baby)
      subject        =
         qform          = -
         definiteness   = definite
         number         = singular
         predicate      = girl
      object         =
         predicate      = baby
         qform          = -
         definiteness   = definite
         number         = singular
      format         = subject, -, object
   format         = subject, -, relative

----------------------------------------------------------------------

[i, want, the, girl, to, kiss, the, baby]

   subject        =
```

```
      qform          = -
      predicate      = i
   predicate      = want(i, kiss(girl, baby))
   vcomp          =
      predicate      = kiss(girl, baby)
      subject        =
         qform          = -
         definiteness   = definite
         number         = singular
         predicate      = girl
      object         =
         predicate      = baby
         qform          = -
         definiteness   = definite
         number         = singular
      format         = subject, -, object
   object         =
      qform          = -
      definiteness   = definite
      number         = singular
      predicate      = girl
   format         = subject, -, object, vcomp

----------------------------------------------------------------------

[i, want, the, girl, to, promise, me, to, kiss, the, baby]

   subject        =
      qform          = -
      predicate      = i
   predicate      = want(i, promise(girl, i, kiss(girl, baby)))
   vcomp          =
      predicate      = promise(girl, i, kiss(girl, baby))
      subject        =
         qform          = -
         definiteness   = definite
         number         = singular
         predicate      = girl
      object         =
         predicate      = i
         qform          = -
      vcomp          =
         predicate      = kiss(girl, baby)
         subject        =
            predicate      = girl
            qform          = -
            definiteness   = definite
            number         = singular
         object         =
            predicate      = baby
            qform          = -
            definiteness   = definite
            number         = singular
         format         = subject, -, object
```

```
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      definiteness    = definite
      number          = singular
      predicate       = girl
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, girl, did, want, me, to, promise, the, boy, to, kiss, the, baby]

   subject         =
      predicate       = girl
      qform           = +
      number          = singular
   predicate       = want(girl, promise(i, boy, kiss(i, baby)))
   vcomp           =
      predicate       = promise(i, boy, kiss(i, baby))
      subject         =
         qform           = -
         predicate       = i
      object          =
         predicate       = boy
         qform           = -
         definiteness    = definite
         number          = singular
      vcomp           =
         predicate       = kiss(i, baby)
         subject         =
            predicate       = i
            qform           = -
         object          =
            predicate       = baby
            qform           = -
            definiteness    = definite
            number          = singular
         format          = subject, -, object
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, boy, did, the, girl, want, me, to, promise, to, kiss, the, baby]

   subject         =
      qform           = -
      definiteness    = definite
      number          = singular
      predicate       = girl
   predicate       = want(girl, promise(i, boy, kiss(i, baby)))
```

```
     vcomp          =
        predicate      = promise(i, boy, kiss(i, baby))
        subject        =
           qform          = -
           predicate      = i
        object         =
           predicate      = boy
           qform          = +
           number         = singular
        vcomp          =
           predicate      = kiss(i, baby)
           subject        =
              predicate      = i
              qform          = -
           object         =
              predicate      = baby
              qform          = -
              definiteness   = definite
              number         = singular
           format         = subject, -, object
        format         = subject, -, object, vcomp
     object         =
        qform          = -
        predicate      = i
     format         = subject, -, object, vcomp

----------------------------------------------------------------------

[which, baby, did, the, girl, want, me, to, promise, the, boy, to, kiss]

     subject        =
        qform          = -
        definiteness   = definite
        number         = singular
        predicate      = girl
     predicate      = want(girl, promise(i, boy, kiss(i, baby)))
     vcomp          =
        predicate      = promise(i, boy, kiss(i, baby))
        subject        =
           qform          = -
           predicate      = i
        object         =
           predicate      = boy
           qform          = -
           definiteness   = definite
           number         = singular
        vcomp          =
           predicate      = kiss(i, baby)
           subject        =
              predicate      = i
              qform          = -
           object         =
              predicate      = baby
              qform          = +
```

```
            number          = singular
         format          = subject, -, object
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, girl, did, want, me, to, persuade, the, boy, to, kiss, the, baby]

   subject         =
      predicate       = girl
      qform           = +
      number          = singular
   predicate       = want(girl, persuade(i, boy, kiss(boy, baby)))
   vcomp           =
      predicate       = persuade(i, boy, kiss(boy, baby))
      subject         =
         qform           = -
         predicate       = i
      object          =
         predicate       = boy
         qform           = -
         definiteness    = definite
         number          = singular
      vcomp           =
         predicate       = kiss(boy, baby)
         subject         =
            predicate       = boy
            qform           = -
            definiteness    = definite
            number          = singular
         object          =
            predicate       = baby
            qform           = -
            definiteness    = definite
            number          = singular
         format          = subject, -, object
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, boy, did, the, girl, want, me, to, persuade, to, kiss, the, baby]

   subject         =
      qform           = -
      definiteness    = definite
      number          = singular
```

```
      predicate       = girl
   predicate       = want(girl, persuade(i, boy, kiss(boy, baby)))
   vcomp           =
      predicate       = persuade(i, boy, kiss(boy, baby))
      subject         =
         qform           = -
         predicate       = i
      object          =
         predicate       = boy
         qform           = +
         number          = singular
      vcomp           =
         predicate       = kiss(boy, baby)
         subject         =
            predicate       = boy
            qform           = +
            number          = singular
         object          =
            predicate       = baby
            qform           = -
            definiteness    = definite
            number          = singular
         format          = subject, -, object
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, baby, did, the, girl, want, me, to, persuade, the, boy, to, kiss]

   subject         =
      qform           = -
      definiteness    = definite
      number          = singular
      predicate       = girl
   predicate       = want(girl, persuade(i, boy, kiss(boy, baby)))
   vcomp           =
      predicate       = persuade(i, boy, kiss(boy, baby))
      subject         =
         qform           = -
         predicate       = i
      object          =
         predicate       = boy
         qform           = -
         definiteness    = definite
         number          = singular
      vcomp           =
         predicate       = kiss(boy, baby)
         subject         =
            predicate       = boy
            qform           = -
```

```
            definiteness    = definite
            number          = singular
         object          =
            predicate       = baby
            qform           = +
            number          = singular
         format          = subject, -, object
      format          = subject, -, object, vcomp
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp

----------------------------------------------------------------------

[which, toys, did, the, girl, want, me, to, handle, the, baby]

   subject         =
      qform           = -
      definiteness    = definite
      number          = singular
      predicate       = girl
   predicate       = want(girl, hand(i, baby, toy))
   vcomp           =
      predicate       = hand(i, baby, toy)
      subject         =
         qform           = -
         predicate       = i
      object          =
         predicate       = baby
         qform           = -
         definiteness    = definite
         number          = singular
      object2         =
         predicate       = toy
         qform           = +
         number          = plural
      format          = subject, -, object, object2
   object          =
      qform           = -
      predicate       = i
   format          = subject, -, object, vcomp
```

ß

D.2 Grammaires catégorielles

Pour une implémentation plus complète de la grammaire de Montague (Voir (Evrard, 1998b)) on se référera à (Ricordel, 1996). Nous allons présenter ici une implémentation du calcul de Lambek/Gentzen

D.2.1 Grammaire catégorielle (version de base)

Dans cette version, les catégories seront combinées entre elles par les trois opérateurs classiques (/, \, .). L'algorithme utilisé est un algorithme ascendant. On utilise deux règles ($(x/y).y \rightarrow x$ et $y.(x\backslash y) \rightarrow x$) pour reconstituer la catégorie 's'. On ne se préoccupe ici que du problème de reconstruction de la structure de la phrase. Le moteur de recherche avec retour arrière de Prolog permet d'écrire simplement cet analyseur. Le but de cette partie est de fournir un programme qui cherche la structure d'une phrase et affiche les différentes étapes du calcul.

Définition des catégories

Les catégories utilisées sont celles du support de cours, ce qui donne, en Prolog :

```
categorie(phrase,s).
categorie(verbe_intransitif,iv).
categorie(nom_commun,cn).
categorie(article,(s/iv)/cn).
categorie(nom_propre,(s/iv)).
categorie(pronom_personnel,(s/iv)).
categorie(verbe_transitif,iv/(s/iv)).
categorie(adjectif,cn/cn).
categorie(adjectif,cn\cn).
categorie(verbe_d_etat,iv/(cn/cn)).
categorie(verbe_d_etat,iv/(cn\cn)).
categorie(pronom_relatif_qui,(((s/iv)\(s/iv))/iv)).
categorie(de_complement_de_nom,((cn\cn)/(s/iv))).
categorie(de_complement_de_nom,((cn\cn)/cn)).
categorie(adverbe,(X/X)):-accepte_adverbe(X).
categorie(adverbe,(X\X)):-accepte_adverbe(X).
categorie(preposition_circonstancielle,(s/s)/(s/iv)).
categorie(preposition_circonstancielle,(s\s)/(s/iv)).
categorie(conjonction_de_coordination,((X/X)\X)).

accepte_adverbe(s).
accepte_adverbe(iv).
accepte_adverbe((iv/(s/iv))).
accepte_adverbe((X/X)).
accepte_adverbe((X\X)).
```

Ainsi, le lexique est constitué par le prédicat mot qui associe à un mot une catégorie précédemment définie, par exemple :

```
mot(le,article).
mot(chat,nom_commun).
mot(dort,verbe_intransitif).
```

Algorithme d'analyse

La première phase est la construction de la liste des catégories des mots de la phrase à analyser. A l'aide des prédicats `mot` et `categorie`, on construit une liste dont chaque élément est la définition de la catégorie du mot correspondant. L'analyse se fera sur les deux listes en parallèle : la phrase et la liste des catégories.

C'est un simple parcours de liste en Prolog, réalisé par le prédicat `liste_categories/2`.

La deuxième phase est l'analyse ascendante. Une fois la liste des catégories construite, on peut appliquer les deux règles syntaxiques suivantes pour reconstruire l'axiome :

- $(x/y).y \rightarrow x$

– $y.(x\backslash y) \rightarrow x$

Le principal problème est de savoir dans quel ordre appliquer ces règles. On retrouve deux actions classiques en analyse syntaxique ascendante : le décalage et la réduction. Dans le cas de cet analyseur, on tentera toutes les réductions possibles avant de continuer l'analyse sur la suite de la phrase. L'algorithme est donc le suivant :

1. Si l'une des règles est applicable, on l'applique et on poursuit l'analyse sur la nouvelle structure produite.
2. Si 1. a échoué (pas de règle applicable, ou poursuite de l'analyse impossible), on poursuit l'analyse sur le reste de la phrase, puis, on recommence en réintégrant la partie de la phrase mise temporairement de coté.

Puisque les deux règles s'appliquent sur deux éléments, il faut traiter le cas particulier correspondant au dernier mot de la phrase. Dans ce cas, le prédicat de réduction doit retourner le mot et la catégorie inchangés.

On notera au passage qu'à chaque application de règle, on mémorise le calcul effectuer, pour pouvoir le restituer à la fin de l'analyse.

Le prédicat `reduire/4` prend quatre arguments :

– La phrase à analyser
– La liste des catégories correspondant aux mots de la phrase
– La phrase structurée après analyse
– La catégorie obtenue ('s' à la fin de l'analyse)

Lors des calculs, le point permet de marquer les deux catégories qui vont se simplifier. Les simplifications se font de la manière suivante :

– Réduction par la règle $(x/y).y \rightarrow x$:

```
reduire([A,B|LM],[(X/Y),Y|LC],NPH,NLC):-
     memoriser([A,'.',B],[(X/Y),'.',Y]),
     reduire([[A,B]|LM],[X|LC],NPH,NLC).
```

– Réduction par la règle $y.(x\backslash y) \rightarrow x$:

```
reduire([A,B|LM],[Y,(X\Y)|LC],NPH,NLC):-
     memoriser([A,'.',B],[Y,'.',(X\Y)]),
     reduire([[A,B]|LM],[X|LC],NPH,NLC).
```

– Analyse du reste de la phrase :

```
reduire([A|LM],[X|LC],NPH,NLC):-
     reduire(LM,LC,LM1,LC1),
     LC\=LC1,         % sinon, ça boucle sur des morceaux irréductibles
     reduire([A|LM1],[X|LC1],NPH,NLC).

reduire(X,C,X,C).
```

Il faut aussi prévoir la mémorisation des étapes de calcul. On mémorise les calculs dans une pile dans l'espace de travail, pour ne pas alourdir le passage de paramètres dans la pile de Prolog. Pour cela, le prédicat `liste_calculs` possède un argument qui est la liste des calculs effectués depuis le début de l'analyse. Avant de commencer, le prédicat `raz_calcul` vide cette liste. Ensuite, à chaque étape, le prédicat `memoriser` permet d'empiler un couple (liste de mots, liste de catégories) correspondant à l'application d'une règle. Cependant, ce prédicat doit être réversible lorsqu'il y a plusieurs possibilités. Ainsi, un deuxième passage dans `memoriser` dépile la pile courante, et se termine par `fail` pour éviter de refaire la même analyse.

Programme

```
/****************************************************************************/
/*                                                                          */
/* Christophe DELORD                                                        */
/*                                                                          */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                 */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*                                                                          */
/* Analyse de phrases : utilisation de la grammaire catégorielle            */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*                                                                          */
/* Définition des catégories                                                */
/*                                                                          */
/****************************************************************************/

:-op(400,yfx,'/').
:-op(400,yfx,'\').

categorie(phrase,s).
categorie(verbe_intransitif,iv).
categorie(nom_commun,cn).
categorie(article,(s/iv)/cn).
categorie(nom_propre,(s/iv)).
categorie(pronom_personnel,(s/iv)).
categorie(verbe_transitif,iv/(s/iv)).
categorie(adjectif,cn/cn).
categorie(adjectif,cn\cn).
categorie(verbe_d_etat,iv/(cn/cn)).
categorie(verbe_d_etat,iv/(cn\cn)).
categorie(pronom_relatif_qui,(((s/iv)\(s/iv))/iv)).
categorie(de_complement_de_nom,((cn\cn)/(s/iv))).
categorie(de_complement_de_nom,((cn\cn)/cn)).
categorie(adverbe,(X/X)):-accepte_adverbe(X).
categorie(adverbe,(X\X)):-accepte_adverbe(X).
categorie(preposition_circonstancielle,(s/s)/(s/iv)).
categorie(preposition_circonstancielle,(s\s)/(s/iv)).
categorie(conjonction_de_coordination,((X/X)\X)).

accepte_adverbe(s).
accepte_adverbe(iv).
accepte_adverbe((iv/(s/iv))).
accepte_adverbe((X/X)).
accepte_adverbe((X\X)).

/****************************************************************************/
/*                                                                          */
/*  Associations mots/catégories                                            */
```

```
/*                                                                          */
/****************************************************************************/

mot(le,article).
mot(chat,nom_commun).
mot(dort,verbe_intransitif).
mot(souris,nom_commun).
mot(regarde,verbe_transitif).
mot(mange,verbe_transitif).
mot(la,article).
mot(une,article).
mot(qui,pronom_relatif_qui).
mot(du,article).
mot(fromage,nom_commun).
mot(christophe,nom_propre).
mot(vite,adverbe).
mot(tres,adverbe).
mot(blanche,adjectif).
mot(grise,adjectif).
mot(gris,adjectif).
mot(il,pronom_personnel).
mot(court,verbe_intransitif).
mot(dans,preposition_circonstancielle).
mot(jardin,nom_commun).
mot(petit,adjectif).
mot(petite,adjectif).
mot(et,conjonction_de_coordination).
mot(parle,verbe_intransitif).
mot(bien,adverbe).
mot(quant,preposition_circonstancielle).
mot(soleil,nom_commun).
mot(brille,verbe_intransitif).
mot(fait,verbe_transitif).
mot(fait,verbe_d_etat).
mot(est,verbe_d_etat).
mot(beau,adjectif).
mot(mon,article).
mot(oncle,nom_commun).
mot(de,de_complement_de_nom).
mot(ma,article).
mot(tante,nom_commun).

/****************************************************************************/
/*                                                                          */
/*  Affichage d'une liste de catégories                                     */
/*                                                                          */
/****************************************************************************/

afficher_liste_categories([]).
afficher_liste_categories([C|LC]):-
    ( C='.' -> writef(". ") ; writef("(%t) ",[C]) ),
    afficher_liste_categories(LC).

/****************************************************************************/
```

```
/*                                                                         */
/*  Construction de la liste de catégories                                 */
/*                                                                         */
/***************************************************************************/

liste_categories([],[]).
liste_categories([M|PH],[CM|CPH]):-
    mot(M,T),
    categorie(T,CM),
    liste_categories(PH,CPH).

/***************************************************************************/
/*                                                                         */
/*  Mémorisation des étapes de calcul                                      */
/*                                                                         */
/***************************************************************************/

% liste_calculs([...]). % couples : PHRASE:CATEG
:-dynamic liste_calculs/1.

raz_calculs:-retractall(liste_calculs(_)),asserta(liste_calculs([])).

memoriser(PH,LC):-                          % 1ř passage :
    retract(liste_calculs(OLD)),
    asserta(liste_calculs([PH:LC|OLD])).    % on mémorise l'étape

memoriser(_,_):-                            % 2ř passage : (après un échec)
    retract(liste_calculs([_|OLD])),
    asserta(liste_calculs(OLD)),            % on restaure la liste
    fail.                                   % et on ne recommence pas

afficher_calculs:-
    liste_calculs(L),
    afficher_calculs(L).
afficher_calculs([]).
afficher_calculs([PH:LC|SUITE]):-
    afficher_calculs(SUITE),
    writef("%t : ",[PH]),
    afficher_liste_categories(LC),
    nl.

/***************************************************************************/
/*                                                                         */
/*  Grammaire catégorielle (version de base)                               */
/*                                                                         */
/***************************************************************************/

/***************************************************************************/
/*                                                                         */
/* Réduction d'une liste de catégories : (x/y).y=x et y.(x\y)=x            */
/*                                                                         */
/***************************************************************************/

reduire([A,B|LM],[(X/Y),Y|LC],NPH,NLC):-
```

```
    memoriser([A,'.',B],[(X/Y),'.',Y]),
    reduire([[A,B]|LM],[X|LC],NPH,NLC).

reduire([A,B|LM],[Y,(X\Y)|LC],NPH,NLC):-
    memoriser([A,'.',B],[Y,'.',(X\Y)]),
    reduire([[A,B]|LM],[X|LC],NPH,NLC).

reduire([A|LM],[X|LC],NPH,NLC):-
    reduire(LM,LC,LM1,LC1),
    LC\=LC1,                % sinon, ça boucle sur des morceaux irréductibles
    reduire([A|LM1],[X|LC1],NPH,NLC).

reduire(X,C,X,C).

/****************************************************************************/
/*                                                                          */
/*  Analyse d'une phrase                                                    */
/*                                                                          */
/****************************************************************************/

analyser(PH):-
    raz_calculs,
    liste_categories(PH,L),
    reduire(PH,L,NPH,[s]),
    afficher_calculs,
    writef("%t : (s)\n\n",[NPH]).

test:-
    analyser([la,souris,mange,tres,vite,du,fromage]).
```

Exemple d'analyse

Sur la phrase « la souris mange très vite du fromage dans le jardin », (on remarque l'utilisation d'adverbe, d'adverbe d'adverbe et de complément circonstanciel) l'analyseur effectue les calculs suivants :

```
[la,'.',souris] : (s/iv/cn).(cn)
[tres,'.',vite] : (iv/(s/iv)\(iv/(s/iv))/(iv/(s/iv)\(iv/(s/iv)))).
                                                (iv/(s/iv)\(iv/(s/iv)))
[du,'.',fromage] : (s/iv/cn).(cn)
[le,'.',jardin] : (s/iv/cn).(cn)
[dans,'.',[le,jardin]] : (s\s/(s/iv)).(s/iv)
[mange,'.',[tres,vite]] : (iv/(s/iv)).(iv/(s/iv)\(iv/(s/iv)))
[[mange,[tres,vite]],'.',[du,fromage]] : (iv/(s/iv)).(s/iv)
[[la,souris],'.',[[mange,[tres,vite]],[du,fromage]]] : (s/iv).(iv)
[[[la,souris],[[mange,[tres,vite]],[du,fromage]]],'.',[dans,[le,jardin]]] :
                                                (s).(s\s)
[[[[la,souris],[[mange,[tres,vite]],[du,fromage]]],[dans,[le,jardin]]]] : (s)
```

D.2.2 Grammaire catégorielle généralisée de Lambek-Gentzen

Par rapport à la grammaire présentée dans la première partie, on utilisera ici un jeu de règles plus puissant et on ajoutera l'analyse sémantique à l'analyse syntaxique. Pour cela, on a besoin de pouvoir faire du lambda calcul.

Lambda Calcul

Les lambda-termes seront représentés en Prolog de la manière suivante :

MN	$\rightarrow$	$M@N$
$\lambda X.f(X)$	$\rightarrow$	$X\#f(X)$, où X est une variable Prolog
$\lambda Z.Z(a,b)$	$\rightarrow$	$Z\#Z@a@b$

La construction d'un lambda-terme se fait de manière triviale en Prolog, il suffit de l'écrire en utilisant la syntaxe précédente. Il faut ensuite pouvoir simplifier un lambda-terme (béta-réductions et construction des prédicats). Le prédicat `simplifier/2` permet de simplifier un terme en faisant les béta-réductions possibles.

Définition des catégories

Ici, les catégories syntagmatiques seront les suivantes :

P phrase

SN syntagme nominal

SP syntagme prépositionnel

SA syntagme adjectival

Algorithme d'analyse

L'algorithme est le même que précédemment. Les seules modifications correspondent à la construction du prédicat représentant le sens de la phrase.

Dans le programme, A, B, F, G représentent des mots de la phrase, VSA, VSB, VSF, VSG leurs valeurs sémantiques, X, Y, Z des catégories syntagmatiques, SPH la suite de la phrase, NPH la nouvelle phrase (structurée), ST la suite des types (des catégories), NT la nouvelle liste de types, NVS la nouvelle valeur sémantique.

Les règles de simplification sont celles du tableau D.2.

Dans le cas des règles 4, 5, 6 et P, on ajoute comme condition la présence d'une catégorie permettant de se simplifier avec la catégorie obtenue en appliquant la règle. Cela permet d'éviter de s'engager dans des branches infinies lors de l'analyse.

Programme

```
/***************************************************************************/
/*                                                                         */
/* Christophe DELORD                                                       */
/*                                                                         */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel               */
/*                                                                         */
/***************************************************************************/

/***************************************************************************/
/*                                                                         */
/* Analyse de phrases : grammaire catégorielle généralisée de Lambek-Gentzen */
/*                                                                         */
/***************************************************************************/

:-use_module(log_int).

/***************************************************************************/
/*                                                                         */
/* Définition des catégories syntagmatiques                                */
/*                                                                         */
```

Règle 1 :	Applicativité $f : x/y, a : y \to f(a) : x$ $a : y, f : x\backslash y \to f(a) : x$
Règle 2 :	Composition $f : x/y, g : y/z \to \lambda v[f(g(v))] : x/z$ $g : y\backslash z, f : x\backslash y \to \lambda v[f(g(v))] : x\backslash z$
Règle 3 :	Associativité $f : (x\backslash z)/y \to \lambda v[\lambda w[f(w)(v)]] : (x/y)\backslash z$ $f : (x/y)\backslash z \to \lambda v[\lambda w[f(w)(v)]] : (x\backslash z)/y$
Règle 4 :	Montée $a : x, b : (y\backslash x) \to \lambda v[v(a)] : y/(y\backslash x), b : (y\backslash x)$ $b : (y/x), a : x \to b : (y/x), \lambda v[v(a)] : y\backslash(y/x)$
Règle 5 :	Division (foncteur principal) $f : x/y, g : y/z \to \lambda v[\lambda w[f(v(w))]] : (x/z)/(y/z), g : y/z$ $g : y\backslash z, f : x\backslash y \to g : y\backslash z, \lambda v[\lambda w[f(v(w))]] : (x\backslash z)\backslash(y\backslash z)$
Règle 6 :	Division (foncteur subordonné) $g : z/x, f : x/y \to g : z/x, \lambda v[\lambda w[v(f(w))]] : (z/y)\backslash(z/x)$ $f : x\backslash y, g : z\backslash x \to \lambda v[\lambda w[v(f(w))]] : (z\backslash y)/(z\backslash x), g : z\backslash x$
Règle P :	Permutation $t : z/y, a : x, b : y \to t : z/y, b : y, a : x$ $a : x, b : y, t : z\backslash x \to b : y, a : x, t : z\backslash x$
Règle C :	Contraction $a : x, b : x \to (a, b) : x$

TAB. D.2: Grammaire catégorielle de Lambek/Gentzen

```
/***************************************************************************/

:-op(400,yfx,'/').
:-op(400,yfx,'\').

categorie(jean,sn).
categorie(marie,sn).
categorie(pierre,sn).
categorie(anne,sn).
categorie(est,(p\sn)/sa).
categorie(apparente,sa/sp).
categorie(a,sp/sn).
categorie(il,sn).
categorie(juge,(((p\sn)/sa)/sn)).
categorie(incompetente,sa).

/***************************************************************************/
/*                                                                         */
/*  Construction de la liste de catégories                                 */
/*                                                                         */
/***************************************************************************/

liste_categories([],[],[]).
liste_categories([M|PH],[CM|CPH],[M|SVS]):-
    categorie(M,CM),
    liste_categories(PH,CPH,SVS).
```

```
/**************************************************************************/
/*                                                                        */
/*  R1  f: x/y, a: y -> f(a): x                          applicativité */
/*      a: y, f: x\y -> f(a): x                                          */
/*                                                                        */
/**************************************************************************/

reduire([F,A|SPH],[(X/Y),Y|ST],[VSF,VSA|SVS],NPH,NT,NVS):-
    reduire([[F,A]|SPH],[X|ST],[VSF@VSA|SVS],NPH,NT,NVS).
reduire([A,F|SPH],[Y,(X\Y)|ST],[VSA,VSF|SVS],NPH,NT,NVS):-
    reduire([[A,F]|SPH],[X|ST],[VSF@VSA|SVS],NPH,NT,NVS).

/**************************************************************************/
/*                                                                        */
/*  R2  f: x/y, g: y/z -> lv[f(g(v))]: x/z                 composition */
/*      g: y\z, f: x\y -> lv[f(g(v))]: x\z                               */
/*                                                                        */
/**************************************************************************/

reduire([F,G|SPH],[(X/Y),(Y/Z)|ST],[VSF,VSG|SVS],NPH,NT,NVS):-
    reduire([[F,G]|SPH],[(X/Z)|ST],[V#VSF@(VSG@V)|SVS],NPH,NT,NVS).
reduire([G,F|SPH],[(Y\Z),(X\Y)|ST],[VSG,VSF|SVS],NPH,NT,NVS):-
    reduire([[G,F]|SPH],[(X\Z)|ST],[V#VSF@(VSG@V)|SVS],NPH,NT,NVS).

/**************************************************************************/
/*                                                                        */
/*  R3  f: (x\z)/y -> lv[lw[f(w)(v)]]: (x/y)\z           associativité */
/*      f: (x/y)\z -> lv[lw[f(w)(v)]]: (x\z)/y                           */
/*                                                                        */
/**************************************************************************/

reduire([F|SPH],[((X\Z)/Y)|ST],[VSF|SVS],NPH,NT,NVS):-
    reduire([F|SPH],[((X/Y)\Z)|ST],[V#W#VSF@W@V|SVS],NPH,NT,NVS).
reduire([F|SPH],[((X/Y)\Z)|ST],[VSF|SVS],NPH,NT,NVS):-
    reduire([F|SPH],[((X\Z)/Y)|ST],[V#W#VSF@W@V|SVS],NPH,NT,NVS).

/**************************************************************************/
/*                                                                        */
/*  R4  a: x, b: (y\x) -> lv[v(a)]: y/(y\x), b: (y\x)            montée */
/*      b: (y/x), a: x -> b: (y/x), lv[v(a)]: y\(y/x)                    */
/*                                                                        */
/**************************************************************************/

reduire([A,B|SPH],[X,(Y\X)|ST],[VSA,VSB|SVS],NPH,NT,NVS):-
    reduire([A,B|SPH],[(Y/(Y\X)),(Y\X)|ST],[V#V@VSA,VSB|SVS],NPH,NT,NVS).
reduire([B,A|SPH],[(Y/X),X|ST],[VSB,VSA|SVS],NPH,NT,NVS):-
    reduire([B,A|SPH],[(Y/X),(Y\(Y/X))|ST],[VSB,V#V@VSA|SVS],NPH,NT,NVS).

/**************************************************************************/
/*                                                                        */
/*  R5  f: x/y, g: y/z -> lv[lw[f(v(w))]]: (x/z)/(y/z), g: y/z   division */
/*      g: y\z, f: x\y -> g: y\z, lv[lw[f(v(w))]]: (x\z)\(y\z)  foncteur */
/*                                                                principal */
```

```
/****************************************************************************/

reduire([F,G|SPH],[(X/Y),(Y/Z)|ST],[VSF,VSG|SVS],NPH,NT,NVS):-
    reduire([F,G|SPH],[((X/Z)/(Y/Z)),(Y/Z)|ST],[V#W#VSF@(V@W),VSG|SVS],
                                                            NPH,NT,NVS).
reduire([G,F|SPH],[(Y\Z),(X\Y)|ST],[VSG,VSF|SVS],NPH,NT,NVS):-
    reduire([G,F|SPH],[(Y\Z),((X\Z)\(Y\Z))|ST],[VSG,V#W#VSF@(V@W)|SVS],
                                                            NPH,NT,NVS).

/****************************************************************************/
/*                                                                          */
/*  R6  g: z/x, f: x/y -> g: z/x, lv[lw[v(f(w))]]: (z/y)\(z/x)      division */
/*      f: x\y, g: z\x -> lv[lw[v(f(w))]]: (z\y)/(z\x), g: z\x      foncteur */
/*                                                                subordonné */
/****************************************************************************/

reduire([G,F|SPH],[(Z/X),(X/Y)|ST],[VSG,VSF|SVS],NPH,NT,NVS):-
    reduire([G,F|SPH],[(Z/X),((Z/Y)\(Z/X))|ST],[VSG,V#W#V@(VSF@W)|SVS],
                                                            NPH,NT,NVS).
reduire([F,G|SPH],[(X\Y),(Z\X)|ST],[VSF,VSG|SVS],NPH,NT,NVS):-
    reduire([F,G|SPH],[((Z\Y)/(Z\X)),(Z\X)|ST],[V#W#V@(VSF@W),VSG|SVS],
                                                            NPH,NT,NVS).

/****************************************************************************/
/*                                                                          */
/*  P   t: z/y, a: x, b: y -> t: z/y, b: y, a: x               permutation */
/*      a: x, b: y, t: z\x -> b: y, a: x, t: z\x                          */
/*                                                                          */
/****************************************************************************/

reduire([T,A,B|SPH],[(Z/Y),X,Y|ST],[VST,VSA,VSB|SVS],NPH,NT,NVS):-
    reduire([T,B,A|SPH],[(Z/Y),Y,X|ST],[VST,VSB,VSA|SVS],NPH,NT,NVS).
reduire([A,B,T|SPH],[X,Y,(Z\X)|ST],[VSA,VSB,VST|SVS],NPH,NT,NVS):-
    reduire([B,A,T|SPH],[Y,X,(Z\X)|ST],[VSB,VSA,VST|SVS],NPH,NT,NVS).

/****************************************************************************/
/*                                                                          */
/*  C   a: x, b: x -> (a,b): x                                 contraction */
/*                                                                          */
/****************************************************************************/

reduire([A,B|SPH],[X,X|ST],[VSA,VSB|SVS],NPH,NT,NVS):-
    reduire([[A,B]|SPH],[X|ST],[contraction(VSA,VSB)|SVS],NPH,NT,NVS).

/****************************************************************************/
/*                                                                          */
/*  Cas où l'on ne peut pas simplifier => on avance                        */
/*                                                                          */
/****************************************************************************/
reduire([A|SPH],[X|ST],[VSA|SVS],NPH,NT,NVS):-
    reduire(SPH,ST,SVS,NPH1,NT1,NVS1),
    ST\=NT1,                            % sinon, ça boucle
    reduire([A|NPH1],[X|NT1],[VSA|NVS1],NPH,NT,NVS).
reduire(A,X,V,A,X,V).
```

```
/******************************************************************************/
/*                                                                            */
/*  Analyse d'une phrase                                                      */
/*                                                                            */
/******************************************************************************/
analyser(PH,VS_S):-
    liste_categories(PH,LC,LVS),
    reduire(PH,LC,LVS,NPH,[p],[VS]),
    writef("%t <=> \n\t%t\n",[NPH,VS]),
    simplifier(VS,VS_S),
    writef("\t%t\n",[VS_S]).

test:-
    analyser([il,juge,incompetente,marie],_).
```

D.2.3 Exemples d'analyse

```
?-analyser([il,juge,marie,incompetente],_).
[[il,[[juge,marie],incompetente]]] <=>
juge @ marie @ incompetente @ il
juge @ marie @ incompetente @ il

?-analyser([jean,est,apparente,a,marie],_).
 [[jean,[[[est,apparente],a],marie]]] <=>
(G2152 # (G2044 # est @ (apparente @ G2044)) @ (a @ G2152)) @ marie @ jean
est @ (apparente @ (a @ marie)) @ jean
```

Annexe E

Analyseur ascendant dépendant du contexte

E.1 Caractéristiques de l'analyseur

Pour être utilisable dans le cadre d'un langage naturel, cet analyseur doit être suffisamment général, c'est-à-dire capable de traiter des langages dépendants du contexte. Cela implique que la tête des règles de grammaire peut avoir plusieurs symbôles. Pour effectuer l'analyse sémantique en parallèle, chaque symbôle peut avoir plusieurs arguments qui, au fur et à mesure de l'analyse, seront unifiés avec les arguments de même nom apparaissant dans la partie droite de la règle. Les actions sémantiques apparaîtront en partie droite de règle, entre accolades. Pour éviter le problème des règles récursives à gauche et pouvoir s'adapter facilement aux grammaires catégorielles, l'analyseur sera ascendant. Pour l'analyse lexicale, on programmera un prédicat qui sera appelé par l'analyseur syntaxique.

E.2 Algorithme

L'analyseur doit rester très général. L'algorithme de recherche des règles à appliquer est donc très général lui aussi.

A chaque étape, on mémorise les symbôles déjà reconnus dans une liste. A partir de cette liste, on a le choix entre appliquer une règle de grammaire, c'est-à-dire remplacer une partie de la liste constituant une partie droite de règle par la partie gauche de cette règle ou, si aucune règle n'est applicable, lire l'unité lexicale suivante. Si aucune règle ne s'applique, on lit une unité lexicale en utilisant le prédicat `lex` d'arité 3. Ce prédicat renvoie dans le premier argument la prochaine unité lexicale, le deuxième argument est le flot d'entrée, le troisième, le flot d'entrée après avoir retiré l'unité lexicale. Il est bien évident que pour des langages simples, cet algorithme d'analyse n'est pas efficace du tout.

Ceci est l'algorithme général. On remarque que l'on peut éviter beaucoup de travail lors de la recherche de règle applicable, en effet, lorsque l'on effectue un décalage, il est inutile de scruter toute la liste de symbôles pour chercher une règle applicable, il suffit de regarder la fin de la liste (le décalage ne fait que rajouter un symbôle à la fin de la liste, donc ne pas prendre ce symbôle en compte conduirait aux mêmes échecs que lors des tentatives précédantes). Les symbôles internes sont considérés après une réduction puisque seule la réduction peut modifier la structure de la liste des symbôles. Ainsi, l'algorithme final devient :

1. Commencer l'analyse avec une liste de symbôles vide.
2. Décalage : Lire une unité lexicale et l'ajouter à la liste des symbôles
3. Chercher une règle applicable à la fin de la liste des symbôles

4. S'il n'y en a pas, recommencer à décaler en 2.
5. Appliquer cette règle
6. Réduction : Tant qu'il y a des règles applicables (pas seulement à la fin de la liste)
7. Appliquer ces règles
8. Effectuer un nouveau décalage en 2.

L'analyse est terminée lorsque la liste des symbôles ne contient que l'axiôme de la grammaire et qu'il n'y a plus rien à analyser.

E.3 Implantation en Prolog

E.3.1 Syntaxe de la grammaire

La syntaxe utilisée permet d'écrire les règles de grammaires de manières intuitives, elle conserve le formalisme des règles de grammaires tout en intégrant les facilités offertes par Prolog. Pour illustrer ces propos, on prendra l'exemple de l'analyse d'une expression arithmétique (les applications exposées plus loin sont cependant plus représentatives des possibilités de l'analyseur).

Analyse lexicale

L'utilisateur doit fournir un prédicat `lex(TOKEN, IN, OUT)`. `TOKEN` est unifié avec l'unité lexicale reconnue en tête de `IN`. `OUT` correspond à l'entrée `IN` à laquelle l'unité lexicale a été retirée.

```
lex(separateur)-->" ".
lex(operateur(+,2))-->"+". % priorité 2
lex(operateur(-,2))-->"-".
lex(operateur(*,1))-->"*".
lex(operateur(/,1))-->"/".
lex(openpar)-->"(".
lex(closepar)-->")".
lex(end)-->";"
lex(reel(V))-->entier(IP),fp(FP),{V is IP+FP}.
lex(entier(E))-->entier(E).

lex(bidon)-->[_].

entier(I)-->chiffres(C), {name(I,C)}.

fp(FP)-->".",{!},chiffres(C),{name(X,C), length(C,L), FP is X/10^L}.
fp(0)-->[].

chiffres([C|S])-->chiffre(C),(chiffres(S);{S=[]}),{!}.
chiffre(C)-->[C],{between(0'0,0'9,C)}.
```

La syntaxe utilisée est compatible avec celle des règles de grammaires reconnue par Prolog, cela simplifie l'écriture de l'analyseur lexical.

Analyses syntaxique et sémantique

Les règles de grammaires sont de la forme `TETE := QUEUE`. Les actions sémantiques sont intégrées dans `QUEUE`, entre accolades. `[]` permet de spécifier une tête vide, par exemple pour ignorer des commentaires.

```
[]:=separateur.     % On ignore tous les séparateurs
```

```
expr(V,P):=expr(V1,P1),operateur(OP,P),expr(V2,P2),
  {calc(V1,P1,OP,P,V2,P2,V)}.
expr(V,P):=operateur_unaire(OP), expr(V1,P),{calc_un(OP,V1,V)}.
expr(V,0):=openpar,expr(V,_),closepar.
expr(V,0):=reel(V).
expr(V,0):=fonction(F),openpar,expr(X,_),closepar,{V=..[F,X]}.
expr(V,0):=expr(V,_),end.

end:=end,bidon. % Ignore tout après le point virgule
```

E.3.2 Implantation de l'analyseur syntaxique et sémantique

Il s'agit de la transcription de l'algorithme ci-dessus en Prolog. Tous les points de choix (règle à utiliser, ...) présentés dans l'algorithme sont testés séquentiellement par l'interprète Prolog. L'axiôme à reconnaître est transmis à tous les prédicats afin de stopper l'analyse dès que celui-ci est reconnu.

Décalage

`lex` construit dans `TOKEN` l'unité lexicale courante. Ensuite on essaie d'utiliser cette unité pour l'application d'une règle, c'est-à-dire en fin de la liste des symbôles. Si cela échoue, on continue à décaler.

```
parse(AXIOM,SOURCE,SYMBOLS):-
    shift(AXIOM,SOURCE,SYMBOLS).

shift(AXIOM,SOURCE,SYMBOLS):-
    lex(TOKEN,SOURCE,REST),
    (
        /* On n'applique les règles qu'à la queue */
        (HEAD := BODY),
        apply_rule((HEAD := BODY),[TOKEN|SYMBOLS],NEW_SYMBOLS),
        parse(AXIOM,REST,NEW_SYMBOLS)
    ;
        /* Si pas de règle applicable en queue : on continue à décaler */
        shift(AXIOM,REST,[TOKEN|SYMBOLS])
    ).
```

Réduction

Pour réduire, on applique une règle (le choix est fait par Prolog). L'application de la règle doit pouvoir se faire sur toute la liste de symbôles et non seulement sur la fin, comme après un décalage. Puis on continue l'analyse avec les nouveaux symbôles (l'analyseur tentera d'abord des réductions avant de décaler).

```
parse(AXIOM,SOURCE,SYMBOLS):-
    (HEAD := BODY),     % On prend une règle
    apply_rule_globally((HEAD := BODY),SYMBOLS,NEW_SYMBOLS),
    parse(AXIOM,SOURCE,NEW_SYMBOLS).
```

Application d'une règle

Pour appliquer une règle, il suffit de remplacer la partie de la liste qui constitue la partie droite de la règle par la partie gauche de la règle. Il faut ensuite exécuter les actions sémantiques de la partie droite. Pour appliquer une règle sur la globalité de la liste des symbôles, on utilise le prédicat `apply_rule_globally`.

```
apply_rule_globally(RULE, SYMBOLS, NEW_SYMBOLS):-
    apply_rule(RULE, SYMBOLS, NEW_SYMBOLS).
apply_rule_globally(RULE, [SYMBOL|SYMBOLS], [SYMBOL|NEW_SYMBOLS]):-
    apply_rule_globally(RULE, SYMBOLS, NEW_SYMBOLS).

apply_rule((HEAD := BODY), SYMBOLS, NEW_SYMBOLS):-
    remove_symbols(BODY, SYMBOLS, REST),
    append_symbols(HEAD, REST, NEW_SYMBOLS),
    apply_semantics(HEAD := BODY).
```

Pour supprimer les symbôles de la fin de la liste, on parcours cette liste en supprimant d'abord le dernier symbôle de la partie droite et en ignorant les actions sémantiques.

```
remove_symbols((SYMBOL,SYMBOLS), LIST, REST):-
    !,
    remove_symbols(SYMBOLS, LIST, [SYMBOL|REST]).

remove_symbols({_},LIST,LIST):-!.   % actions sémantiques

remove_symbols(SYMBOL, [SYMBOL|SYMBOLS], SYMBOLS).  % pour le dernier
```

Sur le même principe, on reconstruit la liste en rajoutant les symbôles de la partie gauche à la liste des symbôles restants.

```
append_symbols([],LIST,LIST):-
    !.

append_symbols((SYMBOL,SYMBOLS), LIST, REST):-
    !,
    append_symbols(SYMBOLS,[SYMBOL|LIST],REST).
append_symbols(SYMBOL, LIST, [SYMBOL|LIST]).
```

Il ne reste plus qu'à appliquer les actions sémantiques. Pour cela on parcours le corps de la règle et on exécute toutes les actions rencontrées de gauche à droite.

```
apply_semantics(HEAD := (ITEM, BODY)):-
    !,
    apply_semantics(HEAD := ITEM),
    apply_semantics(HEAD := BODY).

apply_semantics(HEAD := {SEMANTIC,SEMANTICS}):-
    !,
    SEMANTIC,
    apply_semantics(HEAD := {SEMANTICS}).
apply_semantics(_ := {SEMANTIC}):-
    !,
    SEMANTIC.

apply_semantics(_ := _).
```

Analyse

L'analyse commence par un décalage :

```
parse(AXIOM,SOURCE):-
    shift(AXIOM,SOURCE,[]).
```

Et se termine lorsque l'axiôme est reconnu et qu'il n'y a plus rien à lire :

```
parse(AXIOM,[],[AXIOM]).
```

Il y a ensuite les prédicats permettant la réduction et le décalage (dans cet ordre !).

E.4 Sources de l'analyseur

```
/*
 * Analyseur ascendant en ProLog
 *          ---------
 */

:-module(parser,
    [
        parse/2
    ]
).

pause:-get_single_char(0'x),break.
pause.

/*
 *  Définition des règles :
 *      tete := corps
 *
 *  Actions sémantiques :
 *      { ... }
 *
 *  Token lexicaux :
 *      appel à une règle descendante récursive (-->)
 *
 */

:-op(1200,xfx,:=).
:-module_transparent parse/2, parse/3, shift/3.

parse(AXIOM,SOURCE):-
    shift(AXIOM,SOURCE,[]).

aff([A|SUITE]):-
    aff(SUITE),
%   A=..[F|_],
    writef(" %t |",[A]).
aff([]).

parse(_,_,SYMBOLS):-
    writef("Pile courante : "),
    aff(SYMBOLS), nl, pause,
    fail.

% parse/3 : parse(AXIOM,FLOT_D_ENTREE,LISTE_DES_SYMBOLES_RECONNUS)
% La liste des symboles est renversée : dernier symbole lu en tete
```

```
parse(AXIOM,[],[AXIOM]).% L'axiome est reconstruit et plus rien à lire

% Reduce
parse(AXIOM,SOURCE,SYMBOLS):-
    (HEAD := BODY),     % On prend une règle
    apply_rule_globally((HEAD := BODY),SYMBOLS,NEW_SYMBOLS),
    parse(AXIOM,SOURCE,NEW_SYMBOLS).

% Shift
parse(AXIOM,SOURCE,SYMBOLS):-
    shift(AXIOM,SOURCE,SYMBOLS).

shift(AXIOM,SOURCE,SYMBOLS):-
    lex(TOKEN,SOURCE,REST),
    TOKEN=..[F|_],
    writef("Token : %t\n",[F]),
    (
        /* On n'applique les règles qu'à la queue */
        (HEAD := BODY),
        apply_rule((HEAD := BODY),[TOKEN|SYMBOLS],NEW_SYMBOLS),
        parse(AXIOM,REST,NEW_SYMBOLS)
    ;
        /* Si pas de règle applicable en queue : on continue à décaler */
        shift(AXIOM,REST,[TOKEN|SYMBOLS])
    ).

/* OPTIMISATION : shift
        Après un shift, il suffit de n'appliquer les règles qu'à la
        fin de la liste puisqu'elles ont échouées pendant reduce
        Après un reduce, ce n'est pas le cas
*/

apply_rule_globally(RULE, SYMBOLS, NEW_SYMBOLS):-
    apply_rule(RULE, SYMBOLS, NEW_SYMBOLS).
apply_rule_globally(RULE, [SYMBOL|SYMBOLS], [SYMBOL|NEW_SYMBOLS]):-
    apply_rule_globally(RULE, SYMBOLS, NEW_SYMBOLS).

apply_rule((HEAD := BODY), SYMBOLS, NEW_SYMBOLS):-
    remove_symbols(BODY, SYMBOLS, REST),
    append_symbols(HEAD, REST, NEW_SYMBOLS),
    apply_semantics(HEAD := BODY).
%   writef("Regle appliquee : %t\n",[HEAD:=BODY]),pause.

remove_symbols((SYMBOL,SYMBOLS), LIST, REST):-
    !,
    remove_symbols(SYMBOLS, LIST, REST1),
    remove_symbols(SYMBOL, REST1, REST).

remove_symbols({_},LIST,LIST):-!.   % actions sémantiques

remove_symbols(SYMBOL, [SYMBOL|SYMBOLS], SYMBOLS).
    % pour enlever le dernier

append_symbols([],LIST,LIST):-
```

```
    !.

append_symbols((SYMBOL,SYMBOLS), LIST, REST):-
    !,
    append_symbols(SYMBOLS,[SYMBOL|LIST],REST).
append_symbols(SYMBOL, LIST, [SYMBOL|LIST]).

apply_semantics(HEAD := (ITEM, BODY)):-
    !,
    apply_semantics(HEAD := ITEM),
    apply_semantics(HEAD := BODY).

apply_semantics(HEAD := {SEMANTIC,SEMANTICS}):-
    !,
    SEMANTIC,
    apply_semantics(HEAD := {SEMANTICS}).
apply_semantics(_ := {SEMANTIC}):-
    !,
    SEMANTIC.

apply_semantics(_ := _).
```

E.5 Application : Grammaires catégorielles

On va reprendre ici les grammaires catégorielle de base et généralisée de Lambek-Gentzen.

E.5.1 Grammaire catégorielle de base

Il s'agit de la grammaire développée initialement, réécrite pour l'analyseur présenté dans cette annexe.

Le lexique est le même. Le prédicat `lex` permettant l'analyse lexicale est donc :

```
lex(lex(MOT,CAT,[]))-->[MOT],{mot(MOT,T),categorie(T,CAT)}.
```

Les étapes de calcul sont mémorisées différemment : on utilise une liste en paramètre du symbôle `lex`. Ainsi, pour afficher ces étapes, il suffit d'appeler le prédicat suivant :

```
afficher_calculs([A,B,CA,CB,CAB|S]):-
    !,
    writef("%t . %t : %t . %t => %t\n",[A,B,CA,CB,CAB]),
    afficher_calculs(S).
afficher_calculs([]).
```

Les deux règles de grammaires s'écrivent alors naturellement :

```
lex([A,B],X,CAB):=lex(A,X/Y,CA),lex(B,Y,CB),
    {append(CA,CB,TMP), append(TMP,[A,B,X/Y,Y,X],CAB)}.
lex([A,B],X,CAB):=lex(A,Y,CA),lex(B,X\Y,CB),
    {append(CA,CB,TMP), append(TMP,[A,B,Y,X\Y,X],CAB)}.
```

`CA`, `CB`, `CAB` représentent les listes utilisées pour mémoriser les calculs.

Pour analyser une phrase, on veut obtenir la catégorie s dans le deuxième argument de `lex` :

```
analyser(PH):-
    parse(lex(_,s,CALC),PH),
    afficher_calculs(CALC).
```

Il se trouve que cet analyseur est plus rapide que celui de la première version de cette grammaire. Cela est du à l'optimisation d'application des règles juste après un décalage. Ce qui n'existait pas dans la première version. On notera aussi que l'écriture de la grammaire avec cet analyseur est plus intuitive. Le cas de la grammaire de Lambek-Gentzen le montre bien.

E.5.2 Sources de la grammaire de base

```
/*****************************************************************************/
/*                                                                           */
/* Christophe DELORD                                                         */
/*                                                                           */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                  */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* Analyse de phrases : utilisation de la grammaire catégorielle             */
/*                                                                           */
/*****************************************************************************/

:-use_module(parser).

/*****************************************************************************/
/*                                                                           */
/* Définition des catégories                                                 */
/*                                                                           */
/*****************************************************************************/

:-op(400,yfx,'/').
:-op(400,yfx,'\').

categorie(phrase,s).
categorie(verbe_intransitif,iv).
categorie(nom_commun,cn).
categorie(article,(s/iv)/cn).
categorie(nom_propre,(s/iv)).
categorie(pronom_personnel,(s/iv)).
categorie(verbe_transitif,iv/(s/iv)).
categorie(adjectif,cn/cn).
categorie(adjectif,cn\cn).
categorie(verbe_d_etat,iv/(cn/cn)).
categorie(verbe_d_etat,iv/(cn\cn)).
categorie(pronom_relatif_qui,(((s/iv)\(s/iv))/iv)).
categorie(de_complement_de_nom,((cn\cn)/(s/iv))).
categorie(de_complement_de_nom,((cn\cn)/cn)).
categorie(adverbe,(X/X)):-accepte_adverbe(X).
categorie(adverbe,(X\X)):-accepte_adverbe(X).
categorie(preposition_circonstancielle,(s/s)/(s/iv)).
categorie(preposition_circonstancielle,(s\s)/(s/iv)).
categorie(conjonction_de_coordination,((X/X)\X)).

accepte_adverbe(s).
accepte_adverbe(iv).
```

```
accepte_adverbe((iv/(s/iv))).
accepte_adverbe((X/X)).
accepte_adverbe((X\X)).

/*************************************************************************/
/*                                                                       */
/*  Associations mots/catégories                                         */
/*                                                                       */
/*************************************************************************/

mot(le,article).
mot(chat,nom_commun).
mot(dort,verbe_intransitif).
mot(souris,nom_commun).
mot(regarde,verbe_transitif).
mot(mange,verbe_transitif).
mot(la,article).
mot(une,article).
mot(qui,pronom_relatif_qui).
mot(du,article).
mot(fromage,nom_commun).
mot(christophe,nom_propre).
mot(vite,adverbe).
mot(tres,adverbe).
mot(blanche,adjectif).
mot(grise,adjectif).
mot(gris,adjectif).
mot(il,pronom_personnel).
mot(court,verbe_intransitif).
mot(dans,preposition_circonstancielle).
mot(jardin,nom_commun).
mot(petit,adjectif).
mot(petite,adjectif).
mot(et,conjonction_de_coordination).
mot(parle,verbe_intransitif).
mot(bien,adverbe).
mot(quant,preposition_circonstancielle).
mot(soleil,nom_commun).
mot(brille,verbe_intransitif).
mot(fait,verbe_transitif).
mot(fait,verbe_d_etat).
mot(est,verbe_d_etat).
mot(beau,adjectif).
mot(mon,article).
mot(oncle,nom_commun).
mot(de,de_complement_de_nom).
mot(ma,article).
mot(tante,nom_commun).

lex(lex(MOT,CAT,[]))-->[MOT],{mot(MOT,T),categorie(T,CAT)}.

/*************************************************************************/
/*                                                                       */
/*  Etapes de calcul                                                     */
```

```
/*                                                                          */
/****************************************************************************/

afficher_calculs([A,B,CA,CB,CAB|S]):-
    !,
    writef("%t . %t : %t . %t => %t\n",[A,B,CA,CB,CAB]),
    afficher_calculs(S).
afficher_calculs([]).

/****************************************************************************/
/*                                                                          */
/*  Grammaire catégorielle (version de base)                                */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*                                                                          */
/* Réduction d'une liste de catégories : (x/y).y=x et y.(x\y)=x             */
/*                                                                          */
/****************************************************************************/

lex([A,B],X,CAB):=lex(A,X/Y,CA),lex(B,Y,CB),
    {append(CA,CB,TMP), append(TMP,[A,B,X/Y,Y,X],CAB)}.
lex([A,B],X,CAB):=lex(A,Y,CA),lex(B,X\Y,CB),
    {append(CA,CB,TMP), append(TMP,[A,B,Y,X\Y,X],CAB)}.

/****************************************************************************/
/*                                                                          */
/*  Analyse d'une phrase                                                    */
/*                                                                          */
/****************************************************************************/

analyser(PH):-
    parse(lex(_,s,CALC),PH),
    afficher_calculs(CALC).

test:-
    analyser([la,souris,mange,tres,vite,du,fromage]).
```

E.5.3 Grammaire catégorielle généralisé de Lambek-Gentzen

De la même manière, en reprenant le même lexique, l'analyseur lexical se réduit au prédicat suivant :

```
lex(lex(MOT,MOT,CAT))-->[MOT],{categorie(MOT,CAT)}.
```

Le symbôle `lex` possède trois arguments : le morceau de phrase structuré reconnu, l'expression en lambda-calcul et la catégorie correspondantes.

Les règles s'écrivent alors :

R1 :

```
lex([F,A],VSF@VSA,X):=lex(F,VSF,X/Y),lex(A,VSA,Y).
lex([A,F],VSF@VSA,X):=lex(A,VSA,Y),lex(F,VSF,X\Y).
```

R2 :

```
lex([F,G],V#VSF@(VSG@V),X/Z):=lex(F,VSF,X/Y),lex(G,VSG,Y/Z).
lex([G,F],V#VSF@(VSG@V),X\Z):=lex(G,VSG,Y\Z),lex(F,VSF,X\Y).
```

R3 :

```
lex(F,V#W#VSF@W@V,(X/Y)\Z):=lex(F,VSF,(X\Z)/Y).
lex(F,V#W#VSF@W@V,(X\Z)/Y):=lex(F,VSF,(X/Y)\Z).
```

R4 :

```
lex(A,V#V@VSA,Y/(Y\X)),lex(B,VSB,Y\X):=lex(A,VSA,X),lex(B,VSB,Y\X).
lex(B,VSB,Y/X),lex(A,V#V@VSA,Y\(Y/X)):=lex(B,VSB,Y/X),lex(A,VSA,X).
```

R5 :

```
lex(F,V#W#VSF@(V@W),(X/Z)/(Y/Z)),lex(G,VSG,Y/Z):=
  lex(F,VSF,X/Y),lex(G,VSG,Y/Z).
lex(G,VSG,Y\Z),lex(F,V#W#VSF@(V@W),(X\Z)\(Y\Z)):=
  lex(G,VSG,Y\Z),lex(F,VSF,X\Y).
```

R6 :

```
lex(G,VSG,Z/X),lex(F,V#W#V@(VSF@W),(Z/Y)\(Z/X)):=
  lex(G,VSG,Z/X),lex(F,VSF,X/Y).
lex(F,V#W#V@(VSF@W),(Z\Y)/(Z\X)),lex(G,VSG,Z\X):=
  lex(F,VSF,X\Y),lex(G,VSG,Z\X).
```

P :

```
lex(T,VST,Z/Y),lex(B,VSB,Y),lex(A,VSA,X):=
  lex(T,VST,Z/Y),lex(A,VSA,X),lex(B,VSB,Y).
lex(B,VSB,Y),lex(A,VSA,X),lex(T,VST,Z\X):=
  lex(A,VSA,X),lex(B,VSB,Y),lex(T,VST,Z\X).
```

C :

```
lex([A,B],contraction(VSA,VSB),X):=lex(A,VSA,X),lex(B,VSB,X).
```

Le prédicat d'analyse est :

```
analyser(PH,VS_S):-
    parse(lex(NPH,VS,p),PH),
    writef("%t <=> \n\t%t\n",[NPH,VS]),
    simplifier(VS,VS_S),
    writef("\t%t\n",[VS_S]).
```

E.5.4 Sources de la grammaire généralisée

```
/*****************************************************************************/
/*                                                                           */
/* Christophe DELORD                                                         */
/*                                                                           */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                  */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* Analyse de phrases : grammaire catégorielle généralisée de Lambek-Gentzen */
/*                                                                           */
/*****************************************************************************/
```

```
:-use_module(log_int).
:-use_module(parser).

/**************************************************************************/
/*                                                                        */
/* Définition des catégories syntagmatiques                               */
/*                                                                        */
/**************************************************************************/

:-op(400,yfx,'/').
:-op(400,yfx,'\').

categorie(jean,sn).
categorie(marie,sn).
categorie(pierre,sn).
categorie(anne,sn).
categorie(est,(p\sn)/sa).
categorie(apparente,sa/sp).
categorie(a,sp/sn).
categorie(il,sn).
categorie(juge,(((p\sn)/sa)/sn)).
categorie(incompetente,sa).

lex(lex(MOT,MOT,CAT))-->[MOT],{categorie(MOT,CAT)}.

/**************************************************************************/
/*                                                                        */
/*  R1  f: x/y, a: y -> f(a): x                          applicativité */
/*      a: y, f: x\y -> f(a): x                                           */
/*                                                                        */
/**************************************************************************/

lex([F,A],VSF@VSA,X):=lex(F,VSF,X/Y),lex(A,VSA,Y).
lex([A,F],VSF@VSA,X):=lex(A,VSA,Y),lex(F,VSF,X\Y).

/**************************************************************************/
/*                                                                        */
/*  R2  f: x/y, g: y/z -> lv[f(g(v))]: x/z                 composition */
/*      g: y\z, f: x\y -> lv[f(g(v))]: x\z                                */
/*                                                                        */
/**************************************************************************/

lex([F,G],V#VSF@(VSG@V),X/Z):=lex(F,VSF,X/Y),lex(G,VSG,Y/Z).
lex([G,F],V#VSF@(VSG@V),X\Z):=lex(G,VSG,Y\Z),lex(F,VSF,X\Y).

/**************************************************************************/
/*                                                                        */
/*  R3  f: (x\z)/y -> lv[lw[f(w)(v)]]: (x/y)\z           associativité */
/*      f: (x/y)\z -> lv[lw[f(w)(v)]]: (x\z)/y                            */
/*                                                                        */
/**************************************************************************/

lex(F,V#W#VSF@W@V,(X/Y)\Z):=lex(F,VSF,(X\Z)/Y).
lex(F,V#W#VSF@W@V,(X\Z)/Y):=lex(F,VSF,(X/Y)\Z).
```

```
/****************************************************************************/
/*                                                                          */
/*  R4  a: x, b: (y\x) -> lv[v(a)]: y/(y\x), b: (y\x)               montée */
/*      b: (y/x), a: x -> b: (y/x), lv[v(a)]: y\(y/x)                       */
/*                                                                          */
/****************************************************************************/

lex(A,V#V@VSA,Y/(Y\X)),lex(B,VSB,Y\X):=lex(A,VSA,X),lex(B,VSB,Y\X).
lex(B,VSB,Y/X),lex(A,V#V@VSA,Y\(Y/X)):=lex(B,VSB,Y/X),lex(A,VSA,X).

/****************************************************************************/
/*                                                                          */
/*  R5  f: x/y, g: y/z -> lv[lw[f(v(w))]]: (x/z)/(y/z), g: y/z     division */
/*      g: y\z, f: x\y -> g: y\z, lv[lw[f(v(w))]]: (x\z)\(y\z)     foncteur */
/*                                                                principal */
/****************************************************************************/

lex(F,V#W#VSF@(V@W),(X/Z)/(Y/Z)),lex(G,VSG,Y/Z):=lex(F,VSF,X/Y),lex(G,VSG,Y/Z).
lex(G,VSG,Y\Z),lex(F,V#W#VSF@(V@W),(X\Z)\(Y\Z)):=lex(G,VSG,Y\Z),lex(F,VSF,X\Y).

/****************************************************************************/
/*                                                                          */
/*  R6  g: z/x, f: x/y -> g: z/x, lv[lw[v(f(w))]]: (z/y)\(z/x)     division */
/*      f: x\y, g: z\x -> lv[lw[v(f(w))]]: (z\y)/(z\x), g: z\x     foncteur */
/*                                                               subordonné */
/****************************************************************************/

lex(G,VSG,Z/X),lex(F,V#W#V@(VSF@W),(Z/Y)\(Z/X)):=lex(G,VSG,Z/X),lex(F,VSF,X/Y).
lex(F,V#W#V@(VSF@W),(Z\Y)/(Z\X)),lex(G,VSG,Z\X):=lex(F,VSF,X\Y),lex(G,VSG,Z\X).

/****************************************************************************/
/*                                                                          */
/*  P   t: z/y, a: x, b: y -> t: z/y, b: y, a: x                permutation */
/*      a: x, b: y, t: z\x -> b: y, a: x, t: z\x                            */
/*                                                                          */
/****************************************************************************/

lex(T,VST,Z/Y),lex(B,VSB,Y),lex(A,VSA,X):=lex(T,VST,Z/Y),lex(A,VSA,X),
                                           lex(B,VSB,Y).
lex(B,VSB,Y),lex(A,VSA,X),lex(T,VST,Z\X):=lex(A,VSA,X),lex(B,VSB,Y),
                                          lex(T,VST,Z\X).

/****************************************************************************/
/*                                                                          */
/*  C   a: x, b: x -> (a,b): x                                  contraction */
/*                                                                          */
/****************************************************************************/

lex([A,B],contraction(VSA,VSB),X):=lex(A,VSA,X),lex(B,VSB,X).

/****************************************************************************/
/*                                                                          */
/*  Analyse d'une phrase                                                    */
```

```
/*                                                                          */
/****************************************************************************/
analyser(PH,VS_S):-
    parse(lex(NPH,VS,p),PH),
    writef("%t <=> \n\t%t\n",[NPH,VS]),
    simplifier(VS,VS_S),
    writef("\t%t\n",[VS_S]).

test:-
    analyser([il,juge,incompetente,marie],_).
```

Sources du module de logique

```
/****************************************************************************/
/*                                                                          */
/* Christophe DELORD                                                        */
/*                                                                          */
/* Stage 1997/98 : ENSEEIHT - Traitement du langage naturel                 */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*                                                                          */
/* Logique intensionnelle et lambda calcul                                  */
/*                                                                          */
/****************************************************************************/

:-module(log_int,
    [
        simplifier/2
    ]
).

/*  Syntaxe utilisée pour la représentation des formules
 *
 *      MN          M @ N
 *      lX.f        X # f           X doit etre une variable prolog
 *      lZ.Z(a,b)   Z # Z@a@b
 *      not X       ! X
 *      X and Y
 *      X or Y
 *      X => Y      (implication)
 *      X <=> Y     (équivalance)
 *      pour tout X : P     all(X,P)
 *      il existe X : P     exists(X,P)
 *
 */

% Les priorités sont entre 900 et 950

:-op(900,fy,not).       % C'est la définition de ProLog
:-op(910,yfx,@).        % application @ plus prioritaire que lambda #
:-op(915,xfy,#).
```

```
:-op(920,yfx,and).
:-op(930,yfx,or).
:-op(945,xfy,=>).
:-op(950,xfy,<=>).

simplifier(X,X):-
    var(X),
    !.

simplifier(X@N,X@NS):-
    var(X),
    !,
    simplifier(N,NS).

simplifier((X#F)@X,FS):-
    !,
    simplifier(F,FS).

simplifier(M@N,S):-
    simplifier(M,MS),
    MS\=M,
    !,
    simplifier(MS@N,S).

simplifier(M@N,S):-
    simplifier(N,NS),
    NS\=N,
    !,
    simplifier(M@NS,S).

simplifier(M@N,S):-
    M=..[F|ARGS],
    F\='@', F\='#',                     % cas : (Z@x)@y, ...
    F\='and', F\='or', F\='=>', F\='<=>',
    F\='all', F\='exists',
    F\='int', F\='ext',
    !,
    append(ARGS,[N],NEW_ARGS),
    S=..[F|NEW_ARGS].   % inutile de simplifier S

simplifier(ext(X),ext(X)):-
    var(X),
    !.

simplifier(ext(int(Z)),Z_S):-
    !,
    simplifier(Z,Z_S).

simplifier(T,TS):-
    T=..[FUNC|ARGS],
    simplifier_liste(ARGS,ARGS_S),
    TS=..[FUNC|ARGS_S].

simplifier_liste([],[]).
```

```
simplifier_liste([ARG|ARGS],[ARG_S|ARGS_S]):-
    simplifier(ARG,ARG_S),
    simplifier_liste(ARGS,ARGS_S).
```

Annexe F

Implémentation du simulateur de dialogue

F.1 Déclarations diverses

```
/****************************************************************************/
/*                                                                          */
/* Declarations des operateurs utilises                                     */
/*                                                                          */
/****************************************************************************/

:-module(decls,[]).

:-user:(dynamic
    agents/2,
    max_depth/1,
    inference/6,
    state/6,
    entrance/6,
    invariant/6,
    exit/6,
    perception/6,
    update/7,
    generation/6,
    interdiction/6
).

op([]):-!.
op([Precedence,Type,Name|Others]):-
    op(Precedence,Type,Name),
    op(Others).

:-op(
    [
        1150,   fx,     [game, agent],
        1150,   xfx,    are,

        975,    xfx,    =>,
```

```
        950,    xfx,    [then, when],

        930,    xfx,    #,

        925,    xfx,    |,

        900,    fx,     import,

        700,    xfx,    in,

        500,    yfx,    or,
        400,    yfx,    and,

        350,    xfy,    [b,w,kif,a],

        240,    xfy,    [!,?,'!?',$],

        200,    fx,     -
    ]
).
```

F.2 Compilation des jeux et des agents

```
/**************************************************************************/
/*                                                                        */
/* Analyse et generation du code des jeux et des agents                   */
/*                                                                        */
/**************************************************************************/

:-module(compil,
    [
        analyser/1,
        compiler_kb/6
    ]
).

:-use_module([tools,subst,decls,dialogue,rapport]).

/**************************************************************************/
/*                                                                        */
/* Initialisation du module                                               */
/*                                                                        */
/**************************************************************************/

default_game( default_game =
    [
        entrance = true,
        invariant = true,
        exit = false,
        perception = [],
        update = [],
        generation = [],
        interdiction = []
```

```
    ]
).

default_agent( default_agent =
    [
%       memory = 1000,
        depth = 3,
%       redondances = yes,
        acclist = nobody,       % La chaine doit etre limitee
        kb = [],
        games = [],
        start = default_game
    ]
).

init:-
    empty([game_db,agent_db]),
    default_game(GAME),     recordz(game_db,GAME),
    default_agent(AGENT),   recordz(agent_db,AGENT),
    forall(user:agents(X,_),empty_module(X)),
    retractall(user:agents(_,_)).

%:-init.

/*****************************************************************************/
/*                                                                           */
/* Analyse du fichier de description des jeux et des agents                 */
/*                                                                           */
/*****************************************************************************/

/* Le fichier de description contient des termes (en fait, c'est Prolog qui
 * fait la plus grosse partie de l'analyse). Il faut ensuite fabriquer la
 * description complete des jeux et des agents (importations). A la fin de
 * l'analyse, on a une base de donnees complete des jeux et des agents (c'est
 * la forme developpee de la description initiale) prete a etre compilee en
 * clauses Prolog
 */

/* REMARQUE IMPORTANTE SUR L'IMPORTATION DE REGLES
 * Lors de l'importation d'un jeu, on utilise la cle complete !!!!!
 */

analyser(FICHIER):-
    writef("Analyse du fichier %t\n",[FICHIER]),
    (   open(FICHIER,read,F)
    ->  scan(F)
    ;   writef("ERREUR : Fichier introuvable (%t)\n",[FICHIER])
    ).

scan(F):-
    repeat,
        read(F,TERM),
        (   TERM=end_of_file
        ->  true
```

```
        ;   traite(TERM),
            fail
        ),
    close(F),
    writef("\nAnalyse terminee\n").

dot:-writef(".%f").

traite(TERM0):-
    bracket_to_list(TERM0,TERM),
    (   TERM=(import FILE)                  ->  analyser(FILE)
    ;   TERM=(game KEY=DESCR)               ->  build_game(KEY,DESCR)
    ;   TERM=(agent NAME=DESCR)             ->  build_agent(NAME,DESCR)
    ;   TERM=(BILL and JOHN are speaking)   ->  make_all(BILL,JOHN)
    ;   error("Impossible de traiter %t\n",[TERM])
    ),
    !.

make_all(BILL,JOHN):-
    compiler_agent(BILL),
    compiler_agent(JOHN),
    retractall(user:agents(_,_)),
    user:assertz(agents(BILL,JOHN)),
    user:assertz(agents(JOHN,BILL)).

/****************************************************************************/
/*                                                                          */
/* Description d'un jeu                                                     */
/*                                                                          */
/****************************************************************************/

build_game(KEY,DESCR):-
    (   (   recorded(game_db,KEY=_)
        ->  warning("Le jeu %t est deja defini.\n",[KEY])
        ;   % Construction de la partie importee
            findall(IMPORTED_RULES,
                (   (   GAME_NAME=default_game
                    ;   member(import GAME_NAME,DESCR)
                    ),
                    chk(recorded(game_db,GAME_NAME=IMPORTED_RULES),
                        "Le jeu %t n'est pas defini.\n",[GAME_NAME])
                ),
                ALL_IMPORTED_RULES
            ),
            merge_descr(DESCR,ALL_IMPORTED_RULES,COMPLETE_GAME),
            recordz(game_db,KEY=COMPLETE_GAME)
        )
    ;   error("Impossible de construire le jeu %t.\n",[KEY])
    ),
    dot,
    !.

/****************************************************************************/
/*                                                                          */
```

```
/* Fusion d'une description et d'une liste de description                    */
/*                                                                           */
/*****************************************************************************/

%Il faut traiter le cas ou VAL n'est pas une liste !!!!!

merge_descr(DESCR,ATT=VAL,NEW_DESCR):-
    !,
    (   member(ATT=OLD_VAL,DESCR,REST)
    ->  (   is_list(VAL)
        ->  append(OLD_VAL,VAL,NEW_VAL),
            NEW_DESCR=[ATT=NEW_VAL|REST]
        ;   NEW_DESCR=DESCR
        )
    ;   NEW_DESCR=[ATT=VAL|DESCR]
    ).

merge_descr(DESCR,[],NEW_DESCR):-
    !,
    sublist(not_import,DESCR,NEW_DESCR).

merge_descr(DESCR,[DESCR1|REST],NEW_DESCR):-
    !,
    merge_descr(DESCR,DESCR1,NEW_DESCR1),
    merge_descr(NEW_DESCR1,REST,NEW_DESCR).

not_import(import _):-!,fail.
not_import(_).

/* Version fonctionnelle (qui marche pas encore)
merge_descr(DESCR,ATT=VAL) <-
        member(ATT=OLD_VAL,DESCR,REST) ->
            (is_list(VAL) -> [ATT=(OLD_VAL++VAL)|REST] | DESCR)
        | [ATT=VAL|DESCR].

merge_descr(DESCR,[]) <- subtract(DESCR,[import _]).

merge_descr(DESCR,[DESCR1|REST]) <- merge_descr(merge_descr(DESCR,DESCR1),REST).
*/

/*****************************************************************************/
/*                                                                           */
/* Description d'un agent                                                    */
/*                                                                           */
/*****************************************************************************/

build_agent(NAME,DESCR):-
    (   (   recorded(agent_db,NAME=_)
        ->  warning("L'agent %t est deja defini.\n",[NAME])
        ;   % Construction de la partie importee
            findall(IMPORTED_DESCR,
                (   (   IMPORTED_NAME=default_agent
                    ;   member(import IMPORTED_NAME,DESCR)
                    ),
```

```
                    chk(recorded(agent_db,IMPORTED_NAME=IMPORTED_DESCR),
                        "L'agent %t n'est pas defini.\n",[IMPORTED_NAME])
                ),
                ALL_IMPORTED_DESCR
            ),
            merge_agent_descr(ALL_IMPORTED_DESCR,COMPLETE_IMPORTED_DESCR),
            sublist(not_import,DESCR,NO_IMPORT),
            merge_agent_descr(
                [   COMPLETE_IMPORTED_DESCR,
                    NO_IMPORT
                ],
                COMPLETE_DESCR
            ),
            make_accointances(COMPLETE_DESCR,DESCR_WITH_ACCOINTANCES),
            sublist(not_accointance,COMPLETE_DESCR,NO_ACCOINTANCE),
            merge_agent_descr(
                [   NO_ACCOINTANCE,
                    DESCR_WITH_ACCOINTANCES
                ],
                COMPLETE_DESCR_WITH_ACCOINTANCES
            ),
            make_acc_list(append,NAME,COMPLETE_DESCR_WITH_ACCOINTANCES,
                          FINAL_DESCR),
            recordz(agent_db,NAME=FINAL_DESCR)
        )
    ;   error("Impossible de construire l'agent %t\n",[NAME])
    ),
    dot,
    !.

not_accointance(accointance=_):-!,fail.
not_accointance(_).

make_accointances(DESCR,DESCR_WITH_ACCOINTANCES):-
    (   (   member(accointance=NAME,DESCR),
            chk(recorded(agent_db,NAME=NAME_DESCR),
                "L'agent %t n'est pas defini.\n",[NAME])
        ;   NAME=nobody, NAME_DESCR=[]
        ),
        swap(x,y,NAME_DESCR,DESCRyx),
        subst(
            [
                'B'(X,L) >>> BL if (is_list(L), maplist(add_arg('B'(X)),L,BL)),
                'B'(X,K=V) >>> K=BV if (is_list(V),
                                        maplist(add_arg('B'(X)),V,BV)),
                'B'(X,start=G) >>> start = X b G,
                'B'(X,K=V) >>> K=V,

                'B'(X,(CA|A then B|CB)) >>> ((CA|'B'(X,A)) then ('B'(X,B)|CB)),
                'B'(X,(CA|A then B))    >>> ((CA|'B'(X,A)) then ('B'(X,B))),
                'B'(X,(A then B|CB))    >>> (('B'(X,A)) then ('B'(X,B)|CB)),
                'B'(X,(A then B))       >>> (('B'(X,A)) then ('B'(X,B))),

                'B'(X,(CA|A when B|CB)) >>> ((CA|'B'(X,A)) when ('B'(X,B)|CB)),
```

```
                'B'(X,(CA|A when B))    >>> ((CA|'B'(X,A)) when ('B'(X,B))),
                'B'(X,(A when B|CB))    >>> (('B'(X,A)) when ('B'(X,B)|CB)),
                'B'(X,(A when B))       >>> (('B'(X,A)) when ('B'(X,B))),

                'B'(X,(A->B))       >>> ('B'(X,A)->'B'(X,B)),
                'B'(X,(A and B))    >>> ('B'(X,A) and 'B'(X,B)),
                'B'(X,(A or B))     >>> ('B'(X,A) or 'B'(X,B)),
                'B'(X,-A)           >>> -'B'(X,A),

                'B'(X,A) >>> (X b A)
            ],
            'B'(x,DESCRyx),
            BxDESCRyx
        ),
        forall(
            (   member(start=X b GAME,BxDESCRyx)
            ;   member(games=GAME_LIST,BxDESCRyx),
                member(X b GAME,GAME_LIST)
            ),
            make_game_image(X b GAME)
        ),
        make_acc_list(replace,NAME,BxDESCRyx,BxDESCRyx_WITH_ACCOINTANCES),
        subtract(BxDESCRyx_WITH_ACCOINTANCES,[games=_,start=_],
                 DESCR_WITH_ACCOINTANCES)
    ),
    !.

add_arg(T,A,TA):-
    T=..[F|ARGS],
    append(ARGS,[A],ARGSA),
    TA=..[F|ARGSA].

make_acc_list(REPLACE,NAME,DESCR,NEW_DESCR):-
    member(acclist=ACCLIST,DESCR,REST),
    member(start=START,REST),
    member(depth=DEPTH,REST),
    member(games=GAMES,REST),
    (   REPLACE=replace, ACCLIST=(_,OLDLIST)
    ;   OLDLIST=ACCLIST
    ),
    !,
    NEW_DESCR = [acclist=((NAME,START,DEPTH,GAMES),OLDLIST)|REST].

make_acc_list(_,_,_,[acclist=nobody]).

make_game_image(X b GAME):-
    (   recorded(game_db,X b GAME=_)        % deja fait, ok
    ;   chk(recorded(game_db,GAME=DESCR),
            "Le jeu %t n'est pas defini.\n",[GAME]),
        swap(x,y,DESCR,DESCRyx),
        subst(
            [
                [generation=_|L] >>> L,
                [interdiction=_|L] >>> L,
```

```
                'B'(X,perception=PERCEPTION) >>> perception='BP'(X,PERCEPTION),

                'B'(X,L) >>> BL if (is_list(L), maplist(add_arg('B'(X)),L,BL)),
                'BP'(X,L) >>> BL if (is_list(L),
                                     maplist(add_arg('BP'(X)),L,BL)),
                'B'(X,K=V) >>> K=BV if (is_list(V),
                                        maplist(add_arg('B'(X)),V,BV)),
                'B'(X,K=V) >>> K='B'(X,V),

                ('B'(X,(P=>Q|C#F))) >>> ('B'(X,P)=>'B'(X,Q)|C#F) if dot,
                ('B'(X,(P=>Q|C)))   >>> ('B'(X,P)=>'B'(X,Q)|C) if dot,
                ('B'(X,(P=>Q#F)))   >>> ('B'(X,P)=>'B'(X,Q)#F) if dot,
                ('B'(X,(P=>Q)))     >>> ('B'(X,P)=>'B'(X,Q)) if dot,

                ('BP'(X,(P=>Q|C#F)))    >>> (P=>'B'(X,Q)|C#F) if dot,
                ('BP'(X,(P=>Q|C)))      >>> (P=>'B'(X,Q)|C) if dot,
                ('BP'(X,(P=>Q#F)))      >>> (P=>'B'(X,Q)#F) if dot,
                ('BP'(X,(P=>Q)))        >>> (P=>'B'(X,Q)) if dot,

                'B'(X,(A->B))       >>> ('B'(X,A)->'B'(X,B)),
                'B'(X,(A and B))    >>> ('B'(X,A) and 'B'(X,B)),
                'B'(X,(A or B))     >>> ('B'(X,A) or 'B'(X,B)),
                'B'(X,-A)           >>> -'B'(X,A),

                'B'(X,A) >>> (X b A)
            ],
            'B'(X,DESCRyx),
            BxDESCRyx
        ),
        recordz(game_db,X b GAME=BxDESCRyx)
    ),
    !.


/*****************************************************************************/
/*                                                                           */
/* Fusion de la description de plusieurs agents                              */
/*                                                                           */
/*****************************************************************************/

merge_agent_descr(DESCR_LIST,COMPLETE_DESCR):-
    merge_agents(DESCR_LIST,[],COMPLETE_DESCR).

merge_agents([],DESCR,DESCR):-!.

merge_agents([DESCR|REST],BEFORE,AFTER):-
    !,
    merge2(DESCR,BEFORE,CURRENT),
    merge_agents(REST,CURRENT,AFTER).

merge2([],DESCR,DESCR):-!.

merge2([ATT=VAL|DESCR],BEFORE,AFTER):-
```

```
    !,
    (   is_list(VAL)
    -> (   member(ATT=OLD_VAL,BEFORE,REST)
        ->
            append(OLD_VAL,VAL,CURRENT_VAL),
            CURRENT=[ATT=CURRENT_VAL|REST]
        ;   CURRENT=[ATT=VAL|BEFORE]
        )
    ;   (   member(ATT=_,BEFORE,REST)
        ->  CURRENT=[ATT=VAL|REST]
        ;   CURRENT=[ATT=VAL|BEFORE]
        )
    ),
    merge2(DESCR,CURRENT,AFTER).


/*****************************************************************************/
/*                                                                           */
/* Affichage de BD initiale                                                  */
/*                                                                           */
/*****************************************************************************/

afficher_bd_initiale:-
    writef("Base de donnees initiale :\n\n"),
    (   writef("Jeux :\n\n"), CLE=game_db
    ;   writef("Agents :\n\n"), CLE=agent_db
    ),
    recorded(CLE,ITEM),
    numbervars(ITEM,0,_),
    aff_ensemble(ITEM),
    nl,
    fail.
afficher_bd_initiale:-nl.

aff_ensemble(NOM=ENS):-
    writef("%t =\n{\n",[NOM]),
    member(ATT=VAL,ENS),
    writef("\t%t = ",[ATT]),
    aff_val(VAL),
    nl,
    fail.
aff_ensemble(_):-
    writef("}.\n").

aff_val(VAL):-
    (   is_list(VAL)
    ->  (   writef("\n\t{\n"),
            member(E,VAL),
            writef("\t\t%t\n",[E]),
            fail
        ;   writef("\t}\n")
        )
    ;   writef("%t\n",[VAL])
    ).
```

```
/*****************************************************************************/
/*                                                                           */
/* Simplification d'une clause avant assertion                              */
/*                                                                           */
/*****************************************************************************/

:-module_transparent clean_assertz/2.

clean_assertz(M,(HEAD:-BODY)):-
    simpl(BODY,CLEAN_BODY),
    (   CLEAN_BODY=true, M:assertz(HEAD)
    ;   CLEAN_BODY=fail
    ;   M:assertz(HEAD:-CLEAN_BODY)
    ),
    !.

simpl(A,B):-
    subst(
        [
            (true,X) >>> X,
            (X,true) >>> X,

            state(_,_,_,_,_,F,true) >>> F=9999,
            state(_,_,_,_,_,_,false) >>> fail,

            true and X >>> X,
            X and true >>> X,
            true or X >>> true,
            X or true >>> true,
            false and X >>> false,
            X and false >>> false,
            false or X >>> X,
            X or false >>> X,
            -true >>> false,
            -false >>> true,

            -(-X) >>> X,
            -(X and Y) >>> -X or -Y,
            -(X or Y) >>> -X and -Y,

            state(X,Y,T,K,D,F,E1 and E2) >>>
                (   state(X,Y,T,K,D,F1,E1),
                    state(X,Y,T,K,D,F2,E2),
                    F is min(F1,F2)
                ),

            F is FE*1 >>> F=FE
        ],
        A,B
    ).

expand_sequences(A,B):-
```

```
    subst(
        [
            x b [C|SC] >>> x b ([C|SC]:t),
            x b ([C]:T) >>> x b (C:T),
            x b ([C|SC]:T) >>> x b (SC:T) and x b (C:T-L) if length(SC,L)
        ],
        A,B
    ).

/****************************************************************************/
/*                                                                          */
/* Compilation d'un agent                                                   */
/*                                                                          */
/****************************************************************************/

/* La description d'un agent se fait dans un module dynamique identifie
 * par le nom de l'agent. Tous les jeux connus de l'agent sont aussi
 * compiles dans ce module.
 */

/* Les predicats produits prennent en parametre :
 *      X, Y        : Nom des agents
 *      T           : Instant courant
 *      K           : Cle du jeu en cours
 *      D           : Profondeur restante disponible
 */

compiler_agent(X):-
    writef("\nCompilation de l'agent %t\n",[X]),
    dot,
    chk(recorded(agent_db,X=DESCR),"L'agent %t n'est pas defini.\n",[X]),
    empty_module(X),
    expand_sequences(DESCR,DESCR1),
    subst([x>>>X,y>>>Y,t>>>T],DESCR1,DESCR2),
    forall(member(ITEM,DESCR2),compiler_agent(X,Y,T,ITEM)),
    completer(X),
    writef("\nCompilation terminee\n").

%compiler_agent(_,_,_,_):-trace,fail.

compiler_agent(X,Y,T,kb=KB):-
    !,
    forall(member(ITEM,KB),compiler_kb(X,Y,T,ITEM)).

/*
compiler_agent(X,Y,T,games=GAMES):-
    !,
    forall(member(ITEM,GAMES),compiler_game(X,Y,T,ITEM)).

compiler_agent(X,Y,T,start=START_GAME):-
    !,
    X:asserta(start_game(START_GAME)),
    compiler_game(X,Y,T,START_GAME).
```

```
compiler_agent(X,_,_,depth=MAX_DEPTH):-
    !,
    dot,
    X:asserta(max_depth(MAX_DEPTH)).
*/

compiler_agent(_X,_Y,_T,acclist=nobody):-
    !.

compiler_agent(X,Y,T,acclist=((NAME,START,DEPTH,GAMES),ACCLIST)):-
    !,
    dot,
    compiler_agent(X,Y,T,acclist=ACCLIST),
    X:assertz(start_game(NAME,START)),
    X:assertz(max_depth(NAME,DEPTH)),
    X:assertz(accointance(NAME)),
    forall(member(ITEM,GAMES),
        (   compiler_game(X,Y,T,ITEM),
            X:asserta(one_game(NAME,ITEM))
        )
    ),
    compiler_game(X,Y,T,START).

compiler_agent(_,_,_,games=_):-!.
compiler_agent(_,_,_,start=_):-!.
compiler_agent(_,_,_,depth=_):-!.

compiler_agent(X,_,_,ITEM):-
    error("Impossible de compiler %t de l'agent %t\n",[ITEM,X]).


/****************************************************************************/
/*                                                                          */
/* Compilation d'une base de connaissance                                   */
/*                                                                          */
/****************************************************************************/

/* Compilation d'une regle d'inference : assertZ <- a la fin de la base !!!
 *
 * C0 | P then Q | C1
 * Si C0 ok alors instancier Q puis P puis verifier C1 (chainage arriere)
 * Cette regle est traduite par :
 * inference(X,Y,T,K,D,F,Q):-
 *                          C0,
 *                          D1 is D-1,
 *                          state(X,Y,T,K,D1,F,P),
 *                          C1,
 *                          memoriser_etat(X,Q,F,_).
 *
 * De plus, pour utiliser une regle d'inference en chainage avant,
 * il faut la convertir en une regle de mise a jour des etats mentaux
 * visible dans tous les jeux (K non instancie)
 * C0 | P then Q | C1 devient P => Q | C1
 * (La condition C0 n'est utile qu'en chainage arriere)
```

```
 *
 * C1 | Q when P | C0
 * Regle d'inference qui ne fonctionneme qu'en chainage arriere
 * La traduction est la meme que 'then' sans la compilation de la
 * regle de mise a jour correspondante
 */

compiler_kb(X,Y,T,(A then B)):-
    !,
    dot,
    (   A=(C0|P)
    ;   P=A, C0=true
    ),
    (   B=(Q|C1)
    ;   Q=B, C1=true
    ),
    clean_assertz(X,(inference(X,Y,T,K,D,F,Q):-
                        C0,
                        D1 is D-1,
                        state(X,Y,T,K,D1,F,P),
                        C1,
                        dialogue:memoriser_etat(X,Q,F,_)
                    )),
    compiler_update(X,Y,T,_,(P=>Q|C1)).

compiler_kb(X,Y,T,(B when A)):-
    !,
    dot,
    (   A=(P|C0)
    ;   P=A, C0=true
    ),
    (   B=(C1|Q)
    ;   Q=B, C1=true
    ),
    clean_assertz(X,(inference(X,Y,T,K,D,F,Q):-
                        C0,
                        D1 is D-1,
                        state(X,Y,T,K,D1,F,P),
                        C1,
                        dialogue:memoriser_etat_sans_propager(X,Q,F,_)
                    )).


/*
compiler_kb(X,Y,T,(A then B)):-
    !,
    dot,
    (   A=(C0|P)
    ;   P=A, C0=true
    ),
    (   B=(Q|C1)
    ;   Q=B, C1=true
    ),
    clean_assertz(X,(inference(X,Y,T,K,D,F,Q):-
```

```
                        C0,
                        D1 is D-1,
                        state(X,Y,T,K,D1,F,P),
                        C1,
                        dialogue:memoriser_etat(X,Q,F,_)
                    )),
    compiler_update(X,Y,T,_,(P=>Q|C1)).
*/


/* Compilation d'un etat mental : assertA <- au debut de la base !!!
 *
 * E etat mental est traduit par :
 * state(X,Y,T,K,D,F,ST):-flag(<ref>,F,F+1).
 * sachant qu'il y a dans la base : state(...,ST):-inference(...,ST).
 * (chainage arriere)
 */

compiler_kb(X,Y,T,ST):-
    compiler_kb(X,Y,T,ST,1,_).

% ICI IL FAUDRA PENSER A AJOUTER LE POIDS DE ST A LA PLACE MEMOIRE
% OCCUPPEE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

compiler_kb(X,Y,T,ST0,F0,ST):-      % construit le terme simplifie ST
    subst([-(-(P)) >>> P],ST0,ST),
    (   ifdef(afficher_rapport)
    ->  enregistrer_modification(X,ST)
    ;   true
    ),
    gen_ref(REF),
    X:flag(REF,_,F0),
    X:asserta(state(X,Y,T,_K,_D,F,ST):-flag(REF,F,F+1)).


/*****************************************************************************/
/*                                                                           */
/* Compilation d'un jeu                                                      */
/*                                                                           */
/*****************************************************************************/

/*
compiler_game(X,Y,T,GAME):-
    recorded(game_db,THE_GAME=DESCR),
    functor(THE_GAME,GAME,_),
    expand_sequences(DESCR,DESCR1),
%   subst(DESCR1,[X/x,Y/y,T/t],DESCR2),
    subst([x>>>X,y>>>Y,t>>>T],DESCR1,DESCR2),
    forall(member(ITEM,DESCR2),compiler_game(X,Y,T,THE_GAME,ITEM)).
*/

compiler_game(X,Y,T,GAME):-
    recorded(game_db,ONE_GAME=ONE_DESCR),
    subst([x>>>X,y>>>Y],ONE_GAME,GAME),
```

```
    expand_sequences(ONE_DESCR,DESCR),
    subst([x>>>X,y>>>Y,t>>>T],DESCR,DESCR1),
    forall(member(ITEM,DESCR1),compiler_game(X,Y,T,GAME,ITEM)).


/***************************************************************************/
/*                                                                         */
/* Compilation des differents types de regles                              */
/*                                                                         */
/***************************************************************************/

compiler_game(X,Y,T,K,perception=PERCEPTIONS):-
    forall(member(ITEM,PERCEPTIONS),compiler_perception(X,Y,T,K,ITEM)).

compiler_game(X,Y,T,K,update=UPDATES):-
    forall(member(ITEM,UPDATES),compiler_update(X,Y,T,K,ITEM)).

compiler_game(X,Y,T,K,generation=GENERATIONS):-
    forall(member(ITEM,GENERATIONS),compiler_generation(X,Y,T,K,ITEM)).

compiler_game(X,Y,T,K,interdiction=INTERDICTIONS):-
    forall(member(ITEM,INTERDICTIONS),compiler_interdiction(X,Y,T,K,ITEM)).

/* Conditions d'entree, invariant, de sortie
 *
 * entrance = COND  -->  entrance(X,Y,T,K,D,F,GAME):-state(X,Y,T,K,D,F,COND)
 */

compiler_game(X,Y,T,K,entrance=COND):-
    dot,
    clean_assertz(X,(entrance(X,Y,T,OLD_K,D,F,K):-state(X,Y,T,OLD_K,D,F,COND))).

compiler_game(X,Y,T,K,invariant=COND):-
    dot,
    clean_assertz(X,(invariant(X,Y,T,K,D,F):-state(X,Y,T,K,D,F,COND))).

compiler_game(X,Y,T,K,exit=COND):-
    dot,
    clean_assertz(X,(exit(X,Y,T,K,D,F):-state(X,Y,T,K,D,F,COND))).

/* Regles de perception
 *
 * LC => E :
 * perception(X,Y,T,K,D,M,E):-coups(COUPS),match(LC,COUPS),COND,max_force(M).
 */

compiler_perception(X,Y,T,K,(LC=>E)):-
    dot,
    (   E=(ETAT|COND)
    ;   ETAT=E, COND=true
    ),
    clean_assertz(X,(perception(X,Y,T,K,_D,M,ETAT):-coups(COUPS),
                                        match(LC,COUPS),COND,max_force(M))).
```

```
/* Regles de mise a jour
 *
 * A => B | C
 * update(X,Y,T,K,D,A0,F,B):-state(X,Y,T,K,D,F,A1),C.  % A0=etat qui apparait
 * pour tout A0 in A, A1 etant A sachant A0
 */

/* WARNING !!!!
 * subst doit utiliser l'unification AVEC controle d'occurence
 * (voir tools.pl)
 */

compiler_update(X,Y,T,K,(LHS=>RHS)):-
    dot,
    A=LHS,
    (   RHS=(B|C)
    ;   B=RHS, C=true
    ),
    forall(A0 in A,
        (   %subst(A,[true/A0],A1),   % si l'on fait la substitution,
                                      % on perd la force de A0 !
            %subst([A0>>>true],A,A1),
            A1=A,
            clean_assertz(X,(update(X,Y,T,K,D,A0,F,B):-
                             state(X,Y,T,K,D,F,A1),C))
        )
    ).

/* Regles de generation
 *
 * A => B | C # FR                  FR = Force de la regle
 * generation(X,Y,T,K,D,F,B):-state(X,Y,T,K,D,FE,A),C,F is FE*FR.
 */

compiler_generation(X,Y,T,K,(LHS=>RHS)):-
    dot,
    A=LHS,
    (   RHS=(B|C#FR)
    ;   RHS=(B|C), FR=1
    ;   RHS=(B#FR), C=true
    ;   B=RHS, C=true, FR=1
    ),
    clean_assertz(X,(generation(X,Y,T,K,D,F,B):-state(X,Y,T,K,D,FE,A),
                                                C,F is FE*FR)).

/* Regles d'interdiction
 *
 * A => -B | C # FR
 * interdiction(X,Y,T,K,D,F,B):-state(X,Y,T,K,D,FE,A),C,F is FE*FR.
 */

compiler_interdiction(X,Y,T,K,(LHS=>RHS)):-
    dot,
    A=LHS,
```

```
    (   RHS=(-B|C#FR)
    ;   RHS=(-B|C), FR=1
    ;   RHS=(-B#FR), C=true
    ;   -B=RHS, C=true, FR=1
    ),
    clean_assertz(X,(interdiction(X,Y,T,K,D,F,B):-state(X,Y,T,K,D,FE,A),
                                              C,F is FE*FR)).
```

F.3 Simulation du dialogue

```
/****************************************************************************/
/*                                                                          */
/* Execution des clauses compilees par le module compil                     */
/*                                                                          */
/****************************************************************************/

:-module(dialogue,
    [
        completer/1,
        dialoguer/0
    ]
).

:-use_module([rapport,pasapas,windows,subst]).


init:-
    index(state(1,0,0,0,0,0,1)),
    index(inference(1,0,0,0,1,0,1)),
    index(entrance(1,0,0,1,0,0,0)),
    index(invariant(1,0,0,1,0,0)),
    index(exit(1,0,0,1,0,0)),
    index(perception(1,0,0,1,0,0,0)),
    index(update(1,0,0,1,0,1,0,0)),
    index(generation(1,0,0,1,0,0,0)),
    index(interdiction(1,0,0,1,0,0,1)),
    index(one_game(1,1)).


/****************************************************************************/
/*                                                                          */
/* Liste des clauses a rajouter a chaque agent                              */
/*                                                                          */
/****************************************************************************/

/* Syntaxe :
 *      a : clause          asserta(clause)
 *      z : clause          assertz(clause)
 */


% DANS LES CLAUSES states RAJOUTEES, IL FAUDRA APPELER memoriser_etat SI
```

```
% ON STOCKE LES REDONDANCES
% C'EST EN FAIT DANS memoriser_etat QUE L'ON TRAITERA OU NON LE STOCKAGE
% DES REDONDANCES

moteur(X,
    [
        z:(state(X,_,_,_,_,9999,true):-!),  % Quelle est la force de true
        z:(state(X,_,_,_,_,_,false):-!,fail),
        z:(state(X,_,_,_,_,_,X):-var(X),!,fail),
        z:(state(X,Y,T,K,D,F,A):-
                subst:subst([-(-P)>>>P],A,B),
                A\=B,
                !,
                state(X,Y,T,K,D,F,B)
            ),
        z:(state(X,Y,T,K,D,F,A and B):-state(X,Y,T,K,D,F1,A),
                                       state(X,Y,T,K,D,F2,B),F is min(F1,F2)),
        z:(state(X,Y,T,K,D,F,A or B):-state(X,Y,T,K,D,F,A);
                                      state(X,Y,T,K,D,F,B)),
        z:(state(X,Y,T,K,D,F,-(-A)):-state(X,Y,T,K,D,F,A)),
        z:(state(X,Y,T,K,D,1,-A):- \+ state(X,Y,T,K,D,_,A)),
                                       % Quelle est la force d'un fait nie
        z:(state(X,Y,T,K,D,F,X b (C:U)):- nonvar(U), \+ integer(U),
                                    V is U, state(X,Y,T,K,D,F,X b (C:V))),
        z:(state(X,Y,T,K,D,F,ST):-inference(X,Y,T,K,D,F,ST)),

%       z:interdiction(_,_,_,_,_,0,_),

        a:(inference(_,_,_,_,0,_,_):-!,fail),

        z:(match(C,[C|_]):-!),   % a revoir pour les sequences de coups

        z:(max_force(F):-dialogue:max_force(X,F)),

        (:-dynamic agents/2, max_depth/2, inference/7, state/7),
        (:-dynamic entrance/7, invariant/6, exit/6, one_game/2),
        (:-dynamic perception/7, update/8, generation/7, interdiction/7),
        (:-dynamic jeu_courant/2, cur_acc/1),
        (:-dynamic max_force/1)

    ]
).

/***************************************************************************/
/*                                                                         */
/* Completer la definition d'un agent                                      */
/*                                                                         */
/***************************************************************************/

/*
completer(X):-
    moteur(X,CLAUSES),
    forall(member(OU:CLAUSE,CLAUSES),
        (   OU=z
```

```
        ->  X:assertz(CLAUSE)
        ;   X:asserta(CLAUSE)
        )
    ),
    forall(member((:-INIT),CLAUSES),X:INIT).
*/

completer(X):-
    moteur(X,CLAUSES),
    (   member(a:CLAUSE,CLAUSES), X:asserta(CLAUSE), fail
    ;   member(z:CLAUSE,CLAUSES), X:assertz(CLAUSE), fail
    ;   member((:-INIT),CLAUSES), X:INIT, fail
    ;   true
    ).

/****************************************************************************/
/*                                                                          */
/* Dialogue                                                                 */
/*                                                                          */
/****************************************************************************/

dialoguer:-
    agents(BILL,JOHN),
    (   feature(unix,true)
    ->  win_init(160,50),
        (   ifdef(debug_window)
        ->  BOTTOM=45
        ;   BOTTOM=50
        ),
        win_open(BILL,1,1,65,BOTTOM,BILL),
        win_open(JOHN,95,1,160,BOTTOM,JOHN),
        win_open(coups,65,1,95,BOTTOM,"Coups"),
        ifdef(debug_window,win_open(debug,1,55,160,60,"Debug"))
    ;   win_init(145,40),
        (   ifdef(debug_window)
        ->  BOTTOM=35
        ;   BOTTOM=40
        ),
        win_open(BILL,1,1,55,BOTTOM,BILL),
        win_open(JOHN,90,1,145,BOTTOM,JOHN),
        win_open(coups,55,1,90,BOTTOM,"Coups"),
        ifdef(debug_window,win_open(debug,1,35,145,40,"Debug"))
    ),
    win_link([BILL,JOHN,coups]),
    % Il faut propager les etats presents initialement
    % Il n'apparaissent pas donc ne declenchent pas les regles de maj
    findall(
        X:state(X,Y,T,K,D,F,E),
        (   clause(X:state(X,Y,T,K,D,F,E),flag(FLAG,_,_),REF),
            flag(FLAG,F,F),
            erase(REF)
        ),
        INIT_KB
    ),
```

```
forall(
    member(X:state(X,Y,T,K,D,F,E),INIT_KB),
    memoriser_etat(X,E,F,_)
),
forall(
    (   agents(X,_),
        clause(X:state(X,_,_,_,_,_,E),flag(_,_,_))
    ),
    (   numbervars(E,0,_),
        win_write(X,"%t\n",[E])
    )
),
synchroniser,
retractall(user:coups(_)),
asserta(user:coups([])),
init_rapport,
repeat,
    agents(X,Y),
    /****************************************
        date(T),
        X:max_depth(D),
        redefine(synchro),
        changement_de_jeu(X,Y,T,D),
        ignore(generer_un_coup(X,Y,T,D,CX)),
        undefine(synchro),
        (   ifdef(afficher_rapport)
        ->  afficher_etat_courant(T)
        ;   true
        ),
        dire(X,CX),
        once(regles_de_perception(X,Y,T)),
        once(regles_de_perception(Y,X,T)),
        synchroniser,
        draw_windows,
        pas_a_pas,
        ligne_de_separation,
    ****************************************/

        date(T),
        redefine(synchro),
        (   X:accointance(NAME),
                X:rasserta(cur_acc(NAME)),
                X:max_depth(NAME,D),
                changement_de_jeu(X,Y,T,D),
            fail
        ;   true
        ),
        (   Y:accointance(NAME),
                Y:rasserta(cur_acc(NAME)),
                Y:max_depth(NAME,D),
                changement_de_jeu(Y,X,T,D),
            fail
        ;   true
        ),
```

```
        ignore(generer_un_coup(X,Y,T,D,CX)),
        undefine(synchro),
        ifdef(afficher_rapport,afficher_etat_courant(T)),
        dire(X,CX),
        (   X:accointance(NAME),
                X:rasserta(cur_acc(NAME)),
                once(regles_de_perception(X,Y,T)),
            fail
        ;   true
        ),
        (   Y:accointance(NAME),
                Y:rasserta(cur_acc(NAME)),
                once(regles_de_perception(Y,X,T)),
            fail
        ;   true
        ),
        synchroniser,
        draw_windows,
        pas_a_pas,
        ligne_de_separation,

    fin_du_dialogue,
    !.

fin_du_dialogue:-
    coups([$ X, $ Y|_]),
    agents(X,Y).

synchroniser:-
    agents(BILL,JOHN),
    repeat,
        win_pos(BILL,_,Y_BILL),
        win_pos(JOHN,_,Y_JOHN),
        (   Y_BILL<Y_JOHN
        ->  win_nl(BILL),
            fail
        ;   true
        ),
        (   Y_JOHN<Y_BILL
        ->  win_nl(JOHN),
            fail
        ;   true
        ),
    win_pos(BILL,_,Y_BILL),
    repeat,
        win_pos(coups,_,Y_COUPS),
        (   Y_COUPS<Y_BILL
        ->  win_nl(coups),
            fail
        ;   true
        ),
    !.

ligne_de_separation:-
```

```
    agents(BILL,JOHN),
    windows:maxXY(MAXX,_),
    for(_,1,MAXX>>2,
        (   win_write(BILL,"- "),
            win_write(JOHN,"- "),
            win_write(coups,"- ")
        )
    ),
    win_nl(BILL),
    win_nl(JOHN),
    win_nl(coups),
    !.

/****************************************************************************/
/*                                                                          */
/* Force maximale des etats mentaux                                         */
/*                                                                          */
/****************************************************************************/

max_force(X,MF):-
    flag(max_force,_,0),
    (   X:clause(state(X,_,_,_,_,_,_),flag(REF,_,_)),
            flag(REF,F,F),
            flag(max_force,MF,max(F,MF)),
        fail
    ;   flag(max_force,MF,MF)
    ).

/*
fflag(REF,F1,F2):-
    flag(REF,F1,F2),
    F is F2,
    max_force(MF),
    (   F>MF
    ->  retract(max_force(_)),
        asserta(max_force(F))
    ;   true
    ).
*/

/****************************************************************************/
/*                                                                          */
/* Changement de jeu                                                        */
/*                                                                          */
/****************************************************************************/

changement_de_jeu(X,Y,T,D):-
    sortir_du_jeu_courant(X,Y,T,D),
    entrer_dans_un_jeu(X,Y,T,D).

sortir_du_jeu_courant(X,Y,T,D):-
    X:cur_acc(NAME),
    (   once(X:jeu_courant(NAME,JEU_COURANT)),
        (   X:exit(X,Y,T,JEU_COURANT,D,_)
```

```
        ;   \+ X:invariant(X,Y,T,JEU_COURANT,D,_)
        )
    ->  retract(X:jeu_courant(NAME,JEU_COURANT)),
        win_write(X,"%t sort du jeu %t\n",[/*X*/NAME,JEU_COURANT]),
        win_nl(Y),
        win_nl(coups),
        sortir_du_jeu_courant(X,Y,T,D)
    ;   true
    ),
    !.

entrer_dans_un_jeu(X,Y,T,D):-
    X:cur_acc(NAME),
    (   (   X:jeu_courant(NAME,JEU_COURANT)
        ;   X:start_game(NAME,JEU_COURANT)
        ),
        !,
        findall(F:JEU,
            (   X:one_game(NAME,JEU),
                X:entrance(X,Y,T,JEU_COURANT,D,F,JEU),
                \+ X:jeu_courant(NAME,JEU),
                \+ X:start_game(NAME,JEU)
            ),
            JEUX
        ),
        merge_sort(JEUX,JEUX_TRIES),
        (   member(_:JEU,JEUX_TRIES),
            asserta(X:jeu_courant(NAME,JEU),REF),
            win_write(X,"%t rentre dans le jeu %t\n",[/*X*/NAME,JEU]),
            win_nl(Y),
            win_nl(coups),
            % Il faut encore pouvoir y rester
            (   (   X:exit(X,Y,T,JEU,D,_)
                ;   \+ X:invariant(X,Y,T,JEU,D,_)
                )
            ->  % on sort immediatement du jeu
                erase(REF),
                win_write(X,"... et en resort aussitot.\n"),
                win_nl(Y),
                win_nl(coups),
                fail                % on en prend un autre
            ;   true    % ok, on y reste
            )
        ;   true    % cas ou la liste est vide !!!
        )
    ;   true
    ),
    !.


/******************************************************************************/
/*                                                                            */
/* Gestion des coups                                                          */
/*                                                                            */
```

```
/****************************************************************************/

date(T):-coups(COUPS),length(COUPS,T).

dire(X,CX):-
    (   var(CX)
    ->  win_write(coups,"Coup vide de %t\n",[X])
    ;   retract(user:coups(COUPS)),
        assert(user:coups([CX|COUPS])),
        win_write(coups,"%t\n",[CX])
    ),
    agents(BILL,JOHN),
    win_nl(BILL),
    win_nl(JOHN),
    !.


/****************************************************************************/
/*                                                                          */
/* Generation d'un coup                                                     */
/*                                                                          */
/****************************************************************************/

generer_un_coup(X,Y,T,D,CX):-
    asserta(X:cur_acc(X),REF),
    (   X:jeu_courant(X,K)
    ;   X:start_game(X,K)
    ),
    !,                     % Avec !, on ne traite que le dernier jeu
    findall(F:COUP,
        (   X:generation(X,Y,T,K,D,FG,COUP),
            findall(FI,X:interdiction(X,Y,T,K,D,FI,COUP),FIS),
            max(FIS,FIM),
            F is FG-FIM,
            F > 0
        ),
        COUPS
    ),
    merge_sort(COUPS,COUPS_TRIES),
    ifdef(debug_window,
          win_write(debug,"Coups generables : %t\n",[COUPS_TRIES])),
    member(_:CX,COUPS_TRIES),
    erase(REF).


/****************************************************************************/
/*                                                                          */
/* Regles de perception                                                     */
/*                                                                          */
/****************************************************************************/

regles_de_perception(X,Y,T):-
    X:cur_acc(NAME),
    (   X:jeu_courant(NAME,K)
```

```
    ;   X:start_game(NAME,K)
    ),
    !,
    X:max_depth(NAME,D),
    forall(X:perception(X,Y,T,K,D,F,E),memoriser_etat(X,E,F,_)).


/***************************************************************************/
/*                                                                         */
/* Memorisation d'un etat mental                                           */
/*                                                                         */
/***************************************************************************/

memoriser_etat(X,E,F,E1):-memoriser(X,E,F,E1,true).
memoriser_etat_sans_propager(X,E,F,E1):-memoriser(X,E,F,E1,fail).

memoriser(X,A and B,F,A1 and B1,PROPAGER):-
    !,
    memoriser(X,A,F,A1,PROPAGER),
    memoriser(X,B,F,B1,PROPAGER).

memoriser(X,A or B,F,A1 or B1,PROPAGER):-
    !,
    user:agents(X,Y),
    date(T),
    (   renforcer(X,A or B,F)
    ->  A1=A, B1=B
    ;   compiler_kb(X,Y,T,A or B,F,A1 or B1),
        (   PROPAGER
        ->  propager_etat(X,A1 or B1)
        ;   true
        )
    ).

% IL FAUT DISTINGUER LES ETATS QUI APPARAISSENT ET DISPARAISSENT

memoriser(X,E,F,E1,PROPAGER):-
    subst([-(-P)>>>P],E,E1),
    liberer(X,-E1),
    (   clause(X:state(X,_,_,_,_,_,E1),flag(REF,_,_))
    ->  X:flag(REF,F0,F0+F)     % renforcement de l'etat
    ;   % faire_de_la_place(X,E1),
        (   ifdef(afficher_nouveaux),
            E1\=(_ kif _)
        ->  %numbervars(E1,0,_),
            win_write(X,"%t\n",[E1]),
            (   ifdef(synchro)
            ->  agents(X,Y),
                win_nl(Y),
                win_nl(coups)
            ;   true
            )
        ;   true
        ),
```

```
        compiler_kb(X,_Y,_T,E1,F,_),
        (   PROPAGER
        ->  propager_etat(X,E1)
        ;   true
        )
    ),
    !.

memoriser(X,E,F,E,PROPAGER):-
    warning("%t\n",[memoriser(X,E,F,_,PROPAGER)]).      % debug

propager_etat(X,E):-
    % Ici, on applique les regles de mise a jour
%   (   X:jeu_courant(K)
%   ;   X:start_game(K)
%   ),
%   !,
%   X:max_depth(D),
%   forall(X:update(X,Y,T,K,D,E,FNE,NE),memoriser_etat(X,NE,FNE,_)).
    (   X:cur_acc(NAME)
    ;   NAME=X
    ),
    X:max_depth(NAME,D),
    forall(
        (
            (   X:jeu_courant(NAME,K)
            ;   X:start_game(NAME,K)
            ),
            X:update(X,_Y,_T,K,D,E,FNE,NE)
        ),
        memoriser_etat(X,NE,FNE,_)
    ),
    !.


liberer(X,-(-E)):-
    !,
    liberer(X,E).

liberer(X,E):-
    forall(clause(X:state(X,_,_,_,_,_,E),flag(_,_,_),REF),erase(REF)).

% Maintenir la force d'une disjonction : Renforcer tous les termes connus
% Echec si aucun terme n'a pu etre renforce

renforcer(X,A or B,F):-
    !,
    (   renforcer(X,A,F)            % on peut renforcer A
    ->  (   renforcer(X,B,F)        % donc b n'a pas d'importance
        ;   true
        )
    ;   renforcer(X,B,F)            % sinon il faut pouvoir renforcer B
    ).
```

```
renforcer(X,A,F):-
    clause(X:state(X,_,_,_,_,_,A),flag(REF,_,_)),
    X:flag(REF,F0,F0+F-F+1).
```

F.4 Affichage du dialogue

```
/*****************************************************************************/
/*                                                                           */
/* Generation du rapport de l'evolution des etats mentaux                    */
/*                                                                           */
/*****************************************************************************/

:-module(rapport,
    [
        init_rapport/0,
        enregistrer_modification/2,     % param : X,E
        afficher_etat_courant/1
    ]
).

/* Le rapport est constitue pendant le dialogue en appelant le predicat
 * enregistrer_modification/2
 * On memorise les couples (X,E) avec la cle rapport.
 * On memorise chaque etape avec la cle etape.
 */

init_rapport:-
    empty([rapport,etape]).

enregistrer_modification(X,E):-
    enlever(X,-E),
    recordz(rapport,(X,E)).

enlever(X,-(-E)):-
    !,
    enlever(X,E).

enlever(X,E):-
    (   recorded(rapport,(X,E),REF)
    ->  erase(REF)
    ;   true
    ).

afficher_etat_courant(T):-
    (   agents(X,_),
            findall(ETAT,recorded(rapport,(X,ETAT)),ETATS),
            recordz(etape,(T,X,ETATS)),
            forall(recorded(rapport,(X,E)),writef("%t : %t\n",[X,E])),
        fail
    ;   true
    ).
```

F.5 Outils de trace

```
/*************************************************************************/
/*                                                                       */
/* Execution pas a pas du dialogue                                       */
/*                                                                       */
/*************************************************************************/

:-module(pasapas,
    [
        pas_a_pas/0
    ]
).

pas_a_pas:-
    (   ifdef(trace)
    ->  (   user:coups([COUP|_])
        ->  writef("Coup precedent : %t\n",[COUP])
        ;   writef("Debut du dialogue.\n")
        ),
        repeat,
            writef("Trace : "),
            get_single_char(C),
            (   C\=32, C\=13, C\=10
            ->  exec(C),
                fail
            ;   writef("Suite...\n")
            )
    ;   true
    ),
    !.

exec(0'h):-
    !,
    writef("Aide\n"),
    writef("espace : continuer\n"),
    writef("     h : aide\n"),
    writef("     c : modifier le dernier coup\n"),
    writef("     t : trace prolog\n"),
    writef("     a : abort\n"),
    writef("     x : exit\n").

exec(0'c):-
    !,
    (   retract(user:coups([_|COUPS]))
    ->  writef("Modifier le dernier coup\n"),
        writef("Entrez le nouvel coup : "), read(0,COUP),
        assert(user:coups([COUP|COUPS]))
    ;   writef("Debut du dialogue\n")
    ).

exec(0't):-
    !,
    trace.
```

```
exec(0'x):-
    !,
    abort.

exec(0'a):-
    !,
    abort.

exec(_):-
    writef("Tapez h pour l'aide\n").
```

F.6 Multifenétrage textuel

```
/*****************************************************************************/
/*                                                                           */
/* Fenetres en mode texte                                                    */
/*                                                                           */
/*****************************************************************************/

:-module(windows,
    [
        win_init/2,
        win_open/6,
        win_close/1,
        win_write/2,
        win_write/3,
        win_nl/1,
        win_clear/1,
        win_pos/3,
        draw_windows/0,
        win_link/1
    ]
).

:-use_module(tools).

test:-
    win_init(80,25),
    win_open(my_window,10,10,50,20,"Hello world"),
    for(I,1,10,win_write(my_window,"I=%t\n",[I])),
    win_write(my_window,"Hallo Welt\n"),
    draw_windows,
    win_write(my_window,"Salut monde\n"),
    draw_windows,
    win_clear(my_window),
    win_write(my_window,"That's all folks..........\n..."),
    draw_windows,
    win_close(my_window).

/*  win_descr(W,X1,Y1,X2,Y2,TITLE)
 *  W : handle
```

```
 *  W est le nom du module contenant les caracteres
 */

/*  W:char(X,Y,CH)
 */

win_init(MAXX,MAXY):-
    retractall(maxXY(_,_)),
    asserta(maxXY(MAXX,MAXY)),
    retractall(real_char(_,_,_)),
    forall(win_descr(W,_,_,_,_,_),retractall(W:char(_,_,_))),
    retractall(win_descr(_,_,_,_,_,_)),
    forall(recorded(win_pos,_,K),erase(K)),
    retractall(link(_,_)),
    !.

:-dynamic
    win_descr/6,
    real_char/3.

:-index(win_descr(1,0,0,0,0,0)).
:-index(real_char(1,1,0)).

win_open(W,X1,Y1,X2,Y2,TITLE):-
    (   var(W)
    ->  gen_sym(win_,W)
    ;   true
    ),
    (   win_descr(W,_,_,_,_,_)
    ->  error("Fenetre deja definie : %t\n",[W])
    ;   true
    ),
    (   is_list(TITLE)
    ->  THE_TITLE=TITLE
    ;   name(TITLE,THE_TITLE)
    ),
    assertz(win_descr(W,X1,Y1,X2,Y2,THE_TITLE)),
    dynamic(W:char/3),
    index(W:char(1,1,0)),
    win_clear(W),
    !.

win_close(W):-
    (   clause(win_descr(W,_,_,_,_,_),_,REF)
    ->  win_clear(W),
        erase(REF),
        W:retractall(char(_,_,_))
    ;   error("Fenetre non ouverte : %t\n",[W])
    ),
    retractall(link(W,_)),
    retractall(link(_,W)),
    !.

win_clear(W):-
```

```
    W:retractall(char(_,_,_)),
    (   recorded(win_pos,W=(_,_),REF)
    ->  erase(REF)
    ;   true
    ),
    recordz(win_pos,W=(1,1)),
    !.

win_pos(W,X,Y):-
    recorded(win_pos,W=(X,Y)),
    !.

win_put(W,CH):-
    recorded(win_pos,W=(X,Y),REF),
    erase(REF),
    W:retractall(char(X,Y,_)),
    (   (   CH=10
        ;   CH=13
        )
    ->  X1=1,
        (   win_descr(W,_,Ym,_,YM,_),
            Y is YM-Ym-1
        ->  win_scroll(W,fail),
            Y1=Y
        ;   Y1 is Y+1
        )
    ;   X1 is X+1,
        Y1=Y,
        W:assertz(char(X,Y,CH))
    ),
    recordz(win_pos,W=(X1,Y1)),
    !.

win_scroll(W,LINK):-
    findall(char(X,Y,CH),W:char(X,Y,CH),CHARS),
    retractall(W:char(_,_,_)),
    forall(member(char(X,Y,CH),CHARS),
        (   (   Y>1
            ->  Y1 is Y-1,
                W:assertz(char(X,Y1,CH))
            ;   true
            )
        )
    ),
    (   LINK
    ->  recorded(win_pos,W=(X,Y),REF),
        erase(REF),
        (   Y>1
        ->  Y1 is Y-1
        ;   Y1 is 1
        ),
        recordz(win_pos,W=(X,Y1))
    ;   forall(
            link(W,WL),
```

```
            win_scroll(WL,true)
        )
    ),
    !.

win_write(W,FORMAT):-
    win_write(W,FORMAT,[]),
    !.

win_write(W,FORMAT,ARGS):-
    swritef(ST,FORMAT,ARGS),
    string_to_list(ST,LST),
    win_put_all(W,LST),
    !.

win_nl(W):-
    win_write(W,"\n"),
    !.

win_put_all(_,[]):-!.
win_put_all(W,[CH|ST]):-
    win_put(W,CH),
    win_put_all(W,ST),
    !.

:-dynamic used/2.

:-index(used(1,1)).

draw_windows:-
    retractall(real_char(_,_,_)),
    forall(win_descr(W,X1,Y1,X2,Y2,TITLE),
        (   retractall(win_adjust(_,_,_,_)),
            asserta(
                (   win_adjust(XA,YA,XB,YB):-
                        XB is XA+X1,
                        X1 < XB, XB < X2,
                        YB is YA+Y1+1,
                        Y1+1 < YB, YB < Y2
                )
            ),
            length(TITLE,L),
            XT is (X1+X2-L)>>1,
            YT is Y1+1,
            draw_title(XT,YT,TITLE),
            forall(W:char(X,Y,CH),draw_char(X,Y,CH)),
            box(X1,Y1,X2,Y2)
        )
    ),
    maxXY(MAXX,MAXY),
    nl,
    (   feature(unix,true)
    ->  tty_clear,
        retractall(used(_,_)),
```

```
        forall(
            (   real_char(X,Y,CH),
                \+ used(X,Y)
            ),
            (   tty_goto(X,Y),
                format('~c',[CH]),
                asserta(used(X,Y))
            )
        ),
        retractall(used(_,_)),
        tty_goto(1,MAXY),
        nl
    ;   for(Y,1,MAXY,
            (   for(X,1,MAXX,
                    (   (   real_char(X,Y,CH)
                        ->  format('~c',[CH])
                        ;   format(' ')
                        )
                    )
                ),
                nl
            )
        )
    ),
    retractall(real_char(_,_,_)),
    !.

draw_title(_,_,[]):-!.
draw_title(X,Y,[CH|ST]):-
    assertz(real_char(X,Y,CH)),
    X1 is X+1,
    draw_title(X1,Y,ST),
    !.

draw_char(X,Y,CH):-
    (   win_adjust(X,Y,XD,YD)
    ->  assertz(real_char(XD,YD,CH))
    ;   true
    ),
    !.

box(X1,Y1,X2,Y2):-
    YT is Y1+1,
    for(X,X1,X2,
        (   asserta(real_char(X,Y1,0'-)),
            asserta(real_char(X,Y2,0'-)),
            assertz(real_char(X,YT,0'_))
        )
    ),
    for(Y,Y1+1,Y2-1,
        (   asserta(real_char(X1,Y,0'|)),
            asserta(real_char(X2,Y,0'|))
        )
    ),
```

```
    !.

:-dynamic link/2.

win_link([]):-!.
win_link([_]):-!.
win_link([WIN|LINKED_WIN_LIST]):-
    (   member(LINKED_WIN,LINKED_WIN_LIST),
            assertz(link(WIN,LINKED_WIN)),
            assertz(link(LINKED_WIN,WIN)),
            fail
    ;   win_link(LINKED_WIN_LIST)
    ).
win_link(_).
```

F.7 Programme principal

```
/****************************************************************************/
/*                                                                          */
/* Interface avec le simulateur                                             */
/*                                                                          */
/****************************************************************************/

:-use_module([define,decls,tools,dialogue,compil]).

:-undefine(afficher_rapport).
:-redefine(afficher_nouveaux).
:-undefine(debug_window).
:-redefine(trace).

:-dynamic fichier/1.

state:-
    user:agents(X,_),
    writef("\n********* %t\n",[X]),
    clause(X:state(X,_,_,_,_,_,E),flag(_,_,_)),
    numbervars(E,0,_),
    writef("%t\n",[E]),
    fail
    ;
    true.

main:-
    retractall(fichier(_)),
    sim.

main(FICHIER):-
    retractall(fichier(_)),
    assertz(fichier(FICHIER)),
    sim.

/****************************************************************************/
/*                                                                          */
```

```
/* Menu principal                                                            */
/*                                                                           */
/*****************************************************************************/

sim:-
    reinit,
    repeat,
        once(fichier(FICHIER);FICHIER=''),
        menu("Simulation de dialogue : Menu principal",
            [
                "r" : "Remise a zero"                 :   reinit,
                "f" : ("Fichier (%t)",[FICHIER])    :   fichier,
                "e" : "Editer"                        :   editer,
                "c" : "Compiler"                      :   compiler,
                "s" : "Simuler"                       :   simuler,
                "o" : "Options"                       :   options,
                "q" : "Quitter"                       :   (QUITTER=oui)
            ],
            "==> "
        ),
    QUITTER==oui.

/*****************************************************************************/
/*                                                                           */
/* Choix du fichier de description                                           */
/*                                                                           */
/*****************************************************************************/

fichier:-
    writef("Nom du fichier principal de la description du dialogue : "),
    readln(FICHIER),
    !,
    (   FICHIER=''
    ;   retractall(fichier(_)),
        asserta(fichier(FICHIER))
    ).


/*****************************************************************************/
/*                                                                           */
/* Edition du fichier                                                        */
/*                                                                           */
/*****************************************************************************/

editer:-
    (   fichier(FICHIER)
    ->  vi(FICHIER)
    ;   writef("Vous devez d'abord definir un fichier !\n")
    ).

vi(FICHIER):-
    concat('vi ',FICHIER,VI),
    shell(VI,STATUS),
    (   STATUS==0
```

```
    ;   writef("Impossible d'executer la commande %t\n",[VI])
    ),
    !.


/***************************************************************************/
/*                                                                         */
/* Compilation                                                             */
/*                                                                         */
/***************************************************************************/

compiler:-
    (   fichier(FICHIER)
    ->  analyser(FICHIER)
    ;   writef("Vous devez d'abord definir un fichier !\n")
    ).

/***************************************************************************/
/*                                                                         */
/* Simulation                                                              */
/*                                                                         */
/***************************************************************************/

simuler:-
    (   agents(BILL,JOHN)
    ->  writef("Dialogue entre %t et %t\n\n",[BILL,JOHN]),
        dialoguer
    ;   writef("Vous devez specifier les agents du dialogue !\n")
    ).


/***************************************************************************/
/*                                                                         */
/* Menu des options                                                        */
/*                                                                         */
/***************************************************************************/

options:-
    repeat,
        once((ifdef(afficher_nouveaux),NOUV="oui";NOUV="non")),
        once((ifdef(afficher_rapport),ETAPE="oui";ETAPE="non")),
        once((ifdef(trace),TRACE="oui";TRACE="non")),
        once((ifdef(debug_window),DEBUG="oui";DEBUG="non")),
        menu("Simulation de dialogue : Menu des options",
            [
                "n" : ("Afficher les nouveaux etats : %s",[NOUV])
                    :   switch(afficher_nouveaux),
                "e" : ("Afficher chaque etape       : %s",[ETAPE])
                    :   switch(afficher_rapport),
                "t" : ("Trace                       : %s",[TRACE])
                    :   switch(trace),
                "d" : ("Fenetre Debug               : %s",[DEBUG])
                    :   switch(debug_window),
                "c" : "Continuer"
```

```
                    :   (QUITTER=oui)
            ],
            "==> "
        ),
    QUITTER==oui.

/******************************************************************************/
/*                                                                            */
/* (Des)Activation d'un define                                                */
/*                                                                            */
/******************************************************************************/

switch(DEF):-
    (   ifdef(DEF)
    ->  undefine(DEF)
    ;   define(DEF)
    ).
```

Bibliographie

Asher, N. (1993). *Reference to Abstract Objects in Discourse.* Kluwer Academic Publishers.

Asher, N. and Lascaride, A. (1995). Questions in dialogue. *Linguistic and philosophy*, 21.

Awada, A. (1993). *Vers un copilote informatique automatique dialoguant : Etude de la communication intentionnelle et coopérative dans un système de compréhension de phrases écrites.* PhD thesis, ENSEEIHT.

Beun, R.-J. (1991). A framework for cooperative dialogue. In *Proccedings of the Venaco Workshop on multimodal dialogue.*

Beun, R.-J. (1996). Speech act generation in cooperative dialogue. In *Proceedings of the Twente Workshop on language technology.*

Bunt, H. (1994). Context and dialogue control. Technical report, Institute for Language Technology and Artificial Intelligence, Tilburg, The Netherlands.

Bunt, H. (1996). Dynamic interpretation and dialogue theory. In *The structure of multimodal dialogue.* Taylor, M. M., Bouwhuis, D. G. and Neels F., editors, John Benjamin, Amsterdam.

Burton, M. and Brna (1996). Clarissa : an exploration of collaboration through agent-based dialogue games. In *Proceedings of the EuroAIED*, Lisbon.

Carletta, J. T. and Mellish, C. (1995). Requirements for belief models in cooperative dialogue. Technical Report Technical Report HCRC/RP-66, Human communication research center, University of Edinburgh.

Cayrol and Prade (1998). *Cours de DEA : Raisonnement Plausible et Incertitude.* IRIT.

Chellas, B. F. (1980). *Modal Logic.* Cambridge University Press, New York.

Evrard, F. (1998a). *Cours : Traitement et Gestion Electronique de Documents Textuels.* ENSEEIHT.

Evrard, F. (1998b). *Cours de DEA : Linguistique et Logique Computationnelle.* ENSEEIHT.

Gleizes and Fade (1998). *Cours de DEA : Intelligence Artificielle Distribuée et Planification.* IRIT.

Grice, H.-P. (1975). *Logic and conversation*, volume 3, pages 41–58. P. Cole (ed.), new york : academic press edition.

Levin, J. and Moore, J. (1980). Dialogue-games : meta-communication structure for natural language interaction. *Cognitive science*, 1(4) :395–420.

MacKenzie, J. (1979). Question-begging in non-cumulative systems. *Journal of philosophical logic*, 8 :117–133.

Mann, W. C. (1988). Dialogue games : conventions of human interaction. *Argumentation*, 2(4) :511–532.

Maudet, N. and Evrard, F. (1998). A generic framework for dialogue game implementation. In *Proceedings of the second workshop on Formal Semantics and Pragmatics of Dialogue*, University of Twente, The Netherlands.

Ricordel, D. (1996). *Extension d'un modèle informatique de traitement de la langue basé sur la grammaire de Montague.* PhD thesis, ENSEEIHT.

Searle, J. R. (1969). *Les actes de langage. Essai de philosophie du langage.* Hermann, éditeurs des sciences et des arts.

Tauli, V. (1968). *Introduction to a theory of Language Planning.* Uppsala : Almqvist and Wiksell.

Traunmuller, H. (1991). Conversational maxims and principles of language planning. *working papers PERILUS XII*, pages 25–47.

Wittgenstein, L. (1957). Philosophical invezstigations. Technical report, Oxford : Blackwell.