MASTER EN SCIENCES INFORMATIQUES

La compréhension du langage naturel
de la théorie à l'implémentation

Bouchez, Olivier; Istace, Olivier

Award date:
1993

Awarding institution:
Universite de Namur

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

La compréhension du langage naturel: de la théorie à l'implémentation

Olivier Bouchez
&
Olivier Istace

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maître
en Informatique

Promoteur: Monsieur JM Jacquet

Année académique: 1992-1993

Abstract

This master thesis tackles three classes of problems in understanding natural language.

- **Syntaxic parsing** aims at decomposing the structure of a sentence. Our approach namely is based on formal grammars, categorial grammars. The main principles are:
  - each word is associated with one or more categories;
  - the categories are defined in terms of elementary categories;
  - the reduction rules are based on the definition of the categories and allow the deduction of the grammatical class of a group of words.
- **Semantical representation** uses the dynamic predicate logic with error state to formally represent the meaning of a sentence. Our approach is based on the λ-calculus. Each couple of words and of categories is associated with a λ-expression. The semantic representation is obtained by reducting λ-expressions.
- **The interpretation** specifies the automate understanding of natural language. We have developed a simple interpretation which evaluates the meaning of a sentence according to a model. This evaluation consists in determining whether the interpretation is true, false or results in an error. It is based on the derivation of preconditions in first ordinary logic and in the evaluation of the precondition as a request to a logical program.

Our work embraces both the theoretical and practical approaches. For each class, we first present the theoretical aspects and then propose an implementation. Moreover, to illustrate the generality of our work, fragments of French and English are treated all along the thesis.

Key words

Natural language, formal grammar, categorial grammar, semantical representation, dynamic predicate logic, presupposition, logical programming, Prolog, Gödel, λ-calculus.

Résumé

Ce mémoire aborde trois classes de problèmes liés à la compréhension automatique du langage naturel.

- **L'analyse syntaxique** a comme but de décomposer la structure d'une phrase. Notre approche est basée se sur des grammaires formelles: les grammaires catégorielles. Les principes de base sont les suivants:
  - chaque mot est associé à une ou plusieurs catégories;
  - les catégories sont définies en termes de catégories élémentaires;
  - les règles de réduction sont basées sur les définitions des catégories et permettent de déduire la classe grammaticale d'un groupe de mots.
- **La représentation sémantique** utilise la logique des prédicats dynamiques avec état d'erreur pour présenter une approche formelle de la signification d'une phrase. Le principe de réduction repose sur le λ-calcul. Chaque couple formé d'un mot et d'une catégorie est associé à une λ-expression. La représentation sémantique est obtenue par réduction des λ-expressions.
- **L'interprétation** détermine la compréhension automatique du langage naturel. Nous avons développé une interprétation élémentaire qui permet d'évaluer le sens de la phrase par rapport à un modèle logique. Cette évaluation consiste à indiquer si une phrase est vraie, fausse ou erronée. Elle repose sur la dérivation de préconditions exprimées en logique du premier ordre et sur l'évaluation de ces préconditions comme une requête d'un programme logique.

Notre travail se situe à la fois au niveau théorique et pratique. Pour chaque classe de problèmes, nous présentons d'abord les aspects théoriques et nous proposons ensuite une implémentation. En outre, afin de suggérer la généralité, des fragments de français et d'anglais sont traités tout au long du mémoire.

Mots clés

Langage naturel, grammaire formelle, grammaire catégorielle, représentation sémantique, logique des prédicats dynamiques, présuppositions, programmation logique, Prolog, Gödel, λ-calcul.

Remerciements

Nous tenons à remercier toutes les personnes qui nous ont encouragés et soutenus durant la réalisation de ce mémoire. Nous avons une pensée particulière pour le personnel du CWI, les organisateurs de la sixth conference of the EACL chapter et la bourse Comett.

Nous remercions personnellement les Professeurs Jean-Marie Jacquet (FUNDP) et Jan van Eijck (CWI) qui nous ont suivis tout au long de ce mémoire.

Table des matières

Préface

Depuis l'invention de l'informatique, le mode de communication entre l'être humain et l'ordinateur n'a cessé d'évoluer. A une époque paraissant si lointaine et pourtant datant d'au plus 20 ans, les systèmes d'alors fonctionnaient en mode batch. Ils recevaient de l'opérateur un certain nombre de données et effectuaient un traitement sans aucune interaction. Aujourd'hui, par contre, les systèmes fonctionnent en mode interactif. L'ordinateur et l'être humain dialoguent régulièrement. Les interfaces ne cessent d'évoluer pour devenir de plus en plus conviviaux. Pour communiquer avec l'ordinateur, l'être humain dispose en général de boîtes de dialogue. Ces boîtes contiennent une série de questions précises telles que le nom, le prénom, le numéro de compte et le montant à verser. L'utilisateur répond par un mot ou une série de mots que l'ordinateur interprète facilement. Ce genre d'échange est adapté pour les applications qui posent toujours le même type de questions et reçoivent le même type de réponses. Cependant, un certain nombre d'applications, comme l'interrogation d'une base de données, peuvent recevoir une quantité importante de requêtes différentes. L'utilisateur dispose alors d'un langage d'interrogation plus ou moins évolué (comme SQL). Il doit donc faire l'effort d'étudier ce langage. Dans cette optique, l'être humain doit s'adapter à l'ordinateur. Dans une optique encore plus proche de l'être humain, le langage humain tend à être directement soumis à l'ordinateur. Cela permet à tout utilisateur d'interroger l'ordinateur dans sa propre langue.

Il existe de nombreuses approches et théories qui offrent des solutions plus ou moins satisfaisantes à ce problème. Nous en évoquerons quelques-unes dans le chapitre 1. Dans le cadre de ce mémoire, nous allons présenter une approche dans la lignée des travaux en linguistique formelle de Montague [Chambreuil 1989]. Elle consiste à traduire un texte en langage naturel dans un langage formel que l'ordinateur peut ensuite interpréter. Cette méthode peut être comparée à un compilateur qui traduit un programme écrit en langage de programmation évolué en un code objet exécutable par un ordinateur. Ce traitement s'effectue en deux étapes.

Dans un premier temps, il est nécessaire d'analyser la syntaxe du texte et de décomposer la structure de chaque phrase. Il existe de nombreuses techniques pour effectuer ces deux opérations. Les plus connues sont les grammaires non-contextuelles (cfr section 1.1.2) et les réseaux de transition (cfr section 1.1.1). Notre choix s'est porté vers un type de grammaires moins courantes: les grammaires catégorielles. En effet, il nous a semblé plus intéressant d'étudier un système pour lequel il existe peu d'implémentations. Dans le cadre de ce mémoire, nous avons également développé une grammaire catégorielle pour un fragment du français. Le chapitre 3

décrit l'analyse automatique d'une phrase au moyen d'un programme.

La seconde étape consiste à traduire la phrase en langage naturel dans un langage formel. Dans le chapitre 4, nous présentons la logique des prédicats dynamiques (DPL) introduite dans [van Eijck 1991] qui permet de représenter la sémantique d'une phrase. Un programme DPL manipule des états (un ensemble de variables et de valeurs). Il contient des opérations de test sur l'état et des opérations d'assignation définie et indéfinie de nouvelles variables. Ainsi, la phrase "Jean voit un homme" est représentée par un algorithme correpondant à:

- soit y une variable qui reçoit comme valeur "Jean";
- soit x une variable;
- un test qui vérifie que x est un homme
- un test qui vérifie que y voit x

Dans le chapitre 5, nous verrons comment élaborer un programme d'ordinateur qui effectue automatiquement la traduction d'une phrase en langage naturel en logique des prédicats dynamiques. Le principe consiste à associer une λ-expression à chaque mot en fonction de sa classe grammaticale. Le programme DPL est obtenu en réduisant l'application de ces λ-expressions.

A partir de cette représentation DPL, l'ordinateur peut effectuer une interprétation. Le chapitre 6 montre le processus d'interprétation au travers de l'évaluation d'une phrase par rapport à une base de données logique. Le principe consiste à traduire un programme DPL en une expression logique du premier ordre.

Cette implémentation et la théorie qui l'accompagne ont fait l'objet d'un article et d'une présentation à la *"Sixth Conference of the European Chapter of the Association for Computational Linguistics"* [EACL 1993].

La lecture de ce mémoire suppose un certain nombre de connaissances de base en programmation logique. Ces bases sont rappelées dans le chapitre 2. Le langage de programmation utilisé est le langage Gödel, développé par Lloyd et Hill à l'Université de Bristol [Hill & Lloyd 1992]. Il nous a semblé intéressant de réaliser notre implémentation dans ce nouveau langage pour pouvoir l'évaluer.

L'organisation du travail repose sur deux parties complémentaires. D'une part, la traduction d'une phrase du langage naturel en un programme DPL et d'autre part l'interprétation et l'exploitation de cette représentation sémantique. La tâche de rédaction a été répartie de la manière suivante:

- Olivier Bouchez s'est occupé de la rédaction des chapitres 2, 3, 4, 5
- Olivier Istace a rédigé les chapitres 1 et 6

Après la rédaction de chaque chapitre, il y a eu un échange et une correction mutuelle.

Le travail d'implémentation a été réalisé, dans un premier temps, en collaboration étroite. D'une part, Gödel nous était complètement inconnu et notre pratique de la programmation logique n'était pas développée. D'autre part, la matière et les théories que nous avons abordées étaient relativement complexes. Après avoir développé une première version de l'analyse syntaxique et de la traduction des phrases en DPL, nous avons séparé la tâche en deux parties.

- Olivier Bouchez a amélioré l'analyse syntaxique en y introduisant la gestion des accords et a développé une approche plus adéquate de la traduction.
- Olivier Istace a approfondi la partie méta-programmation de Gödel et a développé le module d'évaluation.

Chapitre 1

Aperçu de l'analyse syntaxique et sémantique du langage naturel

Depuis de nombreuses années, les linguistes et les chercheurs en intelligence artificielle ont travaillé à la formalisation du langage naturel. Le linguiste tente de représenter le plus fidèlement possible le langage naturel au travers de théories formelles. En concevant des programmes basés sur les théories des linguistes, le chercheur en intelligence artificielle vérifie leur niveau d'exactitude.

Le langage naturel peut être analysé à deux niveaux qui, selon les écoles, sont considérés comme indépendants. Le langage peut tout d'abord être analysé sur le plan syntaxique c'est-à-dire que l'on vérifie l'exactitude grammaticale d'une phrase. Le second plan concerne la sémantique de la phrase ; on s'attache alors à essayer de donner une signification à la phrase.

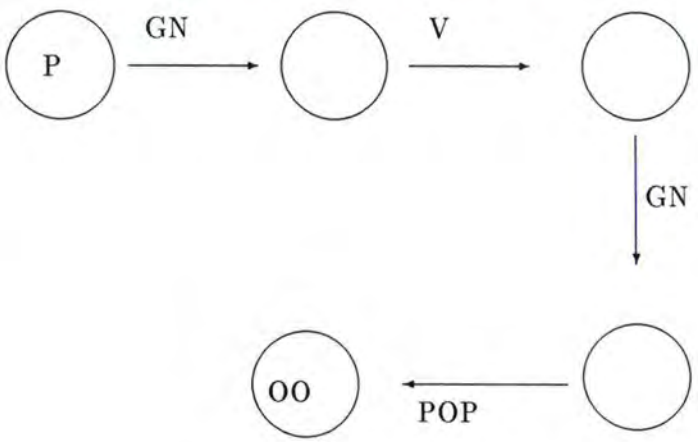
1.1 L'analyse syntaxique du langage naturel

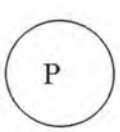
Les recherches en analyse syntaxique sont fort avancées. Il existe actuellement une littérature abondante sur les grammaires formelles et leurs développements informatiques. Nous en voulons pour preuve la prolifération des logiciels de correction orthographique et grammaticale.

Nous présenterons trois méthodes principales d'analyse grammaticale, tout d'abord les réseaux de transition; ensuite les grammaires de type non-contextuel et enfin les grammaires catégorielles. L'objectif de cette section est de montrer de manière informelle les principes de base de chacune de ces méthodes, nous renvoyons le lecteur à [King 1983] pour une approche plus approfondie.

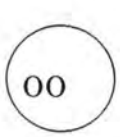
1.1.1 Les réseaux de transition

Cette théorie a été développée par Woods en 1970 [King 1983]. Elle vise à représenter un type de phrase sous forme d'un graphe. Ainsi, le graphe d'une phrase élémentaire comprenant un sujet, suivi d'un verbe et d'un complément est le suivant :





représente l'état initial.



représente l'état final.

Figure 1.1: Réseau de transition simplifié d'une phrase

Afin de vérifier l'exactitude grammaticale d'une phrase, il faut que cette phrase puisse parcourir le graphe, donc il est nécessaire que le sujet (GN) précède le verbe (V) et que ce dernier précède le complément (GN). Il s'agit d'un exemple simplifié. Il faut en effet prendre en compte la forme du verbe (active ou passive) et le type de la phrase (interrogatif ou impératif) ainsi que le traitement des accords, des auxiliaires, des adjectifs, des éventuelles relatives... Les graphes se complexifient donc très rapidement (Cfr. Figure 1.2).

Le premier graphe décrit la structure d'un groupe nominatif. Il est composé d'un déterminant et d'un nom commun. A ce nom commun peut être attaché, par l'intermédiaire d'un pronom relatif une phrase relative. Le second graphe représente la structure d'un groupe verbal. C'est-à-dire un verbe éventuellemnt accompagné d'un auxiliaire et/ou d'un groupe nominal complément.

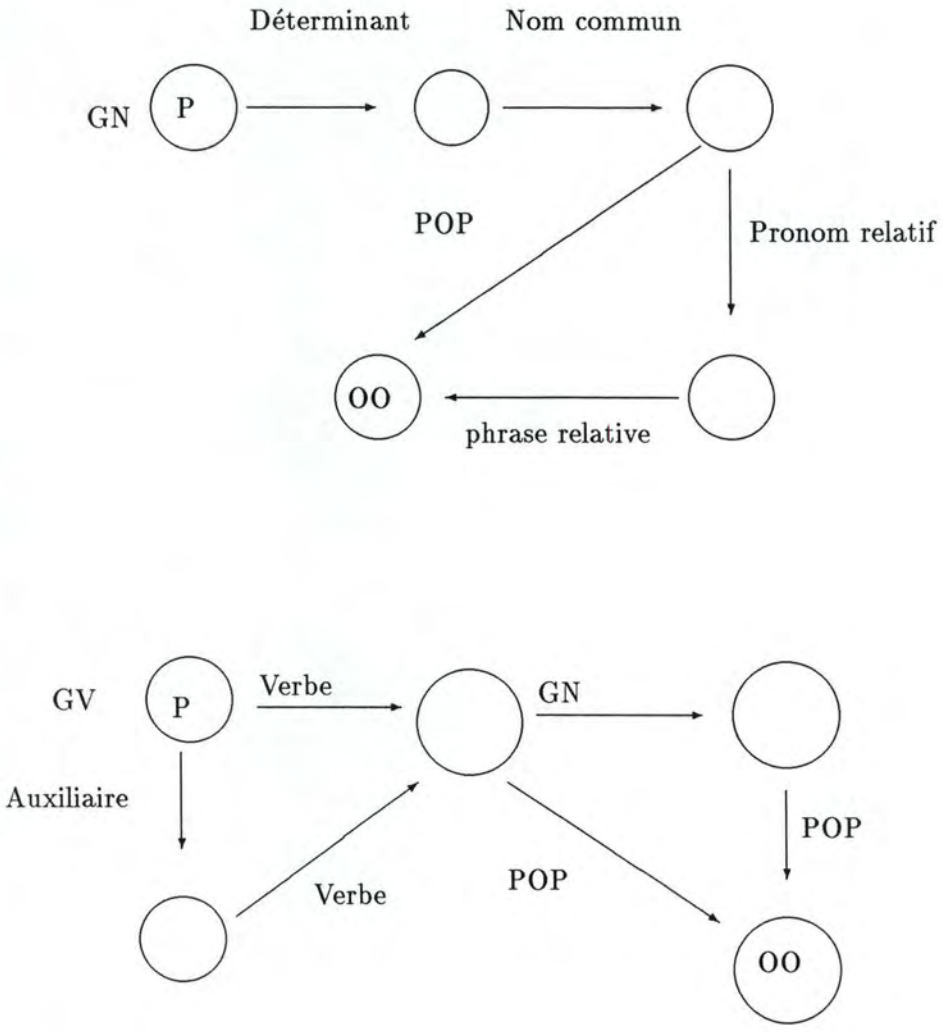


Figure 1.2: Réseau de transition d'une phrase

1.1.2 Les grammaires non-contextuelles

Les langages (naturels ou de programmation) ont souvent une structure récursive. Les grammaires contextuelles permettent de définir aisément ce type de structure.

Nous illustrons le principe de fonctionnement de ce type de grammaire au travers d'un exemple.

La phrase à analyser est la suivante : "Jean voit un homme".

Afin de vérifier son exactitude grammaticale, quatre classes d'objets doivent être identifiées :

i. un ensemble fini, P, de règles de productions.
ii. un ensemble fini, VT, de symboles terminaux.

iii. un ensemble fini, VN, de symboles non terminaux.
iv. le symbole de départ S.

Le symbole terminal est le mot. Le symbole non-terminal représente le résultat d'une ou plusieurs compositions de symboles non-terminaux ou terminaux.

Ces concepts appliqués à notre exemple, nous obtenons le résultat suivant:

VT = {Jean, voit, un, homme}
VN = {N, S, NP, VP, DET, V}

P = {S → NP VP
NP → DET N
VP → V NP
VP → V
N → homme
DET → un
NP → Jean
V → voit}

Avec ces règles, il est possible de vérifier que notre phrase est bien de type S. La figure 1.3 représente l'analyse obtenue en appliquant les règles de production de P.

1.1.3 Les grammaires catégorielles

Le principe est d'attribuer à chaque mot une catégorie. Les catégories sont définies en terme de catégorie de base (P,E) et en terme de composition de catégories au moyen de fonctions ($\backslash$ et /). Sur base des définitions de catégories, des règles de réductions sont définies telles que:

1. $a\backslash b$ suivi de b donne a.
2. a/b précédé de a donne b.

L'analyse d'une phrase est obtenue en réduisant la liste des catégories de chaque mot.

Par exemple, les catégories suivantes:

- DET = GN $\backslash$ N

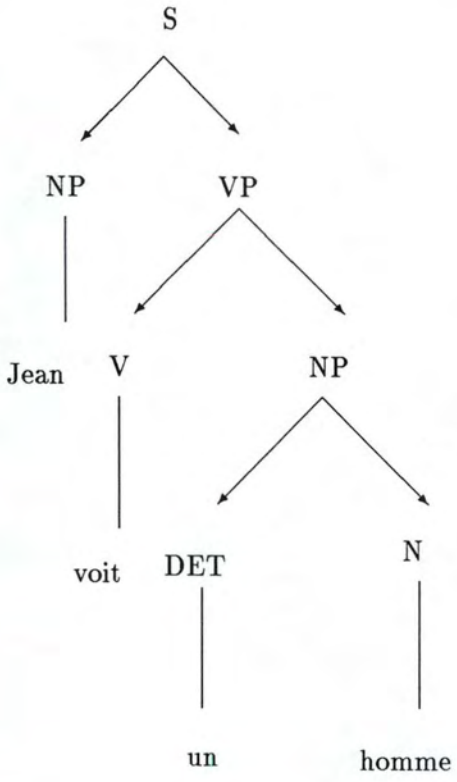


Figure 1.3: Arbre d'analyse d'une phrase par une grammaire non-contextuelle

- GN = P \ (E / P)
- VP = (E / P)
- VT = VP \ GN
- N = (S \ E)

La réduction de ces catégories donnent l'arbre d'analyse illustré par la figure 1.4.

Nous ne nous attarderons pas plus amplement sur ce sujet, pour le moment, car c'est cette méthode que nous avons utilisée pour notre implémentation (Cfr. Chapitre 3).

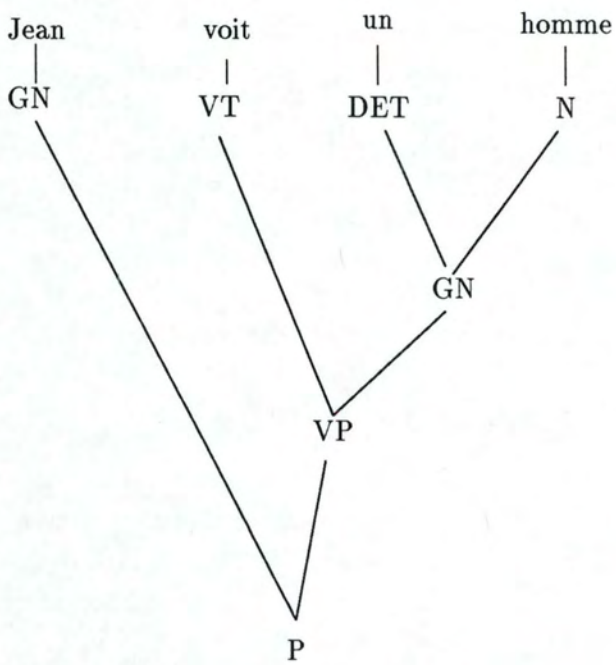


Figure 1.4: Arbre d'analyse d'une phrase au moyen d'une grammaire catégorielle

1.2 Le traitement sémantique

Il importe de préciser que, par rapport au plan syntaxique, les recherches sont nettement moins avancées. Peu d'applications pratiques ont été développées et les systèmes opérationnels tels que l'interrogation d'une base de données en langage naturel résultent le plus souvent d'une série d'heuristiques plutôt que d'une théorie formellement définie [Dahl et Saint-Dizier 1984] et [Grosz, Sparck et Webber 1986].

Notre choix s'est porté sur les programmes les plus connus. Sont ainsi successivement exposés ELIZA, STUDENT, PARRY, GUS, Baseball.

1.2.1 ELIZA

Eliza [Grosz, Sparck et Webber 1986] marque le début de la période des premières applications en intelligence artificielle. Il fut développé par Weizenbaum en 1954. C'est le premier système qui donne l'impression de converser avec son utilisateur. Dans ce cas, il simulait un premier entretien psychiatrique.

Le principe du système consiste à repérer une série de mots clés et mémorise une série d'informations qui lui permettent de poser des questions par la suite.

Ce système est assez rudimentaire : pas de référence contextuelle, pas d'univers du discours. De plus, il est très facile de le piéger. Weizenbaum l'évalue d'ailleurs comme suit : "ELIZA peut être comparé à une actrice maîtrisant un ensemble de techniques mais ne disant rien de son cru."

1.2.2 STUDENT

Il fut mis au point un peu plus tard (en 1967) par Bobrow. La finalité de "student" était de résoudre des problèmes d'algèbre. L'énoncé du problème est introduit en langage naturel et le système fournit la réponse.

Comme pour Eliza, la compréhension est strictement déterminée par la nature du problème : la psychiatrie pour Eliza et l'algèbre pour Student [Grosz, Sparck et Webber 1986].

1.2.3 PARRY

Parry fut réalisé par une équipe de psychiatres et d'informaticiens
[Grosz, Sparck et Webber 1986].

Les psychiatres tentaient de définir la paranoïa. Cette définition déterminait le comportement

de l'ordinateur au cours d'une conversation entre un psychiatre et un patient. Ce système a été testé sur un certain nombre de psychiatres, afin de voir si le psychiatre diagnostiquait bien la paranoïa. L'objectif de ce système était donc d'évaluer la description de la paranoïa. En effet, si la définition est exacte, Parry devait se comporter comme un paranoïaque typique et le psychiatre pouvait diagnostiquer facilement la maladie et son niveau de gravité.

1.2.4 BASEBALL

Il s'agit d'un répondeur automatique développé par B.F. Green, Q.K. Wolf, C. Chomsky et K. Laughery [Grosz, Sparck et Webber 1986]. L'utilisateur posait des questions concernant les matches de baseball et le système donnait lui-même les réponses.

"Where did the Red Sox play on July 7?"

Baseball construit à partir de cette question une liste de spécification du type :

Place = ?
Team = Red Sox
Month = July
Day = 7.

La réponse n'est pas exprimée en anglais, mais sous forme d'une la liste de spécifications remplie.

Comme le contexte est déterminé, cela a permis une représentation aisée des connaissances et une non ambiguïté des termes employés par l'utilisateur.

1.2.5 GUS (Genial Understander system)

Ce système fut développé par le centre de recherche de Xerox à Palo Alto au cours de années 1975-1976 [Grosz, Sparck et Webber 1986].

Il offrait la possibilité de dialoguer en anglais avec un utilisateur dans un but très précis et ceci dans un domaine très restreint du discours. Si le système joue le rôle d'un agent de voyage, le client (c'est-à-dire l'utilisateur) aura qu'une seule possibilité : demander un ticket aller-retour pour une et une seule ville de Californie de son choix. Le contexte est donc très limité.

Ici aussi, les problèmes sont contournés en limitant de manière drastique le contexte d'utilisation du système.

1.3 Conclusion

Dans ce chapitre, nous avons illustré trois méthodes d'analyse syntaxique et quelques implémentations de systèmes d'analyse sémantique. Nous avons vu que ces sytèmes sont relativement limités. Cependant, ils montrent qu'une approche du langage naturel par l'ordinateur est possible dans un contexte donné.

Chapitre 2

Introduction à la logique

2.1 Aperçu de la programmation en logique

Les idées de la programmation logique sont apparues à la fin des années soixantes suite aux recherches en démonstration automatique de théorèmes. Le principe de Robinson consistait à démontrer automatiquement qu'une proposition est une conséquence logique d'un ensemble de propositions supposées vraies par hypothèse. Ce principe a donné jour à l'idée que la logique peut être utilisée comme base d'un langage de programmation. En effet, un programme logique est composé d'un ensemble d'axiomes et son exécution consiste à prouver un but.

Ce type de langage a bouleversé la manière de voir et de construire un programme. Il n'est plus nécessaire de décrire l'algorithme de résolution en terme d'un enchaînement d'actions. Pour résoudre un problème, il suffit de le définir en termes logiques.

Exemple Notre exemple au cours de ce chapitre est basé sur la situation suivante: Arthur et Hector sont les fils de Jean et Antoine est le fils de Lucien. Pour des raisons de simplicité, supposons que deux personnes différentes sont frères si elles sont issues du même père. Notre problème consiste à vérifier si deux personnes sont frères. En Pascal, ou tout autre langage impératif, nous écrivons d'abord l'entête d'une procédure et nous décrivons ensuite une stratégie de résolution du problème. Une méthode consiste, par exemple, à utiliser un tableau de records pour représenter les relations père-fils. Pour vérifier si deux personnes sont frères, il suffit alors de rechercher dans le tableau le père de chaque personne et de vérifier si les deux pères sont identiques.

En programmation logique, l'approche est totalement différente. Il est question non plus de décrire un procédé de résolution d'un problème, mais d'exprimer ou de spécifier ce problème en termes logiques. Ainsi, la première étape consiste à généraliser le problème en introduisant des variables quantifiées universellement: Pour tout X, Y, A, la personne X est le frère de Y si et seulement si A est le père de X, A est le père de Y et X est différent de Y. La seconde étape

consiste à utiliser un formalisme logique en introduisant des prédicats.

$$Frere(x, y) \text{ ssi } Pere(x, a) \text{ et } Pere(y, a) \text{ et } x \neq y.$$

La résolution de $Frere(x, y)$ se base sur la résolution du problème $Pere(x, a)$. La résolution de ce problème consiste à énoncer les faits Jean est le père d'Arthur, Jean est le père d'Hector, Lucien est le père d'Antoine et à les traduire dans le formalisme logique:

$$Pere(Arthur, Jean),$$

$$Pere(Hector, Jean),$$

$$Pere(Antoine, Lucien).$$

Le programme consiste dès lors en un ensemble de règles où toutes les variables sont supposées quantifiées universellement:

$Pere(Antoine, Lucien)$
$Pere(Hector, Jean)$
$Pere(Arthur, Jean)$
$Frere(x, y) \leftarrow Pere(x, a) \wedge Pere(y, a) \wedge x \neq y$

L'exécution du programme pour vérifier si Arthur est le frère d'Hector consiste à prouver que

$$Frere(Arthur, Hector)$$

est déductible du programme. La démarche est la suivante:

1. Si x est remplacé par $Arthur$ et si y est remplacé par $Hector$, la dernière règle du programme devient

   $$Frere(Arthur, Hector) \leftarrow Pere(Arthur, a) \wedge Pere(Hector, a) \wedge Arthur \neq Hector.$$

2. Si nous parvenons à démontrer

   $$Pere(Arthur, a) \wedge Pere(Hector, a) \wedge Arthur \neq Hector$$

   alors

   $$Frere(Arthur, Hector)$$

   sera également démontré.

3. Si nous remplaçons a par $Jean$,

   $$Pere(Arthur, Jean)$$

   correspond à une règle du programme qui est vraie par hypothèse.

4. Si nous prenons $a = Jean$ alors

$$Pere(Arthur, a)$$

est *vrai* aussi. Il reste à prouver

$$Pere(Hector, Jean) \wedge Arthur \neq Hector.$$

Vérifions si

$$Pere(Hector, Jean)$$

est également *vrai*.

5. En outre,

$$Pere(Hector, Jean)$$

correspond à une règle du programme. Donc

$$Pere(Hector, Jean)$$

est également *vrai*.

6. Il ne reste plus qu'à vérifier

$$Arthur \neq Hector.$$

L'inégalité est prédéfinie et correspond au sens usuel.

7. Nous avons donc démontré que

$$Pere(Arthur, a) \wedge Pere(Hector, a) \wedge Arthur \neq Hector$$

est *vrai* si $a = Jean$. Donc

$$Frere(Arthur, Hector)$$

est *vrai*.

Supposons maintenant que le problème qui nous intéresse est de trouver le frère d'une personne. En Pascal, il est nécessaire d'écrire une nouvelle procédure qui recherche les fils du père de cette personne autre que cette personne. En programmation logique, par contre, la seule chose à modifier est la requête à effectuer au programme. En effet, notre problème consiste maintenant à trouver les valeurs de f telles que $Frere(Arthur, f)$ est *vrai*. La démarche est similaire à celle de $Frere(Arthur, Hector)$:

1. Nous allons remplacer le x de la dernière règle du programme par $Arthur$ et y par le f de la requête. La dernière règle du programme devient:

$$Frere(Arthur, f) \leftarrow Pere(Arthur, a) \wedge Pere(f, a) \wedge Arthur \neq f.$$

2. Or

$$Pere(Arthur, a)$$

est *vrai* si $a = Jean$ (cfr exemple précédent)

3. Il reste à prouver

$$Pere(f, Jean) \wedge Arthur \neq f$$

4. Dans

$$Pere(f, Jean),$$

si nous remplaçons f par $Arthur$, nous obtenons

$$Pere(Arthur, Jean)$$

qui correspond à un fait du programme. D'où

$$Pere(f, Jean)$$

est *vrai* si $f = Arthur$.

5. Il reste à prouver que

$$Arthur \neq Arthur,$$

ce qui est *faux*.

6. Revenons en arrière et regardons s'il n'y a pas une autre solution pour

$$Pere(f, Jean).$$

Si nous remplaçons cette fois f par $Hector$,

$$Pere(Hector, Jean)$$

correspond également à un fait.

7. Il reste à vérifier

$$Arthur \neq Hector$$

inégalité qui est *vraie*.

8. De plus

$$Pere(Arthur, a) \wedge Pere(f, a) \wedge Arthur \neq f$$

est *vrai* si $a = Jean$ et si $f = Hector$. Donc

$$Frere(Arthur, f)$$

est *vrai* si $f = Hector$.

Ainsi que suggérés par les exemples précédents, la programmation logique se révèle en pratique bien adéquate pour passer rapidement de la spécification formelle au programme. Cette caractéristique en fait un outil intéressant pour le développement de prototypes.

2.2 Logique du premier ordre

La logique du premier ordre sous-tend la programmation logique. Elle offre un outil de raisonnement sur les programmes en définissant les concepts syntaxiques de bases et en donnant une sémantique des programmes logiques. Les sous-sections suivantes sont essentiellement théoriques par leur lot de définitions. Elles constituent cependant les bases essentielles pour la construction de programmes logiques. Elles sont tirées de [Lloyd 1987].

2.2.1 Syntaxe

Dans la section 2.1, nous avons ébauché la construction d'un programme logique sans définir une syntaxe précise. Dans cette sous-section, nous allons combler cette lacune en apportant un fondement théorique à la syntaxe des programmes logiques. La syntaxe de la logique du premier ordre est construite sur quatre ensembles disjoints deux à deux.

- un ensemble de variables;
- un ensemble de constantes;
- un ensemble de fonctions;
- un ensemble de prédicats.

L'élément de base d'un programme logique est le terme.

Définition Les **termes** sont définis par les règles suivantes:

- une variable est un terme;
- une constante est un terme;
- si f est une fonction n-aire et $t_1 \ldots t_n$ sont des termes, alors $f(t_1 \ldots t_n)$ est un terme.

Exemple *Hector, Jean, Arthur* sont des constantes, donc des termes.
a, x, y sont des variables donc des termes.
$f(a)$, $g(x, y)$ sont des termes.

La syntaxe de la logique des prédicats du premier ordre est définie au moyen de formules bien formées.

Définition Les **formules bien formées** sont définies par les règles suivantes:

- si p est un prédicat n-aire et $t_1 \ldots t_n$ sont des termes alors $p(t_1 \ldots t_n)$ est une formule bien formée (appelée **atome**);
- Si F et G sont des formules bien formées alors $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ sont aussi des formules bien formées.
- Si F est une formule bien formée et x est une variable, alors $(\forall x F)$ et $(\exists x F)$ sont des formules bien formées.

Cette notion est très importante. Pour preuve, elle revient souvent dans les prochaines définitions. Nous supposerons pour la suite que toutes les formules sont bien formées. De même pour la simplicité de l'écriture, les parenthèses inutiles seront omises et $F \leftarrow G$ sera parfois utilisé à la place de $G \rightarrow F$.

Exemple $Pere(Arthur, Jean)$ est un atome
$\forall x, \forall y, \forall a\ (((Pere(x,a) \wedge Pere(y,a) \wedge x \neq y) \rightarrow Frere(x,y))$ est une formule.

La notion de portée permet de distinguer deux types d'occurence de variables. Une occurrence de variable peut être soit libre, soit liée.

Définition La **portée** de $\forall x$ $(\exists x)$ dans $\forall x F$ $(\exists x F)$ est F. Une **occurrence liée** d'une variable dans une formule est:

- soit une occurrence qui suit directement un quantificateur;
- soit une occurrence dans la portée du quantificateur de la variable qui suit directement ce quantificateur.

Toutes les autres occurrences de variables sont **libres**.

Exemple Dans la formule

$$\forall x, \forall y, \forall a(((Pere(x,a) \wedge Pere(y,a) \wedge x \neq y) \rightarrow Frere(x,y)),$$

toutes les occurrences de x, y, a sont liées. Par contre, dans $(\forall x, a\ Pere(x,a)) \wedge Pere(y,a)$, les occurrences de x sont liées, l'occurrence de y est libre, les deux premières occurrences de a sont liées; la troisième est libre.

La notion d'occurrence libre permet de définir la notion de formule close.

Définition Une **formule close** est une formule qui ne contient aucune occurrence de variable libre.

Définition Un **litéral** est soit un atome, soit la négation d'un atome.

La clause est l'élément essentiel d'un programme logique. Si nous comparons un programme logique à un texte, une clause est une phrase et un litéral est un mot.

Définition Une **clause** est une formule de la forme

$$\forall x_1, \ldots \forall x_s (L_1 \vee \ldots \vee L_m)$$

où chaque L_i est un litéral et $x_1, \ldots x_s$ sont des variables présentes dans $L_1, \ldots, L_m$.

Exemple $\forall x$, $\forall y$, $\forall a$ $(((\neg Pere(x,a) \vee \neg Pere(y,a) \vee x = y) \vee Frere(x,y))$ est une clause.

Convention Tirant profit de l'équivalence entre les formules $A \rightarrow B$ et $\neg A \vee B$, nous réécrivons la formule

$$\forall x_1, \ldots \forall x_s (A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n)$$

sous la forme plus simple

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n.$$

Exemple $\forall x, y, a (Frere(x,y) \wedge \neg Pere(x,a) \wedge \neg Pere(y,a) \wedge x = y)$
est réécrit

$$Frere(x,y) \leftarrow Pere(x,a), Pere(y,a), x \neq y.$$

Une clause de Horn est un cas particulier de clauses

Définition Une **clause de Horn** est une clause qui a au plus un atome non nié. C'est une clause de la forme $A \leftarrow B_1, \ldots, B_n$ qui a au plus un seul membre de gauche et où A est un atome non nié et où $B_1, \ldots, B_n$ sont des atomes niés.

Nous pouvons donc aborder la définition des éléments constitutifs des programmes.

Définition Une **règle de programme définie** est une clause de la forme $A \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ qui ne contient qu'un et un seul élément dans le conséquent. A est appelé l'entête de la clause et $B_1, \ldots, B_n$ le corps de la clause. Cependant, nous allons permettre pour plus de facilité que les B_i soient niés ou non niés.

Un fait est un cas particulier de règle de programme.

Définition Un **fait** est une règle particulière de la forme $A \leftarrow$ qui possède un corps vide.

Un programme est défini de la manière suivante:

Définition Un **programme défini** est un ensemble fini de règles.

Exemple $Pere(Arthur, Jean)$
$Pere(Hector, Jean)$

$Pere(Antoine, Emile)$
$Frere(x, y) \leftarrow Pere(x, a), Pere(y, a), x \neq y.$
est un programme d efini.

Dans la section 2.1, nous avons vu que l'exécution d'un programme est guidée par la démonstration d'un but.

Définition Un **but défini** est une clause de la forme $\leftarrow B_1, \ldots, B_n$ $(n \geq 1)$. C'est une clause qui a un conséquent vide. Chaque B_i est appelé un sous-but.

Exemple $\leftarrow Frere(Arthur, Hector)$ est un but.

2.2.2 Théorie des modèles

La sémantique d'un programme logique peut être décrite par la théorie des modèles définissant la valeur de vérité des expressions logiques. Elle exprime de manière formelle le sens donné couramment à une expression logique. Ainsi

$$\forall x, \forall y, \forall a, ((Pere(x, a) \wedge Pere(y, a) \wedge x \neq y) \rightarrow (Frere(x, y)))$$

a la signification intuitive suivante : Quelles que soient les variables x, y, a, si le père de x et de y est a et si x est différent de y alors x est le frère de y.

La notion de base dans la théorie des modèles est l'interprétation. Celle-ci effectue un lien entre les expressions d'un langage du premier ordre et un ensemble non-vide.

Définition Une **interprétation** I d'un langage du premier ordre L consiste en:

- un sous-ensemble non-vide D, appelé le domaine de l'interprétation;
- pour chaque constante dans L, une assignation d'un élément de D;
- pour chaque fonction n-aire dans L, une assignation d'une fonction de D^n dans D;
- pour chaque prédicat n-aire dans L, une assignation d'une fonction de D^n dans $\{vrai, faux\}$

Exemple Considérons les familles suivantes composées de Hector, Arthur, Jean, Antoine et Lucien avec Jean père de Hector et Arthur et Lucien père de Antoine. Supposons que le langage du premier ordre utilisé pour décrire cette famille contient les constantes $Hector$, $Arthur$, $Jean$, $Antoine$ et $Lucien$ ainsi que le prédicat $Pere$.

Une inteprétation est obtenue comme suit:

- choisissons pour domaine D l'ensemble $\{e_1, e_2, e_3, e_4, e_5\}$;

- assignons à *Hector* l'élément e_1, à *Arthur* l'élément e_2, à *Jean* l'élément e_3, à *Antoine* l'élément e_4, à *Lucien* l'élément e_5;
- assignons au prédicat *Pere* la fonction *fpere* définie comme suit:
  $fpere(x, y) = \begin{cases} vrai \text{ si } (x, y) \in \{(e_1, e_3), (e_2, e_3), (e_4, e_5)\} \\ faux \text{ sinon} \end{cases}$

Il y a donc une correspondance entre les constantes d'une expression et les éléments d'un ensemble non-vide, entre les fonctions d'une expression et les fonctions dans cet ensemble et entre les prédicats d'une expression et les fonctions de cet ensemble vers $\{vrai, faux\}$. Les variables d'une expression n'ont pour le moment aucune correspondance dans cet ensemble. Chaque variable est donc assignée à un élément de l'ensemble au moyen d'une assignation de variables.

Définition Soit I une interprétation de domaine D d'un langage du premier ordre. Une **assignation de variables** est une assignation de chaque variable dans L à un élément dans le domaine I.

Exemple Supposons que les variables x, y appartiennent au langage du premier ordre. Une assignation de variables est par exemple:
$x = e_1$
$y = e_3$

A partir de la notion d'interprétation et d'assignation de variables, il est possible de définir une assignation des termes. En effet, pour rappel, un terme est soit une constante, soit une variable, soit une fonction de termes.

Définition Soit I une interpétation de domaine D d'un langage du premier ordre et soit V une assignation de variables.

Une **assignation de termes** est définie par:

- à chaque variable est assignée un élément de D selon V;
- à chaque constante est assignée un élément de D selon I;
- si $t'_1, \ldots, t'_n$ sont les assignations des termes $t_1, \ldots, t_n$ et si f' la fonction de D^n dans D assignée à f, alors $f'(t'_1, \ldots, t'_n)$ est l'assignation du terme $f(t_1, \ldots, t_n)$.

Exemple Soit le terme $Pere(x, Jean)$. Sur base des exemples précédents, ce terme est assigné de la manière suivante:
x est assigné à e_1 selon l'assignation des variables.
Jean est assigné à e_3 selon l'assignation des constantes.
$Pere(x, Jean)$ est assigné à $fpere(e_1, e_3)$.

Définition Soit I une interprétation de domaine D d'un langage du premier ordre et soit V une assignation de variables. A toute formule dans L, nous pouvons lui donner une **valeur de vérité:**

- Si la formule est un atome $p(t_1, \ldots, t_n)$, sa valeur de vérité est obtenue en calculant $p'(t'_1, \ldots, t'_n)$ où p' est la fonction de D^n dans D assignée à p et $t'_1, \ldots, t'_n$ sont les assignations des termes $t_1, \ldots, t_n$ selon I et V.
- Si la formule est de la forme $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$ alors sa valeur de vérité est donnée par la table suivante:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
vrai	*vrai*	*faux*	*vrai*	*vrai*	*vrai*	*vrai*
vrai	*faux*	*faux*	*faux*	*vrai*	*faux*	*faux*
faux	*vrai*	*vrai*	*faux*	*vrai*	*vrai*	*faux*
faux	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*

- Si la formule est de la forme $\exists x F$, alors la valeur de vérité de la formule est *vrai* s'il existe $d \in D$ tel que F a la valeur de vérité *vrai* selon I et $V(x/d)$ où $V(x/d)$ est V sauf x qui est assigné à la valeur d; sinon sa valeur de vérité est *faux*.
- Si la formule est de la forme $\forall x F$, alors la valeur de vérité de la formule est *vrai* si pour tout $d \in D$ tel que F a la valeur de vérité *vrai* selon I et $V(x/d)$; sinon sa valeur de vérité est *faux*.

Exemple L'interprétation de $Pere(x, Jean)$ est donnée par $fpere(e_1, e_3)$ qui est vrai. De même, l'interprétation de $Pere(Hector, Jean) \wedge Pere(Arthur, Jean)$ est donnée par $fpere(e_1, e_3) \wedge fpere(e_2, e_3)$ ou encore $vrai \wedge vrai$ qui par la table donne *vrai*

Remarque Nous constatons que la valeur de vérité d'une formule close ne dépend pas de l'assignation de variables. Pour rappel, une formule close est une formule qui ne contient que des variables liées. Nous pourrons donc parler sans ambiguïté de la valeur de vérité d'une formule close par rapport à une interprétation.

Définition Soit I une interprétation d'un langage du premier ordre L et soit F une formule close de L; alors I est un **modèle** pour F ssi F est *vrai* pour I.

Définition Soit S un ensemble de formules closes, nous dirons alors que I est un **modèle** pour S si et seulement si I est un modèle pour chaque formule de S.

Etant donné qu'un programme est un ensemble de formules closes (cfr section 2.2.1) Nous pourrons parler de modèle pour un programme. De même, vu qu'un but est une formule close, nous pourrons aussi parler d'un modèle pour un but.

Définition Soit S un ensemble de formules closes et F une formule close d'un langage du premier ordre L. La formule F est une **conséquence logique** de S si pour chaque interprétation

I de L, I est un modèle pour S implique I est un modèle pour F. Cette situation est notée par $S \models F$.

La notion de conséquence logique correspond à la notion de déduction du but d'un programme. Un but est une conséquence logique d'un programme si un modèle pour le programme est un modèle pour le but.

Exemple Soit S un ensemble de formules closes correspondant à notre programme
$\{Pere(Arthur, Jean)$,
$Pere(Hector, Jean)$,
$Pere(Antoine, Emile)$,
$\forall x, \forall y, \forall a\ (((Pere(x,a) \wedge Pere(y,a) \wedge x \neq y) \rightarrow Frere(x,y))\ \}$
et F une formule close correspondant au but $Frere(Arthur, Hector)$.

Montrons que F est une conséquence logique de S. Soit I un modèle pour S, montrons que I est également un modèle pour F.

Vu que I est un modèle pour S, la valeur de vérité de

$$\forall x, \forall y, \forall a(((Pere(x,a) \wedge Pere(y,a) \wedge x \neq y) \rightarrow Frere(x,y))$$

est vrai c'est-à-dire pour toute assignation de x, y et a. En particulier si l'on prend $x = Arthur$, $y = Hector$ et $a = Jean$,

$$Pere(Arthur, Jean) \wedge Pere(Hector, Jean) \wedge Arthur \neq Hector) \rightarrow Frere(Arthur, Hector)$$

est vrai. Or en supposant $Arthur \neq Hector$, le prémisse est vrai car

$$Pere(Arthur, Jean$$

est vrai et

$$Pere(Hector, Jean)$$

est vrai; donc Frere(Arthur,Hector) est vrai. D'où I est un modèle pour S.

Pour être complet et démontrer que $Arthur \neq Hector$, nous devrions ajouter les axiomes suivants qui établissent la théorie de l'égalité; ce que nous supposerons acquis implicitement.

1. $c \neq d$ pour toutes paires c, d de constantes distinctes;
2. $\forall x_1, \ldots, \forall x_n, \forall y_1, \ldots, \forall y_m$
   $(f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$, pour toutes paires f, g de symboles de fonctions différentes;
3. $\forall x_1, \ldots, \forall x_n$
   $(f(x_1, \ldots, x_n) \neq c)$ pour chaque constante c et chaque symbole de fonction f;
4. $\forall x$
   $(t[x] \neq x)$ pour chaque terme $t[x]$ contenant x et différent de x;

5. $\forall x_1, \ldots, \forall x_n, \forall y_1, \ldots, \forall y_n$
   $((x_1 \neq y_1 \vee \ldots \vee x_n \neq y_n) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n))$, pour chaque symbole de fonction f.

6. $\forall x$
   $(x = x)$

7. $\forall x_1, \ldots, \forall x_n, \forall y_1, \ldots, \forall y_n$
   $((x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, pour chaque symbole de fonction f

8. $\forall x_1, \ldots, \forall x_n, \forall y_1, \ldots, \forall y_n$
   $((x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \leftarrow p(y_1, \ldots, y_n)))$, pour chaque symbole de prédicat p

2.2.3 Théorie de la preuve

Jusqu'à présent, nous avons défini un programme logique d'un point de vue syntaxique et déclaratif. Nous allons dans cette sous-section nous intéresser à l'exécution d'un programme logique. Celle-ci est basée sur les notions de substitution, d'unification, de résolution SLD.

Avant d'en venir à une série de définitions, voyons d'abord intuitivement comment un programme logique s'exécute.

Soit le programme:

$$\boxed{\begin{array}{l} Pere(Arthur, Jean) \\ Pere(Hector, Jean) \\ Pere(Antoine, Emile) \\ Frere(x, y) \leftarrow Pere(x, a), Pere(y, a), x \neq y. \end{array}}$$

et le but:

$$\leftarrow Frere(Arthur, Hector)$$

La démarche suivante permet de déduire un but d'un programme.

- La première étape consiste à choisir une clause telle que son entête et le but correspondent. Le mécanisme d'unification permet de définir cette correspondance. La clause

  $$Frere(x, y) \leftarrow Pere(x, a), Pere(y, a), x \neq y$$

  correspond au but

  $$Frere(Arthur, Hector).$$

- Si x a comme valeur $Arthur$ et si y a comme valeur $Hector$ alors la clause devient:

$$Frere(Arthur, Hector) \leftarrow Pere(Arthur, a), Pere(Hector, a), Arthur \neq Hector.$$

Ces mécanismes correspondent aux notions de substitution et d'instantiation.

- L'étape suivante consiste à démontrer le but

$$Pere(Arthur, a), Pere(Hector, a), Arthur \neq Hector.$$

Démontrons donc le premier sous-but. Comme à la première étape, recherchons une clause du programme qui correspond à ce but. La clause

$$Pere(Arthur, Jean)$$

est sélectionnée. Cette clause n'a pas de second membre. Elle est donc vraie. D'où

$$Pere(Hector, a)$$

est vrai pour $a = Jean$.

- Le but restant à démontrer est

$$Pere(Hector, Jean), Arthur \neq Hector.$$

Le premier sous-but

$$Pere(Hector, Jean)$$

correspond à la clause

$$Pere(Hector, Jean)$$

qui est vraie et

$$Arthur \neq Jean$$

est également vrai.

- D'où nous déduisons que

$$Pere(Arthur, Jean), Pere(Hector, Jean), Arthur \neq Hector$$

est vrai et donc aussi

$$Frere(Arthur, Hector).$$

Définition Une **substitution** θ est un ensemble fini de termes de la forme

$$\{v_1/t_1, \ldots, v_n/t_n\}$$

où chaque v_i est une variable, chaque t_i est un terme distinct de v_i et où les variables v_1, ..., v_n sont distinctes. Pour tout i, la construction v_i/t_i est appelée une instantiation.

Définition Une **expression** est soit un terme, soit un litéral, soit la conjonction ou la disjonction de litéraux. Une **expression simple** est soit un terme ou soit un atome.

Définition Soient $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ une substitution et E une expression. L'**instance** de E par θ, notée $E\theta$, est l'expression obtenue en remplaçant simultanément chaque occurrence de v_i dans E par t_i. $(1 \leq i \leq n)$

Définition Soient $\theta = \{u_1/s_1\sigma, \ldots, u_m/s_m\sigma\}$ et $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ des substitutions. La **composition** $\theta\sigma$ de θ et σ est obtenue à partir de l'ensemble $\{u_1/s_1, \ldots, u_m/s_m, v_1/t_1, \ldots, v_n/t_n\}$ en supprimant les instantiations pour lesquelles $u_i = s_i\sigma$ ainsi que les instantiations où $v_j \in \{u_1, \ldots, u_n\}$.

Exemple L'ensemble

$$\{x/Hector, y/Jean\}$$

constitue une substitution. Son application à l'expression

$$Pere(x, y)$$

donne

$$Pere(x, y)\{x/Hector, y/Jean\} = Pere(Hector, Jean).$$

De même l'ensemble

$$\{x/y, y/b, z/f\}$$

constitue une substitution et son application à l'expression

$$p(x, y, z(y, z(y, x)))$$

donne

$$p(x, y, z(y, z(y, x)))\{x/y, y/b, z/f\} = p(y, b, f(b, f(b, y)))$$

Définition La substitution donnée par l'ensemble vide correspond à la **substitution identique** et est notée ϵ.

La proposition suivante permet de manipuler facilement les substitutions.

Proposition Soient γ, σ, θ des substitutions

$$\theta\epsilon = \epsilon\theta = \theta$$
$$(E\ \theta)\sigma = E(\theta\sigma)$$
$$(\theta\sigma)\gamma = \theta(\sigma\gamma)$$

Définition Soit $S = \{E_1, \ldots E_n\}$ un ensemble fini d'expressions simples. Une substitution θ est appelée un **unificateur** de S si $E_1\theta = \ldots = E_n\theta$. Un unificateur θ pour S est l'**unificateur le plus général** (most general unifier - **mgu**) pour S si pour chaque unificateur σ pour S, il existe une substitution γ telle que $\sigma = \theta\gamma$.

Exemple $\{Pere(x,a), Pere(Arthur, Jean)\}$ a pour mgu $\{x/Arthur, a/Jean\}$.

Définition Soient $x_1, \ldots x_n$ des variables et $t_1, \ldots t_n$ des termes. Un système d'équation de la forme

$$\begin{cases} x_1 = t_1 \\ \ldots \\ x_n = t_n \end{cases}$$

est sous forme résolue si et seulement si $x_1, \ldots x_n$ sont des variables telles que x_i n'apparait pas t_i $\forall i$ et $x_i \neq x_j \forall i \neq j$

AlgorithmeConsidérons deux expressions simples T et U. L'algorithme suivant permet de trouver le mgu de T et de U:

Soit S un système d'équations initialisé avec $\{T = U\}$.

Tant que possible sélectionner une des équations Eq de la forme suivante et appliquer l'action qui lui est associée:

1. Eq est de la forme [1]

$$f(t_1, \ldots, t_n) = g(u_1, \ldots, u_m)$$

   (a) Si $(f \neq g)$ ou $(n \neq m)$ alors T et U ne s'unifient pas, terminer avec un échec

   (b) sinon remplacer l'équation par les équations [2]

$$t_1 = u_1$$

$$\ldots$$

$$t_n = u_m$$

2. Eq est de la forme

$$X = X$$

   où X est une variable, alors supprimer l'équation du système.

3. Eq est de la forme

$$t = X$$

   où t n'est pas une variable et X est une variable, alors remplacer l'équation par

$$X = t$$

4. Le système contient une équation

$$X = t$$

   où X est une variable qui apparaît dans le système obtenu en enlevant cette équation.

[1] Une constante est considérée comme une fonction sans argument.

[2] Une égalité entre constantes est remplacée par le vide.

(a) Si X apparaît dans t alors T et U ne sont pas unifiables; terminer avec un échec

(b) sinon remplacer X par t dans le système obtenu en enlevant cette équation

Lorsque l'exécution de l'algorithme est terminée: soit l'algorithme a produit un échec, soit s'il produit un système sous forme résolue U et T s'unifie et le mgu est de la forme $\{x_1/t_1, \ldots, x_n/t_n\}$

Définition Soient E et F des expressions. Nous dirons que E et F sont **variants** s'il existe des substitutions θ et σ telles que $E = F\theta$ et $F = E\sigma$.

Exemple Reprenons le programme:

$Pere(Arthur, Jean)$
$Pere(Hector, Jean)$
$Pere(Antoine, Emile)$
$Frere(x, y) \leftarrow Pere(x, a), Pere(y, a), x \neq y.$
et le but: $\leftarrow Frere(Arthur, Hector).$

Montrons comment les concepts de substitution et d'unification interviennent dans la dérivation du but à partir du programme.

- La première étape consiste à trouver une entête de clause du programme qui s'unifie avec le but. La seule clause qui a une chance de s'unifier avec

$$Frere(Arthur, Hector)$$

est

$$Frere(x, y).$$

L'algorithme d'unification donne:

1. $S = \{Frere(Arthur, Hector) = Frere(x, y)$
   Le nom des deux fonctions est identique et le nombre d'arguments également; le système devient donc
2. $S = \left\{ \begin{array}{l} Arthur = x \\ Hector = y \end{array} \right.$
   $Arthur$ est une constante, l'équation est retournée et le système devient donc
3. $S = \left\{ \begin{array}{l} x = Arthur \\ Hector = y \end{array} \right.$
   $Hector$ est une constante, l'équation est retournée et le système devient

4. $S = \left\{ \begin{array}{l} x = Arthur \\ y = Hector \end{array} \right.$

Il n'y a plus de règle applicable. Nous avons donc trouvé le mgu

$$\{x/Arthur, y/Hector\}$$

- L'étape suivante consiste à remplacer le but

$$Frere(Arthur, Hector)$$

par le but correspondant au corps de la clause auquel nous appliquons le mgu trouvé.

$$\leftarrow Pere(Arthur, a), Pere(Hector, a), Arthur \neq Hector.$$

Pour dériver un tel but à partir du programme, nous commençons par dériver le premier sous-but. Nous recherchons une entête de clause qui s'unifie avec

$$Pere(Arthur, a).$$

Trois entêtes de clause du programme peuvent éventuellement s'unifier avec

$$Pere(Arthur, a).$$

Essayons la première clause:

$$Pere(Arthur, a)$$

et

$$Pere(Arthur, Jean).$$

L'algorithme d'unification donne comme mgu $\{Jean/a\}$.

- Comme à l'étape précédente nous remplaçons le but

$$Pere(Arthur, a)$$

par le corps de la clause correspondant auquel nous appliquons le mgu. Dans ce cas, le but résultant est vide ce qui signifie que

$$Pere(Arthur, a)$$

est dérivable avec $\{a/Jean\}$.

- L'expression

$$Pere(Arthur, a)$$

est *vrai* si $\{a/Jean\}$. Nous appliquons cette substitution aux sous-buts restants à démontrer.

$$Pere(Hector, Jean), Arthur \neq Hector$$

- Nous allons maintenant essayer de dériver

$$Pere(Hector, Jean).$$

Comme pour

$$Pere(Arthur, a),$$

il y a trois entêtes de clause potentiellement unifiables. Essayons avec

$$Pere(Arthur, Jean)$$

et voyons ce que donne l'algorithme d'unification.

1. $S = \{Pere(Arthur, Jean) = Pere(Hector, Jean)$
   Les deux fonctions ont le même nom et le même nombre d'arguments. Le système devient
2. $S = \begin{cases} Arthur = Hector \\ Jean = Jean \end{cases}$
   $Arthur$ et $Hector$ sont deux constantes qui non pas le même nom.
   $Pere(Arthur, Jean)$ et $Pere(Hector, Jean)$ ne sont pas unifiables.

Essayons avec une autre entête de clause

$$Pere(Hector, Jean)$$

et le but

$$Pere(Hector, Jean)$$

s'unifient et le mgu est ϵ. Le corps de la clause est vide et donc

$$Pere(Hector, Jean)$$

est dérivable à partir du programme.

- La dernière étape consiste à dériver

$$Hector \neq Arthur$$

qui est *vrai*.

$$\leftarrow Pere(Arthur, a), Pere(Hector, a), Arthur \neq Hector$$

est dérivable à partir du programme si $\{a/Jean\}$. D'où

$$Frere(Arthur, Hector)$$

également.

Cette démarche correspond à la notion formelle de SLD-résolution[3]que nous allons aborder. Nous doterons ainsi les programmes logiques d'une sémantique procédurale.

La méthode consiste à construire une suite de buts terminée par le but vide. A chaque étape, un sous-but quelconque est sélectionné, une clause dont l'entête s'unifie avec ce sous-but est choisie, le sous-but sélectionné est remplacé par le corps de la clause, le mgu est appliqué au buts restants.

Le résultat de l'exécution d'un programme logique est une substitution. Cette réponse est correcte si l'application de cette substitution au but est une conséquence logique du programme.

Définition Soient P un programme défini et G un but défini $\leftarrow A_1, ...A_k$. Une **réponse** pour $P \cup \{G\}$ est une substitution de variables de G. Une **réponse** θ est **correcte** pour $P \cup \{G\}$ si $\forall((A_1 \wedge ... \wedge A_k)\theta)$ est une conséquence logique de P.

Définition Soient G le but $\leftarrow A_1, ...A_k$ et C la clause $A \leftarrow B_1, \ldots B_q$, alors G' est **dérivé** de G et C en utilisant le mgu θ si les conditions suivantes sont vérifiées:

- A_m est un atome, appelé l'atome sélectionné, dans G.
- θ est un mgu de A_m et A.
- G' est le but $(\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, ..., A_k)\theta$.

Définition Soient P un programme défini et G un but défini. Une **SLD-dérivation** de $P \cup \{G\}$ consiste en une séquence (finie ou infinie) de buts G_0, G_1, G_2, ... telle que $G_0 = G$, une séquence C_1, C_2, ... de variants de clause de programme de P et une séquence θ_1, θ_2, ... de mgu tels que chaque G_{i+1} est dérivé à partir de G_i et C_{i+1} en utilisant θ_{i+1}.

Définition Une **SLD-réfutation** de $P \cup \{G\}$ est une SLD-dérivation finie de $P \cup G$ qui possède la clause vide comme dernier but dans la dérivation.

Définition Soient P un programme défini et G un but défini. Une **réponse calculée** θ de $P \cup \{G\}$ est une substitution obtenue en restreignant la composition $\theta_1 \ldots \theta_n$ aux variables de G où $\theta_1, ..., \theta_n$ est la séquence de mgu utilisée dans une SLD-réfutation de $P \cup \{G\}$.

2.3 Prolog

Le langage Prolog a été mis au point au début des années septantes à l'Université de Marseille par Colmerauer. Il constitute une référence en matière de programmation logique. Nous en esquissons les caractéristiques principales dans cette section.

[3]SLD provient de l'anglais, Linear resolution with Selection function for the Definite clauses

2.3.1 Méthode de recherche

La fin de la section 2.2.3 définit le principe de la SLD-resolution pour dériver un but dans un programme. Prolog utilise ce principe de résolution, mais fixe le critère de sélection de sous-buts et le critère de choix de la clause. Prolog sélectionne les buts de la gauche vers la droite et choisit les clauses dans leur ordre d'apparition dans le code du programme (de haut en bas).

La démarche pour l'étape i du programme correspond à:

- Soit G_i le but $A_1 \ldots A_n$ résultant. ($n \geq 1$).
  Prolog recherche la première clause du programme s'unifiant avec A_1 (soit C_1) et crée un point de choix.
  Prolog dérive G_{i+1} à partir de G_i et C_i avec A_1 l'atome sélectionné dans G_i et θ_i l'unificateur de A_1 et C_1.
- S'il n'y a pas de clause s'unifiant avec A_1 , Prolog génère un "échec".
- Si le but résultant est vide alors le but initial est dérivable dans le programme et une réponse peut être calculée en restreignant la composition des substitutions apparues dans la branche de dérivation aux variables présentes dans la requête.
- En cas de "échec", Prolog retourne au point de choix précédent et recommence avec la clause suivante s'unifiant avec A_1.
- S'il n'y a plus de clause s'unifiant avec A_1, Prolog génère un nouvel échec.
- S'il n'y a pas de point de choix précédent, alors le but initial ne peut pas être dérivé dans le programme.

Lorsque l'utilisateur demande une nouvelle solution, Prolog agit comme s'il recevait un "échec". Cette démarche correspond à une recherche en profondeur d'abord de la solution et le mécanisme de retour en arrière s'appelle "backtracking".

Pour démontrer un but du type *not* B, Prolog essaie de prouver B. S'il y parvient alors *not* B n'est pas dérivable. Au contraire, si Prolog trouve que B est faux alors *not* B est vrai. Ce procédé s'appelle la négation par échec. L'inconvénient de cette méthode est qu'il n'y a pas de réponse calculée autre que la substitution identique. En effet, si B échoue, il n'y a pas de réponse calculée. Il n'y en aura donc pas pour *not* B.

Exemple Considérons le programme[4]:

[4] En Prolog, le symbole $\rightarrow$ est représenté par :-, les variables commencent par une majuscule et les constantes, les fonctions et les prédicats commencent par une minuscule

```
pere(antoine,lucien).
pere(hector,jean).
pere(arthur,jean).
frere(X,Y) :- pere(X,A) , pere(Y,A) , not(X = Y).
```

et montrons les appels Prolog pour la démonstration du but:

```
?- frere(hector,X).
```

1. le premier but est `frere(hector,X)` .
   La clause choisie est `frere(X,Y) :- pere(X,A) , pere(Y,A) , not(X = Y)` .
   Le point de choix 1 est créé.
   Le mgu est $\{X/hector, Y/X\}$.
   Le but `pere(hector,A), pere(X,A), not(hector = X)` est dérivé.

2. Le but sélectionné est `pere(hector,A)` .
   La clause choisie est `pere(hector,jean)` .
   Le point de choix 2 est créé.
   Le mgu est $\{A/jean\}$.
   Le but dérivé est `pere(X,jean), not(hector = X)` .

3. Le but sélectionné est `pere(X,jean)` .
   La clause choisie est `pere(hector,jean)` .
   Le point de choix 3 est créé.
   Le mgu est $\{X/hector\}$.
   Le but dérivé est `not(hector = hector)` .

4. `not(hector=hector)` ne peut être dérivé et provoque un retour au dernier point de choix c'est-à-dire au point de choix 3.

5. La clause choisie est `pere(arthur,jean)` .
   Aucun nouveau point de choix n'est créé.
   Le mgu est $\{X/arthur\}$.
   Le but dérivé est `not(hector = arthur)` .

6. `not(hector = arthur)` est dérivable dans le système. Le but résultant est vide.
   `frere(hector,X)` est dérivable et la solution calculée est $\{X/arthur\}$.

7. Si l'utilisateur demande une nouvelle solution, Prolog recommence au dernier point de choix c'est-à-dire au point de choix 3.

8. Il n'y a pas d'autre clause unifiable avec `pere(X,jean)`
   Prolog retourne au point de choix 2.

9. Il n'y a pas d'autre clause unifiable avec `pere(Hector,a)`
   Prolog retourne au point de choix 1.
   Il n'y a pas d'autre clause unifiable avec `frere(hector,X)`

10. Il n'y a plus de point de choix précédent.
    Il n'y a pas d'autre solution. Prolog génère `no` .

Il est évident que l'ordre des clauses et des sous-buts d'une clause détermine l'exécution du programme. En effet, Prolog ne sélectionnera pas dans le même ordre les clauses du programme, ni ne dérivera pas les buts dans le même ordre. Cette méthode de dérivation a donc un effet direct sur la manière d'écrire un programme Prolog. Dans certains cas extrêmes, un programme Prolog qui se termine peut ne plus se terminer si les clauses sont dans un autre ordre.

2.3.2 Mécanismes auxiliaires

La sémantique déclarative d'un programme Prolog est donnée par la théorie des modèles (cfr. section 2.2.2). Par contre la sémantique procédurale est liée à l'exécution du programme. Elle est définie par le procédé de recherche d'un but dans un programme. Un programme Prolog a donc deux sémantiques. Ces deux sémantiques doivent en théorie correspondre. Or ce n'est pas en pratique le cas.

La stratégie de recherche ne permet pas toujours de trouver une solution, alors qu'en théorie il existe une telle solution. En effet, à cause de la recherche en profondeur d'abord (pour des raisons d'efficacité), Prolog peut entrer dans une branche de recherche infinie. De plus, pour améliorer la puissance d'expression et l'efficacité des programmes, Prolog dispose de mécanismes auxiliaires sans contrepartie déclarative.

L'état des variables Prolog dispose de quelques prédicats qui lui permettent de tester l'état des variables. Ces prédicats n'ont aucun fondement logique, mais permettent d'améliorer le pouvoir d'expression d'un programme Prolog.

Le prédicat `var(X)` teste si à un instant donné un terme est une variable non instantiée. Par exemple `var(X)` est vrai. Par contre, `var(hector)` est faux; de même que `X=hector, var(X)` est faux. Il n'est pas possible de définir ce prédicat en terme logique. En effet, le fait `var(X)` signifie que toutes les instances de `X` sont des variables et non que `X` représente une variable.

Les prédicats `nonvar(X)`, `integer(X)`, `ground(X)` sont de la même catégorie. Ils permettent de définir des conditions d'utilisation d'une clause. En effet, certaines clauses ne peuvent être employées que si `X` est une variable ou un entier, ...

Le cut Le prédicat `cut` , noté!, est un prédicat qui affecte le comportement procédural d'un programme Prolog. Sa sémantique déclarative est identique au programme sans le `cut`. Le `cut` permet d'améliorer l'efficacité d'un programme Prolog, en diminuant l'espace de recherche par suppression de backtracking inutile.

Le `cut` produit les effets suivants sur le moteur de recherche. Supposons que le but `P` à démontrer s'unifie avec une clause qui contient un `cut`. Si ce `cut` est franchi, Prolog n'utilisera pas les clauses se trouvant sous cette clause pour produire une autre solution et Prolog ne produira pas de nouvelles solutions pour les buts à gauche du `cut` . Par contre le `cut` n'affecte pas les sous-buts à sa droite. Si ceux-ci produisent un échec, Prolog remonte au dernier point de choix précédant le choix de la clause avec un `cut` .

Le `cut` agit donc en fonction de sa position dans le programme. Son emploi est d'ailleurs plutôt controversé. Il existe deux utilisations du `cut` . La première n'a aucun effet sur la sémantique déclarative d'un programme et améliore uniquement la recherche de la solution. Par exemple:

```
p :- cond, !, exp1
p :- not(cond), !, exp2
```

Si la condition est vérifiée, il est inutile de chercher une solution dans la seconde clause.

Le programme suivant est équivalent au premier, d'un point de vue procédural. Par contre la sémantique déclarative n'a plus de sens vu qu'elle n'impose aucun critère de choix, ni d'ordre des clauses.

```
p :- cond, !, exp1
p :- exp2
```

La méta-programmation La méta-programmation est un concept très intéressant qu'offre la programmation logique. Le principe général consiste à permettre à un programme de manipuler un programme. En Prolog, un programme peut en cours d'exécution effectuer un certain nombre d'opérations sur lui-même. Il peut d'une part accéder directement aux clauses de son programme par le prédicat `clause(Head,Body)` qui donne le corps de la clause qui s'unife avec `Head` . D'autre part, un programme Prolog peut ajouter ou retirer des clauses au programme. Le prédicat `assert` ajoute une clause à la fin du programme, le prédicat `asserta` ajoute une clause au début du programme, par contre `retract` retire une clause du programme. La sémantique déclarative d'un programme est donc modifiée en cours d'exécution de celui-ci.

2.4 Gödel

Gödel est un langage logique, en cours de développement à l'Université de Bristol par Hill et Lloyd [Hill & Lloyd 1992]. Les auteurs veulent une puissance d'expression aussi complète que Prolog tout en ajoutant des mécanismes plus déclaratifs.

2.4.1 Les Mécanismes principaux

Contrairement à Prolog, Gödel est un langage typé et modulaire.

Les types Gödel En Prolog, les variables, les constantes et les termes étaient non typés. Il n'y avait aucune necessité de déclarer les termes et leur type, ni non plus de déclarer les prédicats et les fonctions en indiquant le type des arguments et du résultat. En Gödel, par contre, il est obligatoire de déclarer les types manipulés par le programme et d'indiquer le type de constantes, des fonctions et de ses arguments, ainsi que le type des arguments des prédicats. Par contre, il n'y a aucune déclaration des variables. Le type d'une variable est déduit de leur utilisation. Par exemple, soit `Pere` un prédicat avec deux arguments de type `Personne` .

La déclaration de `Pere` est la suivante:

```
Pere : Personne * Personne
```

De `Pere(a,b)` , il est possible de déduire que `a` et `b` sont des variables de type `Personne` .

De même, `Fact` est une fonction qui reçoit comme argument un entier et produit comme résultat un entier.

Sa déclaration est la suivante:

```
Fact : Integer -> Integer
```

De `a = Fact(b),` on peut déduire que `a` et `b` sont des variables de type `Integer` .

Un mécanisme intéressant en Gödel, pour alléger les contraintes de types est le polymorphisme. Les constructeurs permettent de construire un nouveau type, par exemple, soit `Jour` un type de base et `Liste` un constructeur alors `Jour, Liste(Jour), Liste(Liste(Jour)),` ... sont des types. Le polymorphisme permet d'avoir des déclarations de types qui contiennent un paramètre.

Par exemple `CONSTANT Nil: Liste(a)` permet de définir une constante de type `Liste(a)` . Ceci permet donc d'éviter de déclarer une constante `NilJour` de type `Liste(Jour),` `NilInteger` de type `Liste(Integer),` ....

De même la définition du prédicat `Append Liste(a) * Liste(a) * Liste(a)` permet de définir la concaténation d'une liste quelconque et d'éviter de redéfinir le prédicat `append` pour chaque type de liste.

Les modules Gödel Un programme Gödel est composé d'un ensemble de modules. Un module est constitué de deux parties: "export" et "local". La partie "export" contient la déclaration des

types, des constantes, des fonctions, des prédicats qui peuvent être utilisés en dehors du module. Par contre, la partie "local" contient les déclarations internes au module et la définition des prédicats.

2.4.2 Les mécanismes auxiliaires

Tout comme Prolog, la stratégie de recherche se fait en profondeur d'abord. Les auteurs de Gödel ont voulu améliorer l'aspect déclaratif du langage tout en gardant le même pouvoir d'expression.

L'état des variables En Prolog, nous avons vu qu'il existe des prédicats spéciaux qui permettent de tester l'état d'une variable. L'objectif de ces prédicats est de réserver l'utilisation d'une clause à certain type de termes. En Gödel, il n'existe plus de tel prédicat. Ceux-ci sont remplacés par des contraintes exprimées sous forme de `DELAY` . L'action d'un `DELAY` permet de retarder l'appel de cette clause jusqu'à ce que les variables soient dans un certain état: soit `NONVAR` ou soit `GROUND` . La différence par rapport à Prolog est donc que les prédicats qui ne pouvaient être exprimés logiquement ont été remplacés par des contraintes et que l'utilisation d'une clause n'est plus "empêchée" mais retardée.

Le cut Gödel supporte deux types de `CUT` . Le premier prend la forme de `if then else` . Son utilisation est `if cond then exp1 else exp2` et sa signification correspond à `(cond` $\wedge$ `exp1)` $\vee$ `(`$\neg$ `cond` $\wedge$ `exp2)` . Cette sémantique est donc correcte par rapport à la sémantique déclarative.

Une autre forme de `CUT` est le `COMMIT` . Le "commit" à la forme d'une étiquette qui porte sur une condition. Il est représenté par la condition entre accolades avec une étiquette en indice de la dernière accolade $(\{\texttt{cond}\}_n)$. Si la condition `cond` est vérifiée alors:

- il n'y aura qu'une et une seule solution pour `cond` , ce qui correspond partiellement en Prolog à ne pas rechercher de nouvelles solutions pour les sous-buts à gauche du `cut` ;
- toutes les clauses qui contiennent un commit avec cette étiquette sont élaguées, ce qui correspond plus ou moins en Prolog, à élaguer les clauses en dessous du `cut` .

Par rapport au `cut` Prolog, le `commit` Gödel est plus riche. Il ne tient pas compte de l'ordre des clauses. L'étiquette permet d'écarter certaines clauses et d'en conserver d'autres. La portée du `commit` permet un backtracking à gauche et à droite de sa position. Seule la condition entre les { et } ne donne qu'une et une seule solution. Le `commit` n'est pas un prédicat.

La méta-programmation La méta-programmation en Gödel est l'aspect qui nous semble le plus intéressant et que nous avons exprérimenté en détail. La méta-programmation en Gödel

est plus complexe qu'en Prolog. Vu que Gödel dispose d'une partie déclaration et d'une partie définition, il offre non seulement accès aux clauses du programme mais également à la partie déclaration. En Prolog, un programme a accès à ses clauses. Par contre en Gödel, le programme et le méta-programme sont différents.

Gödel représente de manière interne un programme au moyen de types, de constantes, de fonctions. Un certaine nombre de prédicats permettent de manipuler cette représentation interne. Par exemple le prédicat `And` a trois arguments: une première formule, une seconde formule et la conjonction des deux premières. Il peut donc être utilisé pour obtenir la représentation interne de $A \wedge B$ à partir de A et de B.

Gödel permet également d'exécuter un programme. Le prédicat `Succeed` avec comme arguments une représentation interne d'un programme, une représentation interne d'un but et une représentation interne d'une substitution est vrai si le programme exécuté avec le but donne la substitution. `Succeed` permet donc de trouver la réponse calculée de l'exécution du programme pour un but donné. Le prédicat `Fail` avec comme arguments: une représentation interne d'un programme et d'un but est vrai si le programme exécuté avec le but échoue.

2.5 Conclusion

Dans ce chapitre nous avons abordé trois langages logiques. La logique des prédicats du premier ordre apporte un fondement théorique et mathématique à la programmation logique. Prolog et Gödel sont deux langages de programmation logique. Ces langages s'écartent de la théorie pour rendre l'exécution d'un programme possible et efficace.

Prolog est le premier langage de programmation logique et date du début des années septantes. Il constitue un langage de référence en programmation logique. Au cours des vingt dernières années, il a subi de nombreuses modifications et la théorie a évolué. Gödel, par contre est un langage récent toujours en cours de développement et a donc profité des vingt dernières années d'expériences et de développement en programmation.

Le langage Gödel nous semble intéressant. L'aspect déclaratif apporte un peu plus de lourdeur au programme mais améliore nettement sa lisibilité. Il permet en outre de détecter certaines erreurs lors de la compilation, d'imposer plus de rigueur, et d'améliorer l'efficacité du code objet. Les possibilités de méta-programmation nous intéressent essentiellement car elles ont permis le développement de manière élégante du module évaluation de notre mémoire (cfr 6).

La présentation de l'implémentation au cours des prochains chapitres de ce mémoire se fera en Gödel. Toutefois, le développement pratique peut être réalisé en Prolog aussi bien qu'en Gödel. Dans la suite, plutôt que de spécifier le programme à la virgule près, nous tenterons de présenter les idées sous-jacentes. Le code complet peut être trouvé en annexe.

Chapitre 3

Le langage

3.1 La langue française

La distinction entre langage et langue est assez trouble. La définition du **langage** [Petit Robert] est :"Fonction d'expression de la pensée et de communication entre les hommes, mise en oeuvre au moyen d'un système de signes vocaux (parole) et éventuellement de signes graphiques (écriture) qui constitue une langue". Il distingue ensuite le langage naturel représenté par les langues du monde, du langage artificiel (ou formel) basé sur des axiomes, des règles et des lois de formations des énoncés. La **langue** est définie comme étant :"Système d'expression du mental et de communication commun à un groupe social (communauté linguistique)". Le langage correspond au concept abstrait de communication, d'expression entre les hommes, tandis que la langue est un moyen concret d'expression et de communication.

La langue et en particulier le français constitue l'objet principal d'étude de cette section. La **syntaxe** est l'ensemble des règles qui concernent le rôle et les relations des mots dans la phrase. Elle associe à chaque mot et groupe de mots une classe grammaticale et contient des règles de composition de groupe de mots en fonction de leur classe. Le mot constitue l'élément de base des expressions du langage. La phrase est constituée d'un assemblage logique et organisé de mots (exprimant un sens complet). Une phrase peut être de deux types, soit simple ou soit composé. Une phrase simple ou proposition est composée d'un sujet, d'un verbe et parfois d'un complément, par exemple, "*Jean voit un homme*", "*La femme entre*", "*Elle s'assied* ". Une phrase composée est une phrase constituée de plusieurs expressions, par exemple "*Si un homme admire le roi alors il l'acclame*". Le sujet peut recouvrir plusieurs formes: un groupe nominal (c'est à dire un déterminant et un nom), un pronom ou un nom propre. Il se place en général devant le verbe mais parfois après le verbe (dans une phrase interrogative par exemple). Le complément du verbe se place en général après le verbe et prend la forme d'un groupe nominal, d'un pronom, d'un nom propre, ...

En supposant le français connu du lecteur, notre objectif n'est pas de faire un relevé systé-

matique de la grammaire mais de percevoir les difficultés d'une analyse automatique des constituants d'une phrase. Une telle analyse consiste à décomposer une phrase en propositions, ensuite à décomposer celle-ci en sujet, verbe, complément, et ainsi de suite jusqu'à retrouver la classe grammaticale de chaque mot. Par exemple, la phrase "*Jean voit un homme*" se décompose en

Phrase

Groupe Nominal: Jean

Groupe Verbal

Verbe Transitif: voit

Groupe Nominal

Déterminant: un

Nom: homme

Si toutes les phrases de la langue française étaient aussi simples, elles seraient facilement analysables. Hélas, il n'en est pas toujours ainsi. Le sujet est parfois placé derrière le verbe, le complément d'objet n'a pas toujours la même position, les autres compléments (tels que le complément d'objet indirect, les compléments de lieu, de temps) compliquent l'analyse. Par exemple, "*L'homme que voit Jean joue de la guitare*" contient deux propositions: la proposition principale "*L'homme joue de la guitare*" qui n'a pas d'objet direct, mais un objet indirect "*de la guitare*", et la proposition relative "*que voit Jean*" où "*Jean*" est le sujet de "*voit*" et "*que*" est un pronom relatif à "*homme*". Par contre dans "*Jean joue le rôle du Bourgeois Gentilhomme*", "*le rôle du Bourgeois Gentilhomme*" est complément d'objet direct de "*joue*".

Les règles d'accord compliquent encore la syntaxe de la langue. Le sujet s'accorde avec le verbe; le déterminant et l'adjectif s'accordent avec le nom; ... Le nom est caractérisé par un genre et un nombre: soit singulier ou pluriel et soit féminin ou masculin. De même, les déterminants et les adjectifs ont un genre et un nombre. Le déterminant, l'adjectif et le nom doivent avoir le même genre et le même nombre. La personne et le nombre du sujet doivent correspondre à la personne et au nombre du verbe. Ainsi une phrase telle que "*les enfant jouent dans la bois*" ne respecte pas l'accord du nombre entre "*enfant*" et "*les*" et l'accord du genre entre "*bois*" et "*la*". De même la phrase "*les enfants joues dans le bois*" ne respecte pas l'accord de la personne et du nombre entre "*joues*" et "*les enfants*". Ce dernier exemple met en évidence une autre difficulté de la langue. Dans la phrase précédente "*joues*" est un verbe (jouer deuxième personne du singulier de l'indicatif présent). Par contre dans la phrase "*le clown a les joues rouges*", "*joues*" est un nom féminin pluriel.

Automatiser le traitement de l'analyse d'une expression du langage n'est pas une tâche évidente. Nous nous contenterons d'un petit fragment du français pour illustrer la définition formelle de la syntaxe au moyen d'axiomes et de règles, et pour automatiser l'analyse d'une expression. Ce fragment sera complété par la suite.

3.2 Le langage : approche formelle

L'objectif de cette section est de définir formellement l'ensemble des expressions admises dans le langage et rien qu'elles. Une première approche consisterait à utiliser une définition par extension énonçant toutes les expressions du langage. Par exemple: {j'ai,tu as,il a,nous avons,vous avez,ils ont}. Le nombre d'expression d'un tel langage est évidement limité. Par contre, les expressions du français (ou de tout autre langue) sont illimitées. Pour s'en convaincre considérons la phrase "*l'homme qui a vu l'homme qui a vu l'homme qui a vu l'homme qui a vu l'homme, ...*" se répétant progressivement. Il n'est donc pas possible de caractériser l'ensemble des expressions d'un tel langage par extension. Nous définirons donc un langage comme suit:

Définition Le langage est basé sur deux ensembles:

- un lexique composé de couples de mots et de catégories
- et des règles de composition qui associent un n-uplet de catégories, à une catégorie.

Exemple Par exemple, prenons le lexique suivant

(un, DET)

$(homme, NOM)$

$(Jean, GN)$

$(voit, VT)$

et supposons les règles de composition suivantes:

$R_1(DET, NOM) = GN$

$R_2(VT, GN) = GV$

$R_3(GN, GV) = P$

Ce langage permet d'exprimer une phrase telle que *"Jean voit un homme"*

Définition La **catégorie d'une suite de mots** m est définie:

- s'il existe une partition $m_1, \ldots, m_n$ de m telle que la catégorie C_i de m_i est définie;
- s'il existe une relation R tel que $R(C_1, \ldots, C_n)$ est défini.

La catégorie C de m est $R(C_1, \ldots, C_n)$. L'**analyse d'une suite de mots** consiste à rechercher sa catégorie.

Exemple Par exemple, la catégorie de *un homme* est GN puisque la catégorie de *un* est DET, la catégorie de *homme* est NOM et que $R_1(DET, NOM) = GN$. De même la catégorie de *voit un homme* est GV et la catégorie de *Jean voit un homme* est P.

Définition L'arbre d'analyse d'une suite de mots m est définie comme suit:

- si la suite n'est composée que d'un seul mot m et qu'il existe c tel que (m, c) appartient au lexique, l'arbre d'analyse est (m,c);
- si la suite de mot m peut être partitionnée en sous-suites non vide $m_1, \ldots, m_n$ de catégorie $(C_1, \ldots, C_n)$ et d'arbre d'analyse $a_1, \ldots, a_n$ et s'il existe R tel que $R(C_1, \ldots, C_n) = C$, alors l'arbre d'analyse de m est (C,$a_1, \ldots, a_n$).

Exemple Par exemple, l'arbre d'analyse de *Jean voit un homme* est

$(P$,

$(GN, Jean)$,

$(GV,$

$(VT, voit),$

$(GN,$

$(DET, un),$

$(NOM, homme)$

$)$

$)$

$)$

3.3 Un fragment du français: version 1

Cette section présente un fragment élémentaire du français et le défini de manière formelle au moyen d'un lexique et de règles de composition. Notre objectif est essentiellement de permettre une extension future la plus aisée possible du fragment.

Nous utilisons une présentation différente du formalisme de la section 3.2.

Définition Les catégories sont définies au moyen de catégories élémentaires et de fonctions constructrices.

- Les catégories élémentaires sont P et E.
- Si A et B sont deux catégories alors A/B et $A\backslash B$ sont des catégories.

De manière intuitive nous pouvons dire que A/B correspond à un A moins un B à la fin. Par exemple, un déterminant est un groupe nominal moins un nom. La catégorie DET est donc définie de la manière suivante $DET = GN/N$. De même $A\backslash B$ est un B moins un A à l'avant. Par exemple, un groupe verbal est une phrase moins quelque chose devant. La définition d'un groupe verbal est donc $GV = E\backslash P$

En outre, les règles de compositions ne sont plus basées sur les catégories en tant que telles mais sur leurs définitions.

- $R_1(A/B, B) = A$
- $R_2(A, A\backslash B) = B$

La première règle correspond à l'idée intuitive suivante: Vu que A/B est un A moins un B à la fin, un A/B suivi d'un B est un A. De même un A placé devant un $A\backslash B$ forme un B.

L'avantage de ces règles, sur les règles données dans la section 3.2 est qu'elles ne dépendent plus de la grammaire, mais des catégories de la grammaire. Ces règles sont identiques quel que soit le langage. A la fin de ce chapitre, nous donnons une grammaire anglaise qui utilise les mêmes règles de réduction. De même, lorsque nous étendrons le fragment du français pour tenir compte de nouvelles classes ou des accords, les règles de réduction resterons inchangées. La méthode d'analyse des expressions du langage à partir de ces règles est donc invariable quelque soit le lexique utilisé.

Pour définir un langage, il est maintenant nécessaire de donner:

- un lexique contenant les mots et leurs catégories ;
- un dictionnaire contenant les définitions des catégories en termes de catégories élémentaires.

Le fragment élémentaire du français considéré dans cette section ne contient que des phrases simples composées d'un sujet suivi d'un verbe suivi éventuellement d'un complément d'objet direct (sans tenir compte des accords).

Catégorie	Définition
N	P/E
GN	$P/(E\backslash P)$
GV	$E\backslash P$
VT	GV/GN
DET	GN/N

Le lexique de base est le suivant:

Mot	Categorie
un	DET
homme	N
voit	VT
Jean	GN

L'expression *un homme* correspond à un DET suivi d'un N c'est à dire, en reprenant la définition de DET, un GN/N suivi d'un N. La règle

$$R_1(GN/N, N) = GN$$

nous indique que la catégorie de *un homme* est GN.

L'expression *voit un homme* correspond à VT suivi d'un GN c'est à dire un GV/GN suivi d'un GN d'où, en appliquant la règle

$$R_1(GV/GN, GN) = GV$$

la catégorie de *voit un homme* est GV.

Finalement l'expression *Jean voit un homme* correspond à un GN suivi d'un GV c'est à dire un $P/(E\backslash P)$ suivi d'un $E\backslash P$ par la règle

$$R_1(P/(E\backslash P), E\backslash P) = P$$

la catégorie de *Jean voit un homme* est P.

Le fragment actuel ne permet qu'un nombre très limité de phrases différentes. Le lexique peut être étendu sans problème, en y ajoutant des mots et leurs catégories. Ainsi, l'introduction de

Mot	Catégorie
entre	GV
une	DET
femme	N

permet une phrase telle que *une femme entre.*

La difficulté d'étendre le fragment ne se situe pas au niveau du lexique mais au niveau de la définition des catégories. Par exemple, pour ajouter les phrases conditionnelles telles que *si un homme admire le roi, il l'acclame*, il faut définir la catégorie du *si*. *Si* est un mot tel que, suivi d'une phrase, il constitue une expression intermédiaire qui suivi d'une phrase donne une phrase. La catégorie de cette expression intermédiaire est P/P. En effet, $R_1(P/P, P) = P$. La catégorie

du *si* est donc $(P/P)/P$ puisque $R_1((P/P)/P, P) = P$. Une phrase conditionelle contient parfois le mot *alors*. Il est tel que suivi d'une phrase, il donne une phrase. La catégorie de *alors* est donc P/P.

Le *l'* ou plus généralement le pronom *le* est un mot placé devant un verbe transitif VT. Sa catégorie est donc GV/VT. En effet, un GV/VT suivi d'un VT donne un GV par application de la règle $R_1(GV/VT, VT) = GV$.

Un autre type de phrases qu'il est utile d'ajouter sont les phrases négatives telles que *le directeur n'utilise pas d'ordinateur*. Le mot *pas* est précédé d'un VT ou d'un GV. Nous allons attribuer deux catégories au mot *pas*. Si *pas* est précédé d'un VT, sa catégorie est $VT\backslash VT$. S'il est précédé d'un GV, sa catégorie est $GV\backslash GV$.

La catégorie de *ne* est GV/GV. En effet, *ne* suivi d'un groupe verbal donne un nouveau groupe verbal.

Le dernier mot que nous allons ajouter au fragement est *avec*. Nous pourrons alors utiliser une phrase telle que *Jean voit l'homme avec le chapeau.* La catégorie de *avec* est $(N\backslash N)/GN$. En effet, $(N\backslash N)/GN$ suivi d'un GN, comme *le chapeau*, donne une structure intermédiare *avec le chapeau* de catégorie $N\backslash N$. Cette structure intermédiaire précédée d'un N, par exemple, *homme*, donne un N "*homme avec le chapeau.*"

En résumé, le fragment du français défini actuellement consiste en

Catégorie	Définition	Catégorie	Définition
N	P/E	GN	$P/(E\backslash P)$
GV	$E\backslash P$	VT	GV/GN
DET	GN/N	SI	$(P/P)/P$
$ALORS$	P/P	NE	(GV/GV)
PAS_1	$VT\backslash VT$	PAS_2	$GV\backslash GV$
$PACC$	GV/VT	$AVEC$	$(N\backslash N)/GN$

Mot	Catégorie	Mot	Catégorie
un	DET	le	DET
une	DET	de	DET
homme	N	roi	N
femme	N	directeur	N
ordinateur	N	Jean	NP
il	NP	utilise	VT
entre	VT	voit	VT
admire	VT	si	SI
alors	ALORS	avec	AVEC
ne	NE	n	NE
pas	PAS_1	pas	PAS_2
l	PACC	le	PACC

3.4 Analyse automatique d'une phrase

L'objectif de cette section est de concevoir un programme logique qui analyse automatiquement une phrase. Pour ce faire, deux stratégies sont envisageables: soit une analyse de haut en bas, soit une analyse de bas en haut.

3.4.1 Stratégies

Analyse de haut en bas. Une première technique pour analyser une phrase consiste à partir de la suite de mots. Si cette suite est composée d'un seul mot, l'analyse est terminée. Par contre, si la suite est composée de plusieurs mots, la technique consiste à rechercher deux sous-suites complémentaires "analysables", c'est à dire deux sous-suites sur lesquelles la technique est réappliquée et pour lesquelles il existe une règle de combinaison de leur catégorie.

La difficulté de cette technique est de déterminer la séparation de la suite initiale en deux sous-suites "analysables". Pour ce faire une méthode brutale consiste à générer d'abord deux sous-suites quelconques, à tester ensuite si elles peuvent être analysées et finalement, si l'opération précédente réussi, à tenter de combiner les catégories des deux sous-suites. Il va de soit que pour une suite de m mots, il est possible de générer m-1 couples de sous-suites: le premier mot et les m-1 derniers ; les deux premiers et les m-2 derniers ;...

Cette technique n'est pas très efficace puisque pour m mots il risque d'y avoir m-1 partitions générées. Il faut également ajouter que chaque sous-suite générée est également analysée et produit également un certain nombre de partitions, ...

Analyse de bas en haut. La technique de haut en bas se base sur l'ensemble des mots de la suite, pour redescendre jusqu'à la catégorie de chaque mot. Un autre technique consiste au contraire à partir de la catégorie de chaque mot et à remonter à la catégorie de l'ensemble des mots. Précisément, elle consiste à traduire la suite de mots en une suite de catégories et à réduire cette suite.

La réduction d'une suite de catégories consiste :

1. *primo*, à rechercher dans la suite deux catégories contiguës qui peuvent être combinées par une règle, et à remplacer ces deux catégories par leur combinaison;
2. *secundo*, tant que l'opération précédente est possible la recommencer,
3. *tertio*, s'il n'est plus possible de recommencer l'opération, la suite est réduite. Dans ce cas, deux éventualités sont possibles.
   - Soit la suite ne contient qu'une seule catégorie et la suite de mots est de cette catégorie.
   - Soit la suite contient plusieurs catégories et alors la suite de mots n'est pas analysable.

Cette technique évite le problème de la génération d'une partition. Cependant, elle impose la génération d'une suite de catégories correspondant à chaque mot de la suite de mots. Hors, un mot peut appartenir à plusieurs catégories. Il y a donc également un problème d'efficacité. Il faut générer la bonne liste de catégories. En pratique, il est nécessaire de générer les suites de catégories, puis de tester si elles peuvent être réduites. Le nombre de suites de catégories qui peuvent être générées à partir d'une suite de mots dépend en partie de la longueur de la suite mais surtout du nombre de catégories différentes de chaque mot de la suite. Supposons par hypothèse que le nombre moyen de catégories par mot est m. Pour une suite de longueur n, il y a m^n suites qui peuvent être générées. En effet, pour une suite de 1 mot, il y en a m ; pour une suite de n mots supposons qu'il y en ait m^n, alors pour une suite de $n+1$ mots, il y en a $m * m^n$. Ce nombre moyen m n'est pratiquement pas déterminable vu qu'il dépend du nombre de catégories de chaque mot, mais aussi de la fréquence d'apparition de chaque mot dans une phrase. Toutefois, la plupart des mots n'ont qu'une seule catégorie ce qui fait tendre cette moyenne vers un.

3.4.2 Implémentation

Nous allons utilisé ces deux stratégies pour concevoir un programme logique qui analyse une phrase du fragment. Dans un premier temps, l'objectif du programme est de trouver la catégorie d'une suite de mots donnée. Dans un second temps, nous envisagerons la construction de l'arbre d'analyse.

Avant de passer à la définition des prédicats du programme, il est nécessaire de fixer des conventions de représentations des données manipulées.

Représentation des suites. Une liste est une structure de donnée très utile. Une liste est soit vide, soit composée d'une tête qui contient un élément et d'une queue qui est une liste. Une liste peut être représentée:

- par une constante `vide`
- et une fonction `cons (tete, queue)` .

Ainsi, la suite 1,2,3,4 peut être représentée par la liste:

```
cons(1,cons(2,cons(3,cons(4,vide)))) .
```

La plupart des langages logiques (notamment Prolog et Gödel) permettent d'utiliser cette structure de liste. Par convention, les listes sont notées entre "[","]" et les éléments sont séparés par des virgules ",", (e.g. `[1,2,3,4]`). Le symbole " `|` " est utilisé pour séparer la tête de la queue d'une liste (e.g. `[1|[2,3,4]]`).

Une suite de mots est représentée par une liste. Les mots sont représentés par des chaines de caractères classiques, par convention entre " . Ainsi la phrase *Jean voit un homme* est représentée par `["Jean","voit","un","homme"]` .

Représentation des catégories. Pour rappel, une catégorie élémentaire est soit P, soit E. Les catégories complexes sont "construites" à partir de catégories plus simples grâce aux fonctions / et \.

Les catégories sont représentées par:

- les constantes `P` et `E`;
- les fonctions / et \ infixées qui prennent pour arguments deux catégories et produisent une nouvelle catégorie.

Ainsi A/B est représenté par `A/B` . De même $A\backslash B$ est représenté par `A\ B` .

Les définitions de catégories sont représentées au moyen d'un prédicat `syncat` qui a pour premier argument un nom de catégorie et pour second argument la définition de cette catégorie au moyen de P, E, /, \.

La définition de ce prédicat est assez simple car elle correspond à une retranscription des définitions des catégories données dans la syntaxe.

```
syncat(N,P/E)

syncat(GN,P/(E \ P))

syncat(GV,E \ P)

syncat(VT,(E \ P)/(P/(E \ P)))

syncat(DET,(P/(E \ P))/(P/E))

syncat(SI,(P/P)/P )

syncat(ALORS, P/P)

syncat(NE, (E\ P)/(E\ P)

syncat(PAS1,((E\ P)/(P/(E\ P))) \ ((E\ P)/(P/(E\ P))))

syncat(PACC, GV/VT)

syncat(AVEC,(N\ N)/GN)
```

Il faut ajouter les cas de base:

`syncat(P,P)`

`syncat(E,E)`

et les cas complexes:

`syncat(a/b,c/d)` ← `syncat(a,c)` ∧ `syncat(b,d)`

`syncat(a\b,c\d)` ← `syncat(a,c)` ∧ `syncat(b,d)`

Représentation du lexique. Le lexique est représenté par un prédicat ayant pour premier argument un mot et pour second argument sa catégorie. La définition de ce prédicat est également très simple. Elle se base sur le lexique donné dans la syntaxe.

```
lexique("un", DET)

lexique("le",DET)

...

lexique("homme",N)

lexique("roi",N)

lexique("femme",N)

...

lexique("Jean",NP)

lexique("il",NP)

...

lexique("utilise", VT)

lexique("entre", GV)

...
```

Représentation des règles de combinaisons. Les règles de combinaison sont aussi représentées par un prédicat. Ces règles de combinaison, correspondent en fait à la réduction d'une liste de deux catégories en une liste d'une seule catégorie. La liste `[a/b,b]` (resp. `[a, a\b]`

) se réduit par application des règles de combinaison en `[a]` (resp. `[b]`).

La réduction d'une liste de deux éléments se définit donc de la manière suivante:

```
Red([a/b,b],[a])
```

```
Red([a,a\b],[b])
```

Stratégie de haut en bas. La stratégie de haut en bas est définie par un prédicat à deux arguments qui définit pour chaque suite de mots l, sa catégorie c (si elle existe). Le prédicat échoue si la catégorie n'existe pas.

Si la suite l ne contient qu'un seul élément m, la catégorie c de cet élément est trouvée dans le lexique. La définition du prédicat est alors

```
Categorie(l,b)
        ← l = [m]
        ∧ Lexique(m,c)
        ∧ Syncat(c,b)
```

Si la suite l contient plusieurs éléments, alors supposons que m_1 et m_2 soit une partition de l, telle que la catégorie de m_1 est c_1 et la catégorie de m_2 est c_2. La catégorie c de la suite l est obtenu par application d'une règle de composition avec c_1 et c_2. La définition du prédicat est

```
Categorie(l,c)
        ← Partition(m1,m2,l)
        ∧ Categorie(m1,c1)
        ∧ Categorie(m2,c2)
        ∧ Red([c1,c2],[c])
```

Les sous-listes m_1 et m_2 forment une partition de l si elles sont non vide et si leur concaténation donne l.

```
Partition(m1,m2,l)
        ← m1 ≠ []
        ∧ m2 ≠ []
```

```
        ∧ Append(m1,m2,l)
```

La concaténation de deux listes m_1 et m_2 est définie par analyse de la première liste m_1. Si m_1 est une liste vide, la concaténation de m_1 et de m_2 est m_2. Si m_1 n'est pas une liste vide c'est à dire égale `[e | q]` , la concaténation de m_1 et de m_2, correspond à ajouter e en tête de la liste i formée par la concaténation de q et de m_2.

```
Append([],m2,m2)

Append([e|q],m2,[e|i]))

        ← Append(q,m2,i)
```

Stratégie de bas en haut. La stratégie de bas en haut est composé de deux phases: la génération d'une liste de catégories et la réduction d'une liste de catégories.

Génération d'une liste de catégories. Générer la liste de catégories d'une liste vide de mots correspond à générer une liste vide. Générer la liste de catégories d'une liste de mots dont l'entête est e et la queue est q consiste à premièrement générer cq la liste de catégories de q et à ajouter la catégorie c de e en tête de cette liste .

Le prédicat `Listcat` est défini de la manière suivante:

```
Listcat([],[])

Listcat([e|q],[c|cq])

        ← Listcat(q,cq)

        ∧ Lexique(e,b)

        ∧ Syncat(b,c)
```

Réduction d'une liste de catégories. La réduction d'une liste de catégories consiste à réduire deux catégories contiguës dans la liste tant que c'est possible. La réduction d'une liste est donc basée sur la réduction de deux catégories contiguës. Supposons que nous ayons à notre disposition le prédicat `Red2` qui réduit deux catégories contiguës dans une liste. `Redall` est un prédicat qui a pour argument deux listes de catégories. Le premier argument correspond à une liste non réduite, le second correspond à la liste réduite.

```
Redall(l,c)

        ← Red2(l,i) ∧ Redall(i,c).

Redall(l,l)
```

```
        ← ¬ Red2(l,i).
```

Le prédicat `Red2` définit la réduction de deux catégories contiguës. Pour ce faire, il regarde si les deux premiers éléments de la liste sont "réductibles", si oui il les réduits ; sinon la réduction de la liste correspond à la réduction de la liste en commençant au second élément.

```
Red2([c1,c2|r],[c|r] )

        ← Red([c1,c2],c)

Red2([c1|r],[c1|rr] )

        ← Red2(r,rr)
```

Représentation de l'arbre d'analyse. Un arbre est soit vide, soit composé de trois éléments une information, un sous-arbre gauche et un sous-arbre droit. La constante `vide` représente un arbre vide et la fonction `A` avec trois arguments: l'information, le sous-arbre gauche et le sous-arbre droit.

Les informations sont de deux types, soit le nom de la catégorie d'une suite de mots, soit un mot et sa catégorie. Ce dernier type d'information correspond en fait aux feuilles de l'arbre. Pour représenter les informations nous allons utiliser deux fonctions: La fonction I1 avec pour argument un nom de catégorie et la fonction I2 avec pour argument le mot et sa catégorie.

Par exemple: l'arbre d'analyse de *Jean voit un homme*

$(P$,

$(GN, Jean)$,

$(GV,$

$(VT, voit),$

$(GN,$

$(DET, un),$

$(NOM, homme)$

$)$

$)$

$)$

se représente en terme de programmes,

```
A(P,
    A(I2("Jean",GN),vide,vide),
    A(I1(GV),
        A(I2("voit",VT),vide,vide),
        A(I1(GN),(
            A(I2("un",DET),vide,vide),
            A(I2("homme",NOM),vide,vide)
        )
    )
)
```

Stratégie de haut en bas. L'arbre d'analyse d'une suite de mots réduite à un seul mot est un arbre composé de l'information (mot, catégorie) et des sous-arbres vides.

L'arbre d'analyse d'une suite de mots de catégorie c obtenue à partir de la combinaison des catégories c_1 et c_2 de deux sous-suites s_1 et s_2 d'arbres d'analyses a_1 et a_2 est un arbre ayant comme information c , comme sous-arbre gauche a_1 et comme sous-arbre droit a_2.

Le prédicat `Categorie` est défini de la manière suivante:

```
Categorie(l,b,A(I2(m,c),vide,vide)
        ← l = [m]
        ∧ Lexique(m,c)
        ∧ Syncat(c,b)
Categorie(l,c,A(I(e),t_m1,t_m2))
        ← Partition(m1,m2,l)
        ∧ Categorie(m1,c1, t_1)
        ∧ Categorie(m2,c2,t_2)
        ∧ Red([c1,c2],[c])
        ∧ Syncat(e,c)
```

Stratégie de bas en haut. Pour créer l'arbre d'analyse d'une suite de mots, le principe consiste à associer une liste d'arbres à la liste de catégories. Lors de la réduction de deux catégories contiguës c_1 et c_2 en c, les deux arbres associés sont réduits en un arbre dont l'information est c, le sous arbre gauche est l'arbre associé à c_1 et le sous arbre droit à celui de c_2.

```
Listcat([],[],[])

Listcat([e|q],[c|cq],[A(I2(e,b),vide,vide)|aq])

        ← Listcat(q,cq)

        ∧ Lexique(e,c)

        ∧ Syncat(b,c)

Redall(l,c,al,ac)

        ← Red2(l,b,al,ab)

        ∧ Redall(b,c,ab,ac)

Redall(l,l,al,al) ← ¬ Red2(l,_,al,_).

Red2([m1,m2|r],[m|r],[am1,am2|ar],[A(I1(m,am1,am2))|ar])

        ← Red([m1,m2],[m])

Red2([m1|r],[m1|rr],[am1|ar],[am1|arr])

        ← Red2(r,rr,ar,arr)
```

3.5 Extensions

3.5.1 Un fragment du français : version 2

Le premier fragment ne tient compte d'aucun accord. Dans cette section, nous présentons une extension du fragment et notamment de la notion de catégorie nécessaire pour tenir compte des accords entre le sujet et le verbe et entre le déterminant et le nom.

Le fragment actuel ne permet aucun accord étant donné qu'il n'y a aucune information concernant le genre, le nombre et la personne des mots. Un mot tel que *homme* est masculin, singulier et de catégorie N. Un mot tel que *un* est également masculin singulier mais de catégorie DET. L'expression *un homme* est de catégorie GN si le nom et le déterminant sont de même genre et de même nombre. Notre extension consiste à introduire une information complémentaire

à chaque catégorie.

Les déterminants et les noms sont caractérisés par un genre et un nombre. Les sujets et les verbes sont caractérisés par un nombre et une personne. Les catégories de bases sont représentées par des fonctions: P sans argument et E avec trois arguments, le premier représentant le nombre, le second le genre et le troisième la personne. Les définitions des catégories sont modifiées en conséquence.

On obtient ainsi:

Nom	Catégorie
$N(nb, gr)$	$P/E(nb, gr, *)$
$GN(nb, gr, pers)$	$P/(E(nb, gr, pers)\backslash P)$
$GV(nb, pers)$	$E(nb, *, pers)\backslash P$
$VT(nb, pers)$	$GV(nb, pers)/GN(*, *, *)$
$DET(nb, gr)$	$GN(nb, gr, *)/N(nb, gr)$
SI	$(P/P)/P$
$ALORS$	P/P
NE	$GV(nb, pers)/GV(nb, pers)$
PAS	$VT(nb, pers)/VT(nb, pers)$
$PACC$	$GV(nb, pers)/VT(nb, pers)$

Le symbole * représente une valeur quelconque. Les règles de combinaison ne sont pas modifiées. Cependant, les paramètres des catégories doivent s'unifier. Deux catégories $A(a_1, a_2, a_3)$ et $A(b_1, b_2, b_3)$ s'unifient s'il existe une substitution μ telle que $A(a_1, a_2, a_3)\mu = A(b_1, b_2, b_3)\mu$.

Le lexique est modifié pour tenir compte des caractéristiques liées à chaque mot.

Mot	Nom
un	$DET(Sg, Ma)$
une	$DET(Sg, Fe)$
les	$DET(Pl, *)$
le	$DET(Sg, Ma)$
la	$DET(Sg, Fe)$
...	
homme	$N(Sg, Ma)$
hommes	$N(Pl, Ma)$
femme	$N(Sg, Fe)$
...	
voit	$VT(Sg, 3p)$
voient	$VT(Pl, 3p)$
...	
jean	$GN(Sg, 3p)$
il	$GN(Sg, 3p)$
ils	$GN(Pl, 3p)$
...	
entre	$GV(Sg, 3p)$

Par exemple: *un homme* est de catégorie GN(Sg,Ma,*). En effet, étant donné que *homme* est un $N(Sg, Ma)$ et *un* est un DET(Sg,Ma), la combinaison donne

$$GN(Sg, Ma, *)/N(Sg, Ma) + N(Sg, Ma) = GN(Sg, Ma, *).$$

En guise d'autre exemple, considérons *un homme entre*. Le groupe *un homme* est un $GN(Sg, Ma, *)$ et *entre* est un $GV(Sg, 3p)$. La réduction donne donc

$$P/(E(Sg, Ma, *)\backslash P) + E(Sg, *, 3p) = P.$$

Toutefois *une homme* n'est d'aucune catégorie puisque la combinaison

$$GN(Sg, Fe, *)/N(Sg, Fe) + N(Sg, Ma)$$

n'est pas possible.

3.5.2 Un fragment de l'anglais

L'anglais étant la langue internationale par excellence, le programme développé au cours du stage se basait sur un lexique et des catégories adaptées à cette langue. Nous nous proposons de le présenter brièvement en insistant sur les différences essentielles entre le français et l'anglais. La structure des phrases dans les deux langues sont pratiquement identiques. Cependant, il existe deux différences par rapport au fragment: d'une part la position du pronom complément direct et d'autre part la forme de la négation.

Le pronom objet direct. En anglais, le pronom *him* se place derrière le verbe. Sa catégorie est donc la même que celle d'un groupe nominal (noun phrase). Toutefois, *him sees he* n'est pas correct. Pour éviter ce problème, il faut introduire une nouvelle caractéristique de catégories: le cas (Accusatif ou Nominatif).

Le fragment anglais de base est donc:

Catégorie	Définition
$N(nb)$	$P/E(nb,*)$
$NP(nb,pers,case)$	$P/(E(nb,pers,case)\backslash P)$
$VP(nb,pers,case)$	$E(nb,pers,Nom)\backslash P$
$TV(nb,pers)$	$VP(nb,pers,Nom)/NP(*,*,Acc)$
$DET(nb)$	$NP(nb,*,*)/N(nb)$
IF	$(P/P)/P$
$THEN$	P/P

et le lexique contient:

Mot	Catégorie
him	$NP(Sg,3p,Acc)$
he	$NP(Sg,3p,Nom)$

La phrase négative. En anglais, la négation est exprimée à l'aide d'un auxiliaire (do, en général) suivi d'une négation, suivi du verbe. Un auxiliaire et une négation donne une construction intermédiaire qui suivi d'un groupe verbal donne un groupe verbal. La construction intermédaire est donc $VP(nb,pers)/VP(nb,pers)$. Une négation est donc telle que, précédée d'un auxiliaire, elle forme un $VP(nb,pers)/VP(nb,pers)$. La catégorie d'une négation est $AUX\backslash VP(nb,pers)/VP(nb,pers)$. La catégorie d'un auxiliaire est aussi $VP(nb,pers)/VP(nb,pers)$. En effet, un auxiliaire suivi d'un groupe verbal est encore un groupe verbal.

Par suite le fragment précédent est complété par les catégories suivantes:

Catégorie	Définition
AUX	$VP(nb,pers)/VP(nb,pers)$
NEG	$AUX\backslash AUX$

3.5.3 Les extensions de l'implémentation.

La construction du programme a été réalisée de sorte qu'il n'est nécessaire en général de modifier que le lexique (prédicat Lexique) et la définition des catégories (prédicat Syncat) en fonction des nouvelles extensions.

3.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'aspect syntaxique du langage naturel. Les règles de grammaire usuelles pour l'être humain ne peuvent pas être exploitées directement par l'ordinateur. La syntaxe doit être exprimée au moyen de règles formelles que l'ordinateur applique pour analyser la structure d'une phrase. Il existe de nombreux formalismes. Ceux-ci reposent en général sur deux types d'ensemble: un ensemble de couples formés d'un mot et de sa classe grammaticale, et un ensemble de règles de combinaisons. Nous avons développé un formalisme particulier basé sur la notion de catégorie. Chaque mot est associé à une catégorie et les règles de combinaison sont basées sur la définition des catégories. Celles-ci correspondent à l'idée intuitive qu'un mot (ou groupe de mots) de catégorie A suivi (ou resp. précédé) d'un autre mot (ou groupe de mots) de catégorie $A\backslash B$ (resp. B/A) forment un groupe de mot de catégorie B.

Nous avons vu qu'il était possible de construire un programme reposant sur ces principes pour analyser de manière automatique la structure d'une phrase. L'idée consiste grossièrement:

- soit à partitionner la suite de mots en deux sous-suites et à analyser celles-ci récursivement. La combinaison (si possible) des catégories de ces deux sous-suites donne la catégorie de la suite de mots.
- soit à traduire la suite de mots en une liste de catégories et à réduire cette liste de catégories. La réduction d'une liste de catégorie consiste tant que possible à remplacer deux catégories adjacentes par leur combinaison.

La version finale du programme produit une représentation de l'arbre d'analyse. Nous nous intéresserons dans le chapitre 4 à l'aspect sémantique du langage naturel et nous définirons un langage de représentation. Dans le chapitre 5 nous reviendrons à un aspect plus pratique en traduisant l'arbre d'analyse dans cette représentation au moyen d'un programme.

Chapitre 4

La sémantique

4.1 Introduction

La sémantique définit le sens des objets du langage: les mots, les phrases, les textes, ...Le sens d'un mot dépend de plusieurs facteurs. Dans le chapitre 2, nous avons défini la sémantique d'une expression logique par une interprétation. Par exemple, le sens de "R(a)" est donné par une fonction qui réalise une correspondance entre une expression et un ensemble. Pour le langage naturel, une démarche similaire peut être abordée. Le mot *"pomme"*, par exemple, fait référence à un élément de l'univers qui correspond à une variété particulière de fruits. Ce sens donné au mot dépend de plusieurs paramètres, notamment de sa fonction grammaticale. Dans le chapitre 3, nous avons vu des exemples où un mot n'a pas le même sens selon qu'il est un verbe ou un nom. Dans les phrases suivantes *"Tu joues à la balle"* et *"Le clown a les joues rouges"*, le mot *"joues"* n'a pas le même sens. Dans un cas, il correspond à l'action de jouer et dans l'autre, à une partie du visage. Une analyse correcte d'une phrase est donc nécessaire pour comprendre le sens d'une phrase. Le contexte joue un rôle important dans la compréhension. Ainsi dans l'exemple, *"un homme ramasse un couteau"*, l'objet ramassé correspond-il à un objet coupant qui se trouve dans une cuisine ou à un coquillage qui se trouve au bord de la plage?

Le sens d'une expression est définie par une interprétation qui est une fonction des expressions vers un ensemble. Dans le cas du langage naturel, cet ensemble correspond au monde réel ou plus modestement à une représentation d'une partie du réel. Nous ne représenterons que la partie du réel qui nous intéresse. Par exemple, le contexte: Jean, Arthur, Hector sont des hommes, Jean voit Arthur et Hector sera représenté au moyen de l'ensemble suivant {*Jean , Arthur, Hector*}. Ainsi le mot *"Jean"* correspondra à *Jean* . Le verbe *"voit"* à une relation *voit(x,y)* qui est vraie pour (x, y)=(*Jean,Arthur)* ou pour (x, y) = *(Jean,Hector)*. Le nom *"homme"* correspond également à une relation *homme(x)* qui est vraie pour x appartenant à {*Jean, Arthur, Hector*}. L'interprétation d'une phrase telle que *"Jean voit un homme"* correspond à: il y a un certain *Jean* et un *homme(x)* tels que *voit(Jean,x)*. Les valeurs possibles pour x sont *Arthur* ou *Hector*.

Dans la plupart des langues naturelles et en particulier en français, il existe deux types de déterminants: les déterminants indéfinis tels que un, une, des,...et les déterminants définis tels que le, la, les, ce, cette, mon, ton,...Par suite, la phrase *"Jean voit un homme"* et la phrase *"Jean voit l'homme"* n'ont pas tout à fait le même sens. Dans la première, il s'agit d'un homme quelconque, par contre dans la seconde, il s'agit d'un homme particulier. Cette nuance intervient dans l'interprétation, le sens de *"un homme"* correspond à la relation *"homme(x)"* où la valeur de x est quelconque. Par contre le sens de *"l'homme"* est donné également par la fonction *"homme(x)"* cependant la valeur de x est obligatoirement unique. Dans le cas contraire la phrase est erronée.

4.2 Représentation logique

L'interprétation du langage naturel donnée précédemment est informelle. Une spécification formelle serait beaucoup plus pratique car elle permettrait de faciliter l'interprétation et la manipulation du langage. Cependant, une telle spécification est difficile à réaliser et permet difficilement de rendre compte d'un langage complet et réaliste. Nous pourrons et devrons cependant, nous en contenter car elle facilite grandement la "compréhension" d'un fragment de langage naturel aux ordinateurs. Par compréhension du langage naturel par un ordinateur, nous entendons que ce dernier puisse réagir à une requête en langue française dans un contexte déterminé. Il n'est évidemment pas pensable d'imaginer un dialogue entre un humain et un ordinateur comme un dialogue entre personnes.

Dans la section 4.1, l'interprétation donnée à une expression ressemble à l'interprétation que nous avions donnée à la logique des prédicats du premier ordre. (cfr section 2.2.2 L'interprétation de $\exists v_1(v_1 = Jean \wedge \exists v_2(homme(v_2) \wedge voit(v_1, v_2)))$ correspond à l'interprétation donnée à la phrase *"Jean voit un homme"*. La nuance faite à propos des déterminants peut également être formalisée en utilisant le quantificateur $\exists$ ou $\exists!$.

L'utilisation des quantificateurs n'est pas toujours évidente, notamment dans le cas, par exemple, où l'expression $\exists x\ homme(x)$ n'est pas vérifiée. Il n'est donc pas possible d'assigner une valeur à x. Hilbert et Bernays proposent comme valeur d'assignation de x dans $\exists x\varphi(x)$, soit une valeur de x qui vérifie $\varphi(x)$, soit si $\varphi(x)$ n'est pas vérifiable une valeur arbitraire quelconque. Cette assignation est définie au moyen d'une fonction Φ qui pour tout sous-ensemble d'éléments de l'univers retourne un de ses éléments. La valeur assignée à x est par cette définition l'élément de l'univers donné par Φ avec pour argument le sous-ensemble des valeurs x' vérifiant $\varphi(x')$. Ainsi si φ n'est pas vérifiable, la valeur assignée à x correspond à la valeur donnée par Φ avec pour argument l'ensemble vide.

Revenons à notre exemple. La relation $\exists x homme(x)$ est vraie dans le cas du contexte donné plus haut pour $\forall x \in \{Hector, Arthur, Jean\}$. Supposons que la fonction Φ avec comme argument $\{Hector, Arthur, Jean\}$ donne $Jean$. La valeur assignée à x est donc $Jean$.

Cette définition de l'assignation n'est évidemment pas très pratique. En effet, considérons

le cas où nous avons $\exists x homme(x) \wedge \exists y homme(y)$. Les valeurs assignées à x et à y par Φ sont évidemment identiques; ce qui est loin de nous convenir. En effet, par rapport à l'exemple, nous avons $x = y = Jean$.

Il faudrait que la fonction Φ puisse évoluer dynamiquement et que le processus d'interprétation échoue s'il n'est pas possible de trouver un x qui vérifie $\varphi(x)$. La logique dynamique des prédicats permet de résoudre ce problème, en proposant une meilleur gestion de l'assignation de variables.

4.3 La logique des prédicats dynamiques

4.3.1 Présentation

Hilbert et Bernays ont introduit une notation pour représenter les assignations définies et indéfinies. Le descripteur ι permet d'assigner à x une valeur unique vérifiant une relation φ. Le descripteur η permet lui d'assigner à x une valeur quelconque prise dans un ensemble de valeur vérifiant une relation φ.

Une phrase telle que *"Jean voit un homme"* correspond à l'algorithme suivant:

- soit une première variable v_1 qui reçoit une valeur quelconque;
- un test vérifie que v_1 correspond à un individu qui est $Jean$;
- soit une seconde variable v_2 qui reçoit une valeur quelconque;
- un test vérifie que v_2 vérifie la relation être un homme;
- un test vérifie que v_1 voit v_2.

Ce programme peut s'écrire formellement:
$\eta v_1 : v_1 = "Jean"; \eta v_2 : homme(v_2); voit(v_1, v_2)$

Un tel programme est composé de deux types d'instructions: des tests et des assignations. Les assignations permettent de donner des valeurs aux variables. Les assignations sont de deux types soit ι ou soit η. Dans le premier cas, la valeur possible de x ne peut être que unique. Dans le second cette valeur peut être multiple.

Les tests permettent de vérifier que les valeurs données aux variables vérifient une condition. Cette condition peut être une relation ou en particulier une égalité. Elle peut également être une conjonction (notée;) comme dans l'exemple. Les autres conditions sont la négation ($\neg$) , l'implication ($\Rightarrow$). La syntaxe définira formellement la forme des programmes.

Un programme peut alors être vu comme un manipulateur d'états. Un état est un ensemble de couples (variable, valeur). L'effet d'une instruction est alors soit de modifier la valeur d'une variable, soit d'ajouter une variable, soit de retourner l'état tel quel si le test réussi, soit de retourner un état vide si le test échoue. La sémantique apportera une définition formelle de l'exécution des programmes.

4.3.2 Syntaxe

Soient C un ensemble de constantes de DPL [1]et V un ensemble de variables de DPL. Un terme est un élément de $C \cup V$. Soit un ensemble de relations de DPL.

Pour un ensemble de termes donnés et un ensemble de symboles de relation, l'ensemble des programmes DPL correspond au plus petit ensemble formé à partir de:

1. $\bot$ et $\top$ sont des programmes
2. Si t_1 et t_2, sont des termes, alors $t_1 = t_2$ et $t_1 \neq t_2$ sont des programmes DPL
3. Si R est une relation n-aire et $t_1, \ldots, t_n$ sont des termes alors $Rt_1, \ldots, t_n$ est un programme DPL
4. Si π_1 et π_2 sont des programmes DPL, alors $(\pi_1; \pi_2)$ est un programme DPL
5. Si π_1 et π_2 sont des programmes DPL, alors $(\pi_1 \Rightarrow \pi_2)$ est un programme DPL
6. Si π est un programme DPL, alors $\neg\pi$ est un programme DPL
7. Si π est un programme DPL et v une variable, alors $\eta v : \pi$ est un programme DPL
8. Si π est un programme DPL et v une variable, alors $\iota v : \pi$ est un programme DPL

Pour plus de facilité, nous ignorerons les parenthèses lorsque celles-ci sont inutiles.

4.3.3 Sémantique

Approche informelle

Comme indiqué dans la sous-section 4.3.1, un programme DPL agit comme un manipulateur d'états. Pour un état d'entrée, un programme retourne l'ensemble de tous les états possibles pour cet état d'entrée. Dans le cas d'un test, soit le test réussit et l'état de sortie est ce même état d'entrée; soit le test échoue et l'état de sortie est un état vide. Les tests sont des instructions

[1]DPL = Dynamic Predicate Logic

déterministes car ils n'ont qu'un seul état de sortie. L'instruction non déterministe est l'assignation η. En effet, chaque état de l'ensemble des états de sorties d'un programme tel que $\eta x : \pi$ diffèrent de l'état d'entrée en assignant à x une valeur vérifiant π.

Les programmes $t_1 = t_2$ et $R(t_1, \ldots t_n)$ retournent l'état initial s'il vérifie la relation, l'ensemble vide sinon.

Les programmes de la forme $\pi_1 \Rightarrow \pi_2$ retournent l'état d'entrée si tous les états de sortie B du programme π_1 sont des états d'entrée adéquats pour le programme π_2. Ce type d'instructions est très utile pour rendre compte des implications et des descriptions du langage naturel. Une phrase telle que

"Si un homme admire le roi, alors il l'acclame"

correspond au programme:

$(\eta v_1 : homme(v_1); \iota v_2 : (roi(v_2); admire(v_1, v_2))) \Rightarrow acclame(v_1, v_2)$

Les programmes de la forme $\neg\pi$ retourne l'état d'entrée s'il n'y a pas d'état de sortie de π, l'état vide sinon. Ce type d'instruction permet de rendre compte de la négation en langage naturel.

"Le directeur n'utilise pas d'ordinateur"

correspond au programme:

$\iota v_1 : (directeur(v_1); \eta v_2 : ordinateur(v_2); \neg utilise(v_1, v_2))$

Approche formelle

Vu que DPL est un langage du premier ordre, pour définir sa sémantique, nous allons utiliser la théorie des modèles comme nous l'avions fait pour la logique du premier ordre (cfr section 2.2.2).

Considérons un modèle $\mathcal{M}$. Soient M son domaine et I une fonction qui assigne à chaque constante DPL un élément de M et à chaque relation $n - aire$ un élément de M^n.

Appelons un état propre A pour $\mathcal{M}$ une assignation de variables qui fait correspondre les variables DPL à un élément de M. Un état propre A est donc une fonction de V dans M. Appelons $S_{\mathcal{M}}$ l'ensemble des états propres de $\mathcal{M}$ et notons $A[x := d]$ un état propre pour $\mathcal{M}$ qui est identique à A sauf la variable x qui est assignée à d.

A partir de $\mathcal{M}$ et de A, il est possible de définir une évaluation des termes comme suit:

- si $t \in V$ alors $V_{\mathcal{M}A}(t) = A(t)$

- si $t \in C$ alors $V_{\mathcal{M}A}(t) = I(t)$

Soit π un programme, la fonction $[\![\pi]\!] : S_{\mathcal{M}} \to \mathcal{PS}_{\mathcal{M}}$, définie par récursivité, donne la sémantique d'un programme DPL. Elle retourne pour un état d'entrée, l'ensemble des états de sortie possible. Dans la définition qui suit, nous allons utiliser A, B pour représenter des états propres quelconques. $[\![\pi]\!](A)$ désigne alors l'ensemble des états de sortie que π peut produire pour l'état d'entrée A.

1. $[\![\top]\!]_{\mathcal{M}}(A) = \{A\}$
2. $[\![\bot]\!]_{\mathcal{M}}(A) = \emptyset$
3. $[\![R(t_1 \cdots t_n)]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } \langle V_{\mathcal{M},A}(t_1), \ldots, V_{\mathcal{M},A}(t_n)\rangle \in I(R) \\ \emptyset & \text{sinon.} \end{cases}$
4. $[\![t_1 = t_2]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } V_{\mathcal{M},A}(t_1) = V_{\mathcal{M},A}(t_2) \\ \emptyset & \text{sinon.} \end{cases}$
5. $[\![t_1 \neq t_2]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } V_{\mathcal{M},A}(t_1) \neq V_{\mathcal{M},A}(t_2) \\ \emptyset & \text{sinon.} \end{cases}$
6. $[\![(\pi_1; \pi_2)]\!]_{\mathcal{M}}(A) = \bigcup\{[\![\pi_2]\!]_{\mathcal{M}}(B) \mid B \in [\![\pi_1]\!]_{\mathcal{M}}(A)\}$.
7. $[\![(\pi_1 \Rightarrow \pi_2)]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si pour tout } B \in [\![\pi_1]\!]_{\mathcal{M}}(A), \text{ on a} [\![\pi_2]\!]_{\mathcal{M}}(B) \neq \emptyset \\ \emptyset & \text{sinon.} \end{cases}$
8. $[\![(\neg\pi]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } [\![\pi]\!]_{\mathcal{M}}(A) = \emptyset \\ \emptyset & \text{sinon.} \end{cases}$
9. $[\![\eta x : \pi]\!]_{\mathcal{M}}(A) = \bigcup\{[\![\pi]\!]_{\mathcal{M}}(A[x := d]) \mid d \in \mathcal{M}\}$.
10. $[\![\iota x : \pi]\!]_{\mathcal{M}}(A) = \begin{cases} [\![\pi]\!]_{\mathcal{M}}(A[x := d]) & \textbf{pour l'unique } d \in \mathcal{M} \\ & \text{tel que } [\![\pi]\!]_{\mathcal{M}}(A[x := d]) \neq \emptyset \text{ s'il existe} \\ \emptyset & \text{sinon.} \end{cases}$

Note: Comme proposé en [van Eijck 1991], une définition équivalente de la sémantique de $\top$ est

$$\top \equiv \neg\eta v_0 : v_0 \neq v_0.$$

De même $\bot$ peut être défini comme $\neg\top$ et $(t_1 \neq t_2)$ comme $\neg(t_1 = t_2)$

4.3.4 Sémantique d'erreur

L'utilisation de l'opération d'assignation $\iota x : \pi$ impose une condition d'unicité sur les valeurs possibles de x. Cette assignation correspond intuitivement à une description définie du langage naturel. Par exemple, dans la phrase *"Jean voit l'homme"*, l'homme est clairement identifiable car il ne peut en correspondre qu'un et un seul. Si, dans le contexte Jean voit plusieurs hommes, alors la phrase contient une erreur de sémantique.

Un état d'erreur ϵ est introduit dans la semantique DPL pour traiter les cas d'erreur. Cet état d'erreur ϵ est le résultat d'un programme lorsque celui-ci contient une assignation ι d'une variable x et que l'ensemble des valeurs possibles pour x n'est pas unique.

$$[\![\iota x : \pi]\!]_{\mathcal{M}}(A) = \begin{cases} [\![\pi]\!]_{\mathcal{M}}(A[x := d]) & \textbf{pour l'unique } d \in \mathcal{M} \\ & \text{tel que } [\![\pi]\!]_{\mathcal{M}}(A[x := d]) \not\subseteq \{\epsilon\} \text{ s'il existe} \\ \{\epsilon\} & \text{sinon.} \end{cases}$$

Cette modification entraîne d'autres modifications triviales dans la sématique permettant de reporter l'erreur. La nouvelle sématique est donnée par la définition de fonction suivante où A est supposé $\neq \epsilon$

1. $[\![\pi]\!]_{\mathcal{M}}(\epsilon) = \{\epsilon\}$
2. $[\![\top]\!]_{\mathcal{M}}(A) = \{A\}$
3. $[\![\bot]\!]_{\mathcal{M}}(A) = \emptyset$
4. $[\![R(t_1 \cdots t_n)]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } \langle V_{\mathcal{M},A}(t_1), \ldots, V_{\mathcal{M},A}(t_n)\rangle \in I(R) \\ \emptyset & \text{sinon.} \end{cases}$
5. $[\![t_1 = t_2]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } V_{\mathcal{M},A}(t_1) = V_{\mathcal{M},A}(t_2) \\ \emptyset & \text{sinon.} \end{cases}$
6. $[\![t_1 \neq t_2]\!]_{\mathcal{M}}(A) = \begin{cases} \{A\} & \text{si } V_{\mathcal{M},A}(t_1) \neq V_{\mathcal{M},A}(t_2) \\ \emptyset & \text{sinon.} \end{cases}$
7. $[\![(\pi_1; \pi_2)]\!]_{\mathcal{M}}(A) = \bigcup\{[\![\pi_2]\!]_{\mathcal{M}}(B) \mid B \in [\![\pi_1]\!]_{\mathcal{M}}(A)\}$.
8. $[\![(\pi_1 \Rightarrow \pi_2)]\!]_{\mathcal{M}}(A) = \begin{cases} \{\epsilon\} & \text{s'il y a un état} B \in [\![\pi_1]\!]_{\mathcal{M}}(A) \text{avec} [\![\pi_1]\!]_{\mathcal{M}}(B) = \{\epsilon\} \\ \{A\} & \text{si pour tout } B \in [\![\pi_1]\!]_{\mathcal{M}}(A), \text{ on a} [\![\pi_2]\!]_{\mathcal{M}}(B) \not\subseteq \{\epsilon\} \\ \emptyset & \text{sinon.} \end{cases}$
9. $[\![(\neg\pi]\!]_{\mathcal{M}}(A) = \begin{cases} \{\epsilon\} & \text{si} [\![\pi]\!]_{\mathcal{M}}(A) = \{\epsilon\} \\ \{A\} & \text{si } [\![\pi]\!]_{\mathcal{M}}(A) = \emptyset \\ \emptyset & \text{sinon.} \end{cases}$

10. $[\![\eta x : \pi]\!]_{\mathcal{M}}(A) = \bigcup\{[\![\pi]\!]_{\mathcal{M}}(A[x := d]) \mid d \in \mathcal{M}\}$.

11. $[\![\iota x : \pi]\!]_{\mathcal{M}}(A) = \begin{cases} [\![\pi]\!]_{\mathcal{M}}(A[x := d]) & \textbf{pour l'unique } d \in \mathcal{M} \\ & \text{tel que } [\![\pi]\!]_{\mathcal{M}}(A[x := d]) \not\subseteq \{\epsilon\} \text{ s'il existe} \\ \{\epsilon\} & \text{sinon.} \end{cases}$

Nous dirons :

- qu'un programme produit un succès s'il produit au moins un état non vide;
- qu'un programme produit un échec s'il produit $\emptyset$;
- qu'un programme produit une erreur s'il produit ϵ.

Remarque: Il est intéressant de constater que

$$\eta v : R_1(v); R_2(v)$$

est équivalent à

$$\eta v : (R_1(v_1); R_2(v_2))$$

Par contre

$$\iota v : R_1(v); R_2(v)$$

n'est pas équivalent à

$$\iota v : (R_1(v); R_2(v))$$

En effet, dans le premier cas la condition d'unicité porte sur R_1; dans le second, par contre, la condition d'unicité doit être vérifiée par $R_1 \wedge R_2$.

4.4 Conclusion

Au cours de ce chapitre, nous avons développé un formalisme qui permet de représenter la sémantique d'une expression du langage naturel. Ce formalisme s'appelle la logique des prédicats dynamiques avec un état d'erreur. Il consiste à représenter au moyen de deux types d'opérateurs d'assignations et de tests, l'interprétation logique d'une phrase.

Pour rendre la sémantique du langage naturel accessible aux ordinateurs via cette représentation, il est nécessaire de construire un programme qui traduise de manière automatique l'arbre d'analyse d'une phrase dans sa représentation sémantique. Le chapitre 5 décrit la construction d'un tel programme.

Nous pourrons ainsi grâce au chapitre 3 construire l'arbre d'analyse d'une phrase et au chapitre 5 traduire cet arbre dans sa représentation sémantique. Le chapitre 6 expliquera comment exploiter cet interprétation pour évaluer la sémantique d'une phrase par rapport à un modèle.

Chapitre 5

Traduction en DPL

5.1 Préliminaire: Le problème des références

La plupart des langues naturelles disposent de pronoms. Un pronom fait référence à un mot rencontré précédemment et permet d'éviter la répétition de ce mot. Il n'existe pratiquement aucune règle reliant un pronom au mot. Par exemple dans la phrase *"Si un animal a des ailes alors il vole"*. *"Il"* remplace *"un animal"*. La règle pourrait être que le pronom sujet correspond au sujet de la proposition précédente. Hélas, cette règle n'est pas forcément vérifiée. Dans la phrase *"Si je tend la main à quelqu'un, alors il me la serre."* , le pronom *"il"* correspond à un complément indirect du verbe: le sujet est remplacé par le pronom *"me"* qui est un complément indirect du verbe et *"la"* remplace "la main". Le problème des références n'est pas formalisable facilement. La sémantique de la phrase joue un rôle important. Nous allons nous faciliter la tâche en utilisant des index pour signaler les références. Un numéro en exposant indiquera un référant et un numéro en indice indiquera une référence. Ainsi les exemples précédent seront notés:

Si un^1 *animal a des*^{2} *ailes alors il*$_1$ *vole.*

Si je$_1$ *tend la*^{2} *main à quelqu'un*^{3}, *il*$_3$ *me*$_1$ *la*$_2$ *serre.*

Nous supposons évidement que l'indice 1 fait référence au narateur.

Les pronoms ne sont pas les seuls mots à utiliser ces références. Certains déterminants ou adjectifs les utilisent. Par exemple pour *"Un homme entre. Il s'assied. Un autre homme entre"*, l'adjectif *"autre"* fait référence à l'homme précédent. Cette phrase correspond à:

"Un^1 *homme entre. Il*$_1$ *s'assied. Un*^{2} *autre*$_1$ *homme entre"*

De même, dans l'exemple *"Un homme entre. Cet homme s'assied"*, le déterminant *"cet"* fait référence à l'homme précédent. Les indices sont donc placés comme suit:

"Un^1 *homme entre. Cet*$^2_1$ *homme s'assied"*

5.2 Approche intuitive

L'objectif de cette section est de décrire de manière intuitive un procédé pour traduire automatiquement une phrase en une représentation DPL. Pour ce faire, partons d'une phrase simple telle que *"Jean^1 voit un^2 homme"* dont la traduction attendue est

$$\eta v_1 : v_1 = \text{"}Jean\text{"}; \eta v_2 : homme(v_2); voit(v_1, v_2).$$

Traduisons cette phrase par raffinements successifs.

étape 1 Supposons que nous pouvons traduire séparément le sujet $Jean^1$ en $\eta v_1 : v_1 = \text{"}Jean\text{"}; X(v_1)$ et le groupe verbal *voit un^2 homme* en $\eta v_2 : homme(v_2); voit(X, v_2)$.

Nous pouvons représenter la dernière expression par une fonction $Voit - Un - Homme$ qui prend une variable v comme argument et qui produit un programme. La définition de cette fonction est:

$$Voit - Un^j - Homme(v) = \eta v_j : homme(v_j); voit(v, v_j).$$

De même, l'expression

$$\eta v_1 : v_1 = \text{"}Jean\text{"}; X(v_1).$$

peut également être représentée par une fonction $Jean^i$ qui reçoit une fonction comme argument et produit un programme. Cette fonction est définie de la manière suivant:

$$Jean^i(\pi) = \eta v_i : v_i = \text{"}Jean\text{"}; \pi(v_i)$$

La traduction de la phrase *"Jean^1 voit un^2 homme"* est obtenue par la composition des fonctions $Jean^1(\pi)$ et $Voit - Un^2 - Homme(v)$. En effet,

$$Jean^1(Voit - Un^2 - Homme(v))$$

$$= \eta v_1 : v_1 = \text{"}Jean\text{"}; Voit - Un^2 - Homme(v_i)$$

$$= \eta v_1 : v_1 = \text{"}Jean\text{"}; \eta v_2 : homme(v_2); voit(v_1, v_2)$$

étape 2 Voyons maintenant comment traduire séparément le sujet $Jean^1$ et le groupe verbal *voit un^2 homme*.

La traduction de $Jean^1$ est évidente. En effet, nous pouvons définir un lexique qui donne la fonction associée à chaque mot.

Par contre la traduction de *voit un^2 homme* est moins aisée. Supposons que nous pouvons traduire séparément *voit* et *un^{2}homme*.

La traduction attendue de *un^{2}homme* est

$$\eta v_2 : homme(v_2); X(v_2).$$

Cette expression est similaire à la traduction de *Jean*. Nous allons la représenter par une fonction qui prend une fonction comme argument.

$$Un^j - Homme(\pi) = \eta v_j : homme(v_j); \pi(v_j)$$

La traduction de *voit* est moins évidente. Soit $Voit$ une fonction qui prend une fonction comme argument définie comme suit:

$$Voit(\pi) = \pi(Voit'(v))$$

où $Voit'(v)$ est une fonction intermédiaire qui prend une variable comme argument. La définition de $Voit'(v)$ est

$$Voit'(v) = Voit''(v').$$

$Voit''(v')$ est une fonction qui prend une variable comme argument définie comme suit:

$$Voit''(v') = voit(v', v).$$

La traduction de *voit un*^{2} *homme* est obtenue par la combinaison des fonctions $Voit$ et $Un^2 - Homme(v)$ en effet,

$$\begin{aligned} & Voit(Un^2 - Homme(\pi)) \\ & = Un^2 - Homme(Voit'(v)) \\ & = \eta v_2 : homme(v_2); Voit'(v_2) \\ & = \eta v_2 : homme(v_2); Voit''(v') \\ & = Voit - Un^2 - Homme(v') \\ & = \eta v_2 : homme(v_2); voit(v', v_2) \end{aligned}$$

étape 3 La traduction de *voit* peut être donnée par un lexique. Par contre, la traduction de *un*^{2}*homme* doit être définie.

La traduction attendue de *homme* est

$$homme(x).$$

Cette expression peut être représentée par la fonction suivante:

$$Homme(v) = homme(v)$$

La traduction attendue de *un*^{2} est

$$\eta v_2; \pi_1(v_2); \pi_2(v_2).$$

Cette expression peut être représentée par la fonction Un^j qui a une fonction comme argument et qui produit une fonction. Elle est définie de la manière suivante:

$$Un^j(\pi_1) = Un'^j(\pi_2)$$

où la fonction Un'^j a comme argument un programme et est définie comme suit:

$$Un'^j(\pi_2) = \eta v_j : \pi_1(v_j); \pi_2(v_j).$$

La traduction de $un^2 homme$ correspond à la combinaison des fonctions

$$Un^2(Homme(v)).$$

En effet,

$$\begin{aligned} & Un^2(Homme(v)) \\ & = Un'^2(\pi_2) \\ & = \eta v_2 : Homme(v_2); \pi_2(v_2) \\ & = \eta v_2 : homme(v_2); \pi_2(v_2) \\ & = Un^2 - Homme(\pi_2) \\ & \eta v_2 : homme(v_2); \pi_2(v_2) \end{aligned}$$

- En résumé, la traduction de la phrase *"Jean^1 voit un^2 homme"* est obtenue par la combinaison des fonctions $Jean^1(Voit(Un^2(Homme(v))))$ En effet,

$$\begin{aligned} & Jean^1(Voit(Un^2(Homme(v)))) \\ & = Jean^1(Voit(Un^2 - Homme(\pi))) \\ & = Jean^1(Voit - Un^2 - Homme(v)) \\ & = \eta v_1 : v_1 = "Jean"; \eta v_2 : homme(v_2); voit(v_1, v_2) \end{aligned}$$

Le principe de traduction consiste, en gros, à associer à chaque mot une fonction. La combinaison des fonctions permet aprs simplification de produire un programme DPL.

Un des inconvénients de l'approche ci-dessus est d'imposer un nom pour chaque fonction. De plus, les concepts de composition de fonctions, de fonctions comme argument, de fonctions comme résultats de fonctions ne sont pas manipulables facilement sans une définition rigoureuse. Le λ-calcul permet de remédier à ces inconvénients.

5.3 Le λ-calcul

Nous allons nous contenter de rappeler les concepts fondamentaux qui seront utiles par la suite.

Définition Si M est une expression prenant leur valeur dans un domaine D, alors

$$\lambda\, x : D' \,.\, M$$

représente la fonction $f : D' \rightarrow D$ telle que $f(x)$, pour n'importe quelle valeur de $x \in D'$, peut être obtenue en évaluant M avec cette valeur de x. Nous noterons souvent $\lambda x.M$ si D' peut être déduit du contexte.

Exemple La fonction f(x) = x+1 correspond à la λ-expression suivante:

$$\lambda x.x + 1$$

Ainsi $(\lambda x.x + 1)3 = 3 + 1 = 4$

Définition Les λ-expressions **bien formées** sont définies par les règles suivantes:

1. si v est une variable, alors v est une λ-expression bien formée
2. si M et N sont des λ-expressions bien formées, alors MN est une λ-expression bien formée (appelée **application**)
3. si v est une variable et M une λ-expression bien formée, alors $\lambda v.M$ est une λ-expression bien formée (appelée **abstraction**)

Les définitions données ci-dessous permettent de manipuler les λ-expressions. Les règles de réduction énoncent les "règles d'exécution" d'une λ-expression. La notion de substitution est utilisée par les règles de réduction et la notion d'occurrence libre est utilisée par les règles de substitutions.

Définition Les **Occurences libres** dans une λ-expression sont définies comme suit:

1. x apparaît librement dans x
2. x apparaît librement dans XY s'il apparaît librement dans X ou dans Y
3. x apparaît librement dans $\lambda y.X$ si x et y sont des variables différentes et si x apparaît librement dans X.

Définition La **Substitution** d'une variable x par une expression M dans une expression X est notée $[x/M]X$ et définie de la manière suivante:

1. si X est une variable
   (a) si $X = x$ alors $X' = M$
   (b) si $X \neq x$ alors $X' = X$
2. si X est une application YZ: $X' = ([M/x]Y)([M/x]Z)$
3. si X est une abstraction $\lambda y.Y$:

(a) si $y = x$ alors $X' = X$

(b) si $y \neq x$ alors

i. si x n'apparaît pas librement dans Y ou si y n'apparaît pas librement dans M alors $X' = \lambda y.[M/x]Y$

ii. si x apparaît librement dans Y et si y apparaît librement dans M alors $X' = \lambda z.[M/x]([z/y]Y)$ où z est la première variable dans la séquence des variables telle que z n'apparaît pas librement dans M ou Y

Définition Les **Règles de réduction** permettent de simplifier une λ-expression.

- α si y n'est pas libre dans X, alors $\lambda x.X$ red_α $\lambda y.[y/x]X$
- β $(\lambda x.M)N$ red_β $[N/x]M$
- η si x n'est pas libre dans M, alors $\lambda x.Mx$ red_η M

5.4 Les principes de la traduction automatique

Dans la section 5.2, à chaque mot était associée une fonction. La composition de ces fonctions conduisait à un programme DPL après simplification.

Cette notation n'était pas très pratique, car elle demandait une nomination des fonctions. De plus ni la composition ni la simplification n'était parfaitement définie.

Le λ-calcul offre une meilleure approche pour définir les fonctions. La composition est définie par l'application de deux expressions et la simplification par la réduction d'une expression.

Pour traduire une phrase, l'idée principale consiste donc à associer une λ-expression à chaque mot, en fonction de sa classe grammaticale. La traduction d'une phrase est obtenue par la réduction de l'application de chaque λ-expression associée au mot de la phrase.

Jusqu'à présent, les applications de fonction se font de la gauche vers la droite. En fait, il n'en est pas toujours de même et l'ordre d'application dépend de la catégorie des sous-arbres. La traduction d'un arbre A est définie comme suit:

- L'arbre A contient une information de type 2 (pour rappel, un mot présent dans le lexique et sa catégorie; cfr chapitre 3) et a donc ses sous-arbres vides. La traduction de l'arbre se trouve dans le dictionnaire.
- L'arbre A est de catégorie a, la catégorie du sous-arbre gauche SAG est a/b et la catégorie du sous-arbre droit SAD est b alors la traduction est obtenue en réduisant l'application de la traduction du SAG suivi du SAD.

- L'arbre A est de catégorie a, la catégorie du sous-arbre gauche SAG est b et la catégorie du sous-arbre droit SAD est $b\backslash a$ alors la traduction est obtenue en réduisant l'application de la traduction du SAD suivi du SAG.

5.4.1 Exemple: Jean^1 voit un^2 homme

Le tableau suivant donne les expressions associées à chaque mot:

Mot	λ-expression
homme	$\lambda X.homme(X)$
un^i	$\lambda X.\lambda Y(\eta v_i : X v_i; Y v_i)$
voit	$\lambda X.\lambda Y.(X(\lambda Z.voit(Y,Z))$
Jean^i	$\lambda X.(\eta v_i : v_i = Jean; X v_i)$

Pour rappel, l'arbre d'analyse de *Jean*^{1} *voit un*^{2} *homme* est:

P

GN(Sg,3pers): Jean^1

GV(Sg,3pers):

VT(Sg,3pers): voit^2

GN(Sg,*):

DET(Sg): un^2

N(Sg): homme

Les différentes étapes de la traduction sont donc:

1. La traduction de *Jean*^{1} donne:

$$\lambda X.(\eta v_1 : v_1 = Jean; X v_1)$$

2. La traduction de *voit un*^{2} *homme* donne:

   (a) La traduction de *voit*:

$$\lambda X.\lambda Y.(X(\lambda Z.voit(Y,Z))$$

   (b) La traduction de *un*^{2} *homme* donne:

i. La traduction de un^2 donne:

$$\lambda X.\lambda Y(\eta v_2 : Xv_2; Yv_2)$$

ii. La traduction de *homme* donne:

$$\lambda X.homme(X)$$

iii. la réduction de

$$(\lambda X.\lambda Y(\eta v_2 : Xv_2; Yv_2))(\lambda X.homme(X))$$

donne:

- $\lambda Y(\eta v_2 : (\lambda X.homme(X))v_2; Yv_2)$
- $\lambda Y(\eta v_2 : homme(v_2); Yv_2)$

(c) la réduction de

$$(\lambda X.\lambda Y.(X(\lambda Z.voit(Y,Z)))(\lambda Y(\eta v_2 : homme(v_2); Yv_2))$$

donne:

- $\lambda Y.((\lambda W(\eta v_2 : homme(v_2); Wv_2))(\lambda Z.voit(Y,Z))$
- $\lambda Y.((\eta v_2 : homme(v_2); (\lambda Z.voit(Y,Z))v_2))$
- $\lambda Y.(\eta v_2 : homme(v_2); voit(Y,v_2))$

3. La réduction de

$$(\lambda X.(\eta v_1 : v_1 = Jean; Xv_1))(\lambda Y.(\eta v_2 : homme(v_2); voit(Y,v_2)))$$

donne :

- $\eta v_1 : v_1 = Jean;\ (\lambda Y.(\eta v_2 : homme(v_2); voit(Y,v_2)))v_1)$
- $\eta v_1 : v_1 = Jean; \eta v_2 : homme(v_2); voit(v_1,v_2)$

5.4.2 Le dictionnaire de λ-expression

La table suivante donne les λ-expressions pour les principaux mots en fonction de leur catégorie pour la dernière version du fragment du Français [1]. A partir d'une définition d'un mot, il est souvent facile de déduire la définition des autres mots de cette catégorie. Par exemple, à partir de la définition de *homme* donnée précédemment, il est possible de déduire la définition de la plupart des noms communs. Seul, le nom de la relation est modifié.

[1] La version du Fragment importe peu, seule la deuxième colonne est modifiée

Mot	Categorie	Expression
un^i	DET(Sg,Masc)	$\lambda X \lambda Y.\eta v_i : X v_i; Y v_i$
une^i	DET(Sg,Fem)	$\lambda X \lambda Y.\eta v_i : X v_i; Y v_i$
le^i	DET(Sg,Masc)	$\lambda X \lambda Y.\iota v_i : X v_i; Y v_i$
homme	N(Sg,Masc)	$\lambda X.homme(X)$
femme	N(Sg,Fem)	$\lambda X.femme(X)$
voit	VT(Sg,3pers)	$\lambda X \lambda Y.X(\lambda Z voir(Y,Z))$
admire	VT(Sg,3pers)	$\lambda X \lambda Y.X(\lambda Z admirer(Y,Z))$
Jean^i	GN(Sg,Masc,3pers)	$\lambda X.\eta v_i : v_i = Jean; X v_i$
Marie^i	GN(Sg,Fem,3pers)	$\lambda X.\eta v_i : v_i = Marie; X v_i$
il$_i$	GN(Sg,Masc,3pers)	$\lambda X.X v_i$
le$_i$	PACC	$\lambda X.X(\lambda Y.Y v_i)$
entre	GV(Sg,3pers)	$\lambda X.entrer(X)$
ne	NE	$\lambda X.\neg X$
pas	PAS	$\lambda X.X$
alors	ALORS	$\lambda X.X$
si	SI	$\lambda X \lambda Y.X \Rightarrow Y$
avec	AVEC	$\lambda X \lambda Y \lambda Z.YZ; X(\lambda W.avec(Z,W))$

5.5 Implémentation

Dans cette section, nous allons nous intéresser en détail sur la conception d'un programme logique qui permet de réaliser automatiquement la traduction d'une phrase donnée dans sa représentation DPL.

5.5.1 Les représentations

Avant de définir les prédicats du programme, il est nécessaire de représenter les données qui seront manipulées. Dans le chapitre 3, nous avions déjà défini une représentation pour les catégories et les phrases.

Suite à l'introduction d'indice et d'exposant dans certains mots, il est nécessaire de revoir la représentation des mots.

Les mots. Un mot peut avoir aucun, un ou deux numéros d'index. La fonction U avec comme argument un entier représente un exposant et la fonction D à l'inverse représente un indice.

La fonction S construit un mot avec aucun numéro d'index. La fonction C construit un mot avec un seul numéro d'index et la fonction T construit un mot avec deux numéros d'index.

Les modifications sur le programme initial sont peu nombreuses. Une phrase est constituée

d'une suite de mots. Le dictionnaire tient compte de la nouvelle notation.

La notation est beaucoup plus lourde que précédemment. La représentation de la phrase *Jean*^{1} *voit un*^{2} *homme* est

```
[C("jean",U(1)), S("voit"), C("un",U(2)), S("homme")].
```

Les programmes DPL. Pour représenter un programme DPL nous allons utiliser des fonctions. Avant de représenter un programme, il faut représenter les termes, les variables et les constantes.

La fonction V avec un entier i comme argument représente la variable DPL numéro i. La fonction Var avec une variable comme argument représente une terme DPL de type variable. La fonction Cst avec une chaîne de caractères comme argument représente un terme DPL de type constante.

Une relation est représentée par la fonction Rel avec une chaîne de caractères correspondant aux arguments de la relation et une liste de termes comme arguments.

Les fonctions suivantes permettent de construire un programme DPL:

1. La fonction PR avec une relation comme argument représente un programme de type relation.
2. La fonction Pequal avec deux termes comme arguments représente un programme de type égalité de termes.
3. La fonction Pnot avec un programme comme argument représente la négation d'un programme.
4. La fonction Papplic avec deux programmes comme arguments représente la séquence de deux programmes.
5. La fonction Pimplic avec deux programmes comme arguments représente l'implication de deux programmes.
6. La fonction Peta avec une variable et un programme comme arguments représente l'assignation η d'une variable dans un programme.
7. La fonction Piota avec une variable et un programme comme arguments représente l'assignation ι d'une variable dans un programme.
8. La fonction LE avec une λ-expression comme argument permet d'introduire une λ-expression dans un programme au cours de la traduction.

Les λ-expressions Une variable λ est représentée par la fonction Vl avec comme argument un entier.

Une λ-expression est représentée par les fonctions suivantes:

1. La fonction VE avec comme argument une variable (une variable peut être une variable DPL ou une variable λ).
2. La fonction A avec comme argument deux expressions représente l'application de deux expressions.
3. La fonction L avec comme argument un entier et une expression permet de représenter l'abstraction d'une variable dans une expression.
4. La fonction P avec comme argument un programme permet d'introduire un programme dans une expression au cours de la traduction.

5.5.2 La réduction

Le λ-calcul que nous utilisons est un peu plus complexe que le λ-calcul traditionnel car nous y introduisons la notion de programme.

Une λ-expression peut-être soit:

- une variable;
- une application;
- une abstraction;
- un programme (qui peut lui même contenir une λ-expression).

Pour réaliser le réducteur de λ-expressions, il est nécessaire de définir des prédicats correspondant aux concepts d'occurrence libre, de substitution et de réduction.

Occurence libre. La notion d'occurrence libre doit être étendue à la notion d'occurrence libre d'une variable dans un programme. Intuitivement une variable apparaît dans un programme, si elle est présente au moins une fois dans ce programme. La définition informelle consiste en la règle suivante: si une variable est présente au moins une fois dans un programme, elle apparaît librement dans ce programme.

Le prédicat `Occfree` permet de définir cette notion.

- x apparaît librement dans x:

```
Occfree (x,VE(Vl(x))).
```

- x apparaît librement dans Y Z si x apparaît librement dans Y **ou** si x apparaît librement dans Z:

```
Occfree (x,A(y,_)) ← Occfree (x,y).
Occfree (x,A(_,z)) ← Occfree (x,z).
```

- x apparaît librement dans $\lambda y.Z$, si x $\neq$ y **et** si x apparaît librement dans Z:

```
Occfree (x,L(y,z)) ← x ≠ y ∧ Occfree (x,z).
```

- x apparaît librement dans R(...,x,...).

```
Occfree (x,P(PR(Rel(_,[Var(Vl(x))])))).
Occfree (x,P(PR(Rel(_,[Var(Vl(x))|_])))).
Occfree (x,P(PR(Rel(n,[_|t])))) ← Occfree (x,P(PR(Rel(n,t)))).
```

- x apparaît librement dans x=y ou dans y=x:

```
Occfree (x,P(Pequal(Var(Vl(x)),_))).
Occfree (x,P(Pequal(Var(_),Var(Vl(x))))).
```

- x apparaît librement dans $\neg\pi$ si x apparaît librement dans π:

```
Occfree (x,P(Pnot(p))) ← Occfree (x,P(p)).
```

- x apparaît librement dans $\pi_1 \Rightarrow \pi_2$ si x apparaît librement dans π_1 ou dans π_2:

```
Occfree (x,P(Pimplic(p1,_))) ← Occfree (x,P(p1)).
Occfree (x,P(Pimplic(_,p2))) ← Occfree (x,P(p2)).
```

- x apparaît librement dans $\pi_1;\pi_2$ si x apparaît librement dans π_1 ou dans π_2:

```
Occfree (x,P(Papplic(p1,_))) ← Occfree (x,P(p1)).
Occfree (x,P(Papplic(_,p2))) ← Occfree (x,P(p2)).
```

- x apparaît librement dans $\iota y.\pi$ si x = y ou si x apparaît librement dans π:

```
Occfree (x,P(Piota(_,p))) ← Occfree (x,P(p)).
Occfree (x,P(Piota(Vl(x),_))).
```

- x apparaît librement dans $\eta y.\pi$ si x = y ou si x apparaît librement dans π:

```
Occfree (x,P(Peta(Vl(x),_))).
Occfree (x,P(Peta(_,p))) ← Occfree (x,P(p)).
```

- x apparaît librement dans un programme qui contient une λ-expression si x apparaît librement dans cette λ-expression:

```
Occfree (x,P(LE(e))) ← Occfree (x,e).
```

La substitution. Ici aussi, il est nécessaire de modifier la notion vue précédemment pour y inclure la substitution d'une variable par une expression dans un programme. Informellement, cela consiste à remplacer chaque occurrence de cette variable par l'expression en question. (Par convention, étant donné qu'il n'est pas possible d'utiliser des majuscules et des minuscules comme variable, px représente "petit x" et gm "grand m".)

- $[x/M]x = M$.

```
Subst (px,gm,VE(Vl(px)),gm).
```

- $[x/M]X = X$(si $X \neq x$).

```
Subst (px,_,VE(Vl(gxp)),VE(Vl(gxp)))
            ← px ≠ gxp.
```

- $[x/M]YZ = [x/M]Y\ [x/M]Z$.

```
Subst (px,gm,A(gy,gz),A(gxp1,gxp2))
            ← Subst (px,gm,gy,gxp1) ∧
            Subst (px,gm,gz,gxp2).
```

- $[x/M](\lambda x.X) = \lambda x.X$.

```
Subst (px,_,L (px,gxp),L(px,gxp)).
```

- $[x/M](\lambda y.Y) = \lambda y.[x/M]Y$.
(si $x \neq y$ et si x n'apparaît pas librement dans X).

```
Subst (px,gm,L (py,gy),L(py,gxp))
            ← px ≠ py ∧
            ¬ Occfree(px,gy) ∧
            Subst (px,gm,gy,gxp).
```

- $[x/M]\lambda y.Y = \lambda y.[x/M]Y$.
(si x $\neq$ y et si y n'apparaît pas librement dans M.)

```
Subst (px,gm,L (py,gy),L (py,gxp))
            ← px ≠ py ∧
            ¬ Occfree(py,gm) ∧
            Subst (px,gm,gy,gxp).
```

- $[x/M]\lambda y.Y = \lambda z.[x/M]([y/z]Y)$.
si x $\neq$ y et si x apparaît librement dans Y et si y apparaît librement dans M, z est la variable suivante telle que z n'apparaît pas librement dans Y M.

```
Subst (px,gm,L (py,gy),gxp)
            Occfree (px,gy) ∧
            Occfree(py,gm) ∧
```

```
        Nextnotfree (px,A(gy,gm),pz) ∧
        gxp = L (pz,c) ∧
        Subst (px,gm,b,c) ∧
        Subst (py,VE(Vl(pz)),gy,b).
```

- $[x/M]v = v$ (v est une variable DPL).

```
Subst (_,_,VE(V(gxp)),VE(V(gxp))).
```

- $[x/M]\bot = \bot$.

```
Subst (_,_,P(PBottom),P(PBottom)).
```

- $[x/M]\top = \top$.

```
Subst (_,_,P(PTop),P(PTop)).
```

- $[x/M]\neg\pi = \neg[x/M]\pi$.

```
Subst (px,gm,P(Pnot(p)), P(Pnot(pp)))
        ← Subst (px,gm,P(p),P(pp)).
```

- $[x/M]\pi_1;\pi_2 = [x/M]\pi_1;[x/M]\pi_2$

```
Subst (px,gm,P(Papplic(p1,p2)), P(Papplic(p1p,p2p)))
        ← Subst (px,gm,P(p1),P(p1p))
        ∧ Subst (px,gm,P(p2),P(p2p)).
```

- $[x/M]\pi_1 \Rightarrow \pi_2 = [x/M]\pi_1 \Rightarrow [x/M]\pi_2$

```
Subst (px,gm,P(Pimplic(p1,p2)), P(Pimplic(p1p,p2p)))
        ← Subst (px,gm,P(p1),P(p1p))
        ∧ Subst (px,gm,P(p2),P(p2p)).
```

- $[x/M]R(\ldots,x,\ldots) = R(\ldots,M,\ldots)$

```
Subst (px,gm,P(PR(Rel(m,l))),P(PR(Rel(m,lp))))
        ← LSubst (px,gm,l,lp).
```

- $[x/M](x = y) = (M = y)$ ou $[x/M](y = x) = (y = M)$
  (si M est une expression de type variable)

```
Subst (px,VE(gm),P(Pequal(Var(Vl(px)),t2)), P(Pequal(Var(gm),t2))).
Subst (px,VE(gm),P(Pequal(t1,Var(Vl(px)))), P(Pequal(t1,Var(gm)))).
Subst (px,_,P(Pequal(t1,t2)), P(Pequal(t1,t2)))
        ← ¬TermVle(t1,px)
        ∧ ¬TermVle(t2,px).
Subst (_,_,P(Pequal(t1,t2)),P(Pequal(t1,t2)))
        ← ¬TermVl(t1)
        ∧ ¬TermVl(t2).
```

- $[x/M](\eta x : \pi) = \eta M : [x/M]\pi$
  si M est une variable.
  ou $[x/M](\eta y : \pi) = \eta y : [x/M]\pi$.
  si x $\neq$ y.

```
Subst (px,VE(gm),P(Peta(Vl(px),p)),P(Peta(gm,pp)))
        ← Subst (px,VE(gm),P(p),P(pp)).
Subst (px,gm,P(Peta(Vl(py),p)),P(Peta(Vl(py),pp)))
        ← py ≠ px ∧ Subst (px,gm,P(p),P(pp)).
Subst (px,gm,P(Peta(V(py),p)),P(Peta(V(py),pp)))
        ← Subst (px,gm,P(p),P(pp)).
```

- $[x/M](\iota x : \pi) = \iota M : [x/M]\pi$
  si M est une variable.
  ou $[x/M](\eta y : \pi) = \iota y : [x/M]\pi$
  si x $\neq$ y

```
Subst (px,VE(gm),P(Piota(Vl(px),p)),P(Piota(gm,pp)))
        ← Subst (px,VE(gm),P(p),P(pp)).
Subst (px,gm,P(Piota(Vl(py),p)),P(Piota(Vl(py),pp)))
        ← py ≠ px ∧ Subst (px,gm,P(p),P(pp)).
Subst (px,gm,P(Piota(V(py),p)),P(Piota(V(py),pp)))
        ← Subst (px,gm,P(p),P(pp)).
```

- Si un programme contient une expression X, alors il faut effectuer la substitution de M par x dans X.

```
Subst (px,gm,P(LE(gx)),P(LE(gxp)))
        ← Subst (px,gm,gx,gxp).
```

Ce prédicat utilise les prédicats secondaires:
`Lsubst` , `TermVLe` , `TermVL` , `Nextnotfree`

- Le prédicat `Lsubst` définit la substitution d'une variable par une expression dans une liste de variables:

```
LSubst (_,_,[],[]).
LSubst (px,VE(gm),[Var(Vl(px))|t],[Var(gm)|tp])
        ← LSubst (px,VE(gm),t,tp).
LSubst (px,gm,[Var(Vl(py))|t],[Var(Vl(py))|tp])
        ← py ≠ px ∧ LSubst (px,gm,t,tp).
LSubst (px,gm,[t1|t],[t1|tp])
        ← ¬TermVl(t1) ∧ LSubst (px,gm,t,tp).
```

- Le prédicat `TermVLe` indique le numéro de la variable donnée comme premier argument:
  `TermVle (Var(Vl(x)),x)` .
- Le prédicat `TermVl` vérifie qu'une variable est une variable λ et non une variable DPL:
  `TermVl (Var(Vl(_)))`
- Le prédicat `Nextnotfree` donne le numéro suivant de la variable λ qui n'est pas libre dans une expression.

```
Nextnotfree (i,e,i)
            ← ¬Occfree (i,e).
Nextnotfree (i,e,j)
            ← Occfree (i,e) ∧
            k = i + 1 ∧
            Nextnotfree (k,e,j).
```

Les règles de réduction. Les règles de réduction permettent de simplifier une λ-expression complexe. L'application successive de ces règles conduit à un programme DPL.

- Une première règle correspond à la réduction β. Une application entre une abstraction $(\lambda x.M)$ et une expression N se réduit en substituant x par N dans M:

$$(\lambda x.M)N \text{ red } [x/N]M$$

```
Redl (A(L(px,gm),gn),res)
            ← Subst (px,gn,gm,res).
```

- Une deuxième règle correspond à la réduction γ. L'expression $\lambda x.Mx$ se réduit en M si x n'apparaît pas librement dans M:

$$\lambda x.Mx \text{ red } M(\text{si } x \text{ n'apparaît pas librement dans } M)$$

```
Redl (L(px,A(gm,VE(Vl(px)))),gm)
            ← ¬Occfree (px,gm).
```

Les règles suivantes ne correspondent plus aux règles théoriques mais sont liées à la convention de représentation des expressions par des fonctions.

- Cette règle met en oeuvre l'association à gauche. L'expression $(\lambda X.Mx)(G_1)(G_2)$ se réduit en substituant uniquement G_1 à x dans M. Cette règle est placée en tête des clauses définissant la réduction lorsque celles-ci sont choisies de haut en bas. (comme en Prolog et Gödel)

$$(\lambda x.M)G_1G_2 \text{ red } [x/G_1]M\ G_2$$

```
Redl (A(L(px,gm),A(g1,g2)), A(res,g2))
            ← Subst (px,g1,gm,res).
```

- Les règles suivantes permettent de réduire l'intérieur d'une expression. En effet, une expression comme $((\lambda x.Nx)a)M$ n'est pas réductible par les règles habituelles. Or cette expression contient une sous expression qui peut être réduite.

$$e_1 e_2 \text{ red } e'_1 e_2 \text{si } e_1 \text{ red } e'_1$$

```
Redl (A(e1,e2), A(e1p,e2))
        ← Redl (e1,e1p).
```

$$e_1 e_2 \text{ red } e_1 e'_2 \text{si } e_2 \text{ red } e'_2$$

```
Redl (A(e1,e2), A(e1,e2p))
        ← Redl (e2,e2p).
```

- Dans le cas $\lambda x.((\lambda y, Ny)a)$ l'abstraction sur x ne peut être réduite. Par contre, l'abstraction sur y peut l'être.

$$\lambda x.l \text{ red } \lambda x.l' \text{si } l \text{ red } l'$$

```
Redl (L(x,l),L(x,lp))
        ← Redl (l,lp).
```

- Les règles suivantes permettent de réduire une expression de type programme. Un programme se réduit en réduisant les expressions qu'il contient.

```
Redl (P(Pnot(p)),P(Pnot(pp)))
        ← Redl(P(p),P(pp)).
Redl (P(Papplic(p1,p2)),P(Papplic(p1p,p2)))
        ← Redl(P(p1),P(p1p)).
Redl (P(Pimplic(p1,p2)),P(Pimplic(p1p,p2)))
        ← Redl(P(p1),P(p1p)).
Redl (P(Papplic(p1,p2)),P(Papplic(p1,p2p)))
        ← Redl(P(p2),P(p2p)).
Redl (P(Pimplic(p1,p2)),P(Pimplic(p1,p2p)))
        ← Redl(P(p2),P(p2p)).
Redl (P(Peta(v,p)),P(Peta(v,pp)))
        ← Redl (P(p),P(pp)).
Redl (P(Piota(v,p)),P(Piota(v,pp)))
        ← Redl (P(p),P(pp)).
```

- Si un programme contient une λ-expression, celle-ci est réduite récursivement. Il s'agit en fait du cas de base de la réduction d'un programme.

```
Redl (P(LE(e)),P(LE(ep))) ← Redl (e,ep).
```

- La règle suivante permet de simplifier les notations. En effet, un programme qui contient une expression de type programme est un programme.

```
Redl (P(LE(P(x))),P(x)).
```

- Cette règle permet de résoudre le cas où une abstraction se trouve à l'extrême droite d'une application qui est imbriquée dans une application.

$$c(\lambda a.b)m \text{ red } cm' \text{si } (\lambda a.b)m \text{ red } m'$$

```
Redl (A(A(c,L(a,b)),m),A(c,mp))
          ← Redl(A(L(a,b),m),mp).
```

- Cette règle constitue un cas plus général de la règle précédente. Plusieurs applications peuvent être imbriquées.

$$cabm \text{ red } cam' \text{si } bm \text{ red } m'$$

```
Redl (A(A(c,A(a,b)),m),A(c,mp))
          ← Redl(A(A(a,b),m),mp).
```

- Les prédicats suivants permettent de réduire une expression située à l'extrême droite d'un programme.

```
Redl (A(P(Papplic(a,LE(x))),n),P(Papplic(a,LE(res))))
          ← Redl (A(x,n),res).
Redl (A(P(Papplic(a,p)),n), P(Papplic(a,LE(res))))
          ← Redl (A(P(p),n),res).
Redl (A(P(Pimplic(a,LE(x))),n),P(Pimplic(a,LE(res))))
          ← Redl (A(x,n),res).
Redl (A(P(Pimplic(a,p)),n), P(Pimplic(a,LE(res))))
          <- Redl (A(P(p),n),res).
Redl (A(P(Pnot(LE(x))),n),P(Pnot(LE(res))))
          ← Redl (A(x,n),res).
Redl (A(P(Pnot(p)),n), P(Pnot(LE(res))))
          ← Redl (A(P(p),n),res).
Redl (A(P(Piota(v,LE(x))),n),P(Piota(v,LE(res))))
          ← Redl (A(x,n),res).
Redl (A(P(Piota(v,p)),n), P(Piota(v,LE(res))))
          ← Redl (A(P(p),n),res).
Redl (A(P(Peta(v,LE(x))),n),P(Peta(v,LE(res))))
          ← Redl (A(x,n),res).
Redl (A(P(Peta(v,p)),n), P(Peta(v,LE(res))))
          ← Redl (A(P(p),n),res).
```

La réduction canonique La réduction canonique d'une expression consiste à appliquer les règles de réduction jusqu'à ce qu'il n'y en ait plus une seule applicable.

```
Canred (exp,canexp)
        ← Redl (exp,x)
        ∧ Canred (x,canexp).
Canred (exp,exp)
        ← ¬Redl (exp,_).
```

5.5.3 La traduction

Pour traduire un arbre, nous allons définir le prédicat `Trad` basé sur la définition donnée dans la section 5.4.

La traduction d'un arbre qui contient une information de type 2 (pour rappel, une catégorie et un mot du dictionnaire), se trouve dans le dictionnaire.

```
Trad(A(I2(b,a),vide,vide),l) ← Dicl(a,b,l)
```

La traduction d'un arbre qui contient une information de type 1 (pour rappel, uniquement une catégorie), repose sur deux cas:

- Soit le SAG est de catégorie a/b et le SAD de catégorie b la traduction est obtenue de la manière suivante:
  1. Traduire récursivement le SAG en $TSAG$
  2. De même, le SAD est traduit en $TSAD$
  3. Reduire l'application de $TSAG$ et de $TSAD$

```
Trad (A(I1(ca),A(I1(csag),sag1,sad1),
A(I1(csad),sag2,sad2)),l)
        ← Syncat(ca,a) ∧
        Syncat(csag,a/b) ∧
        Syncat(csad,b) ∧
        Trad (A(I1(csag),sag1,sad1),la) ∧
        Trad (A(I1(csad),sag2,sad2),lb) ∧
        Canred (A(la,lb),l).
```

```
Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
A(I2(csad,nsad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,a/b) ∧
          Syncat(csad,b) ∧
          Trad (A(I2(csag,nsag),sag1,sad1),la) ∧
          Trad (A(I2(csad,nsad),sag2,sad2),lb) ∧
          Canred (A(la,lb),l).
Trad (A(I1(ca),A(I1(csag),sag1,sad1),
A(I2(csad,nsad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,a/b) ∧
          Syncat(csad,b) ∧
          Trad (A(I1(csag),sag1,sad1),la) ∧
          Trad (A(I2(csad,nsad),sag2,sad2),lb) ∧
          Canred (A(la,lb),l).
Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
A(I1(csad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,a/b) ∧
          Syncat(csad,b) ∧
          Trad (A(I2(csag,nsag),sag1,sad1),la) ∧
          Trad (A(I1(csad),sag2,sad2),lb) ∧
          Canred (A(la,lb),l).
```

- Soit le SAG est de catégorie b et le SAD de catégorie $b\backslash a$, la traduction est obtenue de manière similaire en inversant l'ordre des termes de l'application

```
Trad (A(I1(ca),A(I1(csag),sag1,sad1),
A(I1(csad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,b) ∧
          Syncat(csad,b\a) ∧
          Trad (A(I1(csag),sag1,sad1),la) ∧
          Trad (A(I1(csad),sag2,sad2),lb) ∧
          Canred (A(lb,la),l).
```

```
Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
A(I2(csad,nsad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,b) ∧
          Syncat(csad,b\a) ∧
          Trad (A(I2(csag,nsag),sag1,sad1),la) ∧
          Trad (A(I2(csad,nsad),sag2,sad2),lb) ∧
          Canred (A(lb,la),l).
Trad (A(I1(ca),A(I1(csag),sag1,sad1),
A(I2(csad,nsad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,b) ∧
          Syncat(csad,b\a) ∧
          Trad (A(I1(csag),sag1,sad1),la) ∧
          Trad (A(I2(csad,nsad),sag2,sad2),lb) ∧
          Canred (A(lb,la),l).
Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
A(I1(csad),sag2,sad2)),l)
          ← Syncat(ca,a) ∧
          Syncat(csag,b) ∧
          Syncat(csad,b\a) ∧
          Trad (A(I2(csag,nsag),sag1,sad1),la) ∧
          Trad (A(I1(csad),sag2,sad2),lb) ∧
          Canred (A(lb,la),l).
```

5.6 Conclusion

Ce chapitre met en oeuvre la dernière phase du procédé de traduction d'une phrase en sa représentation sémantique. Au chapitre 3, nous avions abordé l'aspect syntaxique du langage naturel et nous avions montré comment construire un programme qui réalise automatiquement l'arbre d'analyse d'une phrase. Dans le chapitre 4, nous avons décrit un formalisme de représentation de la sémantique d'une phrase. Ce chapitre décrit la construction d'un programme qui permet de transformer l'arbre d'analyse d'une phrase en sa représentation sémantique.

Le procédé repose sur une approche fonctionnelle développée à l'aide du λ-calcul. L'idée consiste grossièrement à associer à chaque mot en fonction de sa classe grammaticale une λ-expression. La traduction d'un arbre d'analyse est obtenue soit à partir d'un lexique si l'information consiste en un mot et une catégorie soit en réduisant la traduction du sous-arbre gauche et du sous-arbre droit.

Dans le chapitre 6, nous nous intéresserons à l'utilisation de cette représentation pour évaluer la sémantique d'une phrase par rapport à un modèle.

Chapitre 6

L'interprétation

Nous avons étudié un langage de programmation dynamique (DPL). Il serait intéressant de pouvoir dériver automatiquement de ces programmes des formules logiques du premier ordre, car ces formules pourront être évaluées par rapport à un modèle.

Voyons d'abord comment sont définies les conditions dans la sémantique axiomatique développée par Hoare [Lloyd 1987], sémantique couramment utilisée en méthodologie de la programmation. Nous verrons ensuite comment adapter celle-ci pour les besoins particuliers de l'expression de conditions sur des programmes DPL. Nous verrons enfin comment dériver ces conditions de manière automatique. Nous terminerons par l'implémentation de cette dérivation automatique et d'une méthode d'évaluation d'une condition par rapport à un modèle.

6.1 La sémantique axiomatique

La sémantique axiomatique des programmes, met en jeu deux types de conditions: les préconditions et les postconditions ainsi qu'un programme.

Les triplets sont formalisés le plus souvent de la manière suivante :

- P : précondition
- Q : postcondition
- π : programme
- $\{P\}\pi\{Q\}$

La signification d'une telle formule dans son acceptation forte est que :

Si π est exécuté dans un état où l'assertion P est satisfaite, il se termine et aboutit alors à un état dans lequel l'assertion Q est vérifiée.

Dans une acceptation faible, la terminaison de π est donnée comme hypothèse.

Soit le programme Pascal suivant :

```
b := 6;
c := a + b.
```

Donnons-nous la postcondition : {c = 11}. L'assertion {a = 5;c = 11} est une précondition possible. En effet, elle assure bien que pour un état qui la vérifie après exécution du programme, l'état de sortie vérifie la postcondition. De même, {a = 5} est aussi une précondition possible. Elle est en fait une précondition particulière appelée la précondition la plus faible, car elle contraint au minimum l'état de départ.

En général, une condition (S) est dite plus faible qu'une autre (T) si S $\Rightarrow$ T. Elle est qualifiée de précondition la plus faible pour le programme et l'assertion si elle est plus faible que toute autre précondition pour ce même programme et condition. $\{a = 5; c = 13\}$ est également une précondition de $\pi\{Q\}$. Elle est plus forte que la précédente.

Il existe manifestement une dépendance entre une précondition et le couple programme et postcondition. De façon similaire, on définit une notion de postcondition la plus forte. Les conditions les plus intéressantes en programmation sont la précondition la plus faible et la postcondition la plus forte : c'est ainsi que l'on détermine le champ d'application du programme le plus étendu et que l'on précise le plus possible le résultat obtenu.

La garantie de la terminaison ou sa postulation comme hypothèse déterminent deux types de correction de programme.

On dit que π est totalement correct par rapport à P et à Q s'il respecte la condition suivante :

Si π est exécuté dans un état quelconque où P est vérifiée, l'exécution se termine et l'état résultant vérifie Q.

Par contre si on laisse la non terminaison de π possible, on dit que π est partiellement correct par rapport à P et à Q si et seulement s'il respecte la condition suivante :

Si π est exécuté dans un état quelconque qui vérifie P et si cette exécution se termine, l'état résultant vérifie Q.

Nous nous attacherons aux programmes totalement corrects seulement mais nous distinguerons malgré tout deux types de préconditions.

- Le premier type, que nous appellerons précondition existentielle, exprime, pour tout état vérifiant cette précondition pour un programme (π) et une postcondition (ψ) donnés, que

le programme DPL réussit et qu'au moins un des états de sortie vérifie la postcondition déterminée (ψ). La précondition minimale de ce type sera notée: $\langle\pi\rangle\psi$.

- Le second type de précondition qui nous intéresse est la précondition universelle notée $[\pi]\psi$. Elle exprime la condition sur un état d'entrée assurant que tous les états de sortie vérifient ψ.

Notons ici que contrairement à la programmation impérative séquentielle, il est possible d'obtenir un ensemble d'états de sortie différents pour un même état d'entrée comme il est possible d'obtenir un ensemble vide d'état. Ceci est dû au caractère non déterministe de l'instruction η de DPL.

6.2 Le langage Quantified Dynamic Logic (QDL)

Définissons tout d'abord le langage QDL [van Eijck 1992], inspiré de Pratt [Pratt 1976], qui nous servira à exprimer les conditions sur les programmes DPL.

6.2.1 La syntaxe QDL

L'ensemble des termes de DPL forme l'ensemble des termes QDL. Par contre, les formules bien formées de QDL sont définies par les règles suivantes.

1. Si R est une relation n-aire logique du premier ordre, et $t_1,\cdots,t_n$ sont des termes de QDL, alors $Rt_1\cdots t_n$ est une formule bien formée de QDL.
2. Si t_1,t_2 sont des termes de QDL, alors $t_1 = t_2$ est une formule bien formée de QDL.
3. Si φ ψ est une formule bien formée de QDL, alors $\varphi \wedge \psi$, $\neg$ ψ est une formule bien formée de QDL.
4. Si v $\in$ V et φ est une formule bien formée de QDL, alors $\exists v\varphi$ est une formule bien formée de QDL.
5. Si π est une formule bien formée de DPL et φ est une formule bien formée de QDL, alors $\langle\pi\rangle\varphi$ est une formule bien formée de QDL.

Pour simplifier les notations, nous utiliserons les abréviations habituelles suivantes où π représente un programme DPL et φ et ψ des expressions QDL:

- $\varphi \vee \psi$ pour $\neg(\neg\varphi \wedge \neg\psi)$

- $\varphi \rightarrow \psi$ pour $\neg(\varphi \wedge \neg\psi)$
- $[\pi]\varphi$ pour $\neg\langle\pi\rangle\neg\varphi$
- $\forall v\varphi$ pour $\neg\exists v\neg\varphi$
- $(\varphi \rightarrow \psi) \wedge (\varphi \leftarrow \psi)$ pour $(\varphi \leftrightarrow \psi)$
- $\exists! x\varphi$ pour $\exists x(\forall y\varphi(y/x) \leftrightarrow y = x)$ où y est une variable libre dans φ et φ(y/x) exprime le résultat de la substitution des occurrences libres de x dans φ par y.
- Nous symbolisons par $True$ la condition QDL qui est toujours vérifiée et par $False$ la condition QDL qui ne l'est jamais. Formellement $True$ peut être défini comme $\forall\ v : v = v$ et $False$ comme $\neg True$.

6.2.2 La sémantique QDL

La syntaxe du langage QDL étant définie, il reste à donner un sens aux préconditions. Un modèle est défini comme une paire $\mathcal{M} = \langle M, I\rangle$, avec M, un univers d'individus et I, une fonction d'interprétation pour les constantes et les symboles de relations du langage QDL.

Cela étant, la notion de satisfaction d'une expression QDL par un état (A) pour un modèle $\mathcal{M} = \langle M, I\rangle$ est définie comme suit.

1. $M \models_A Rt_1 \cdots t_n$ *si* $\langle \mathbf{V}_{M,A}(t_1), \cdots, \mathbf{V}_{M,A}(t_n)\rangle \in I(R)$.
2. $M \models_A t_1 = t_2$ *si* $\mathbf{V}_{M,A}(t_1) = \mathbf{V}_{M,A}(t_1)$.
3. $M \models_A \neg\varphi$ si $M \models_A \varphi$ n'est pas vérifié.
4. $M \models_A \varphi \wedge \psi$ si $M \models_A \varphi$ et $M \models_A \psi$.
5. $M \models_A \exists v\varphi$ s'il existe un $d \in M$ tel que $M \models_{A[v:=d]} \varphi$.
6. $M \models_A \langle\pi\rangle\varphi$ s'il y a un état B tel que $B \in [\![\pi]\!]_M(A)$ avec $M \models_B \varphi$.

6.3 Dérivation de DPL

Nous avons défini le langage QDL et nous avons identifié une notion de satisfaction sur l'ensemble du langage QDL. Voyons maintenant comment dériver d'un programme DPL et de sa postcondition une expression QDL.

La méthode utilisée est celle des axiomes de Pratt et des théroèmes dérivés de ceux-ci [Pratt 1976] [EACL 1993]. Elle est constituée d'une série d'équivalences. Le premier membre est constitué d'un programme DPL et d'une postcondition ($\langle\pi\rangle\varphi$ ou $[\pi]\varphi$). Le second membre

est une expression QDL avec soit une transformation terminale de π, soit une simplification de π en plusieurs composantes. La dérivation d'une précondition se fait de manière récursive. Nous allons présenter ces équivalences, exprimer leur signification de manière informelle et les justifier au niveau sémantique ou démontrer les théorèmes de type $[\pi]\varphi$. Nous envisagerons tout d'abord les cas de base, ensuite les cas complexes et nous clôturerons cette section par l'exposé des instructions particulières à DPL.

Par convention et afin d'éviter toute ambiguïté, nous noterons en gras les programmes DPL et en italique les expressions QDL.

6.3.1 Les cas de base

1. $\langle \mathbf{Rt_1 \cdots t_n} \rangle \varphi \leftrightarrow (Rt_1 \cdots t_n \wedge \varphi)$

   - Afin qu'avec un état d'entrée A, le programme test réussisse et qu'il produise au moins un état de sortie B qui vérifie φ, il faut et il suffit que A vérifie $Rt_1 \cdots t_n$ et φ.
   - $(\mathcal{M} \models_A \langle Rt_1 \cdots t_n \rangle \varphi)$
     $pour_{Definition\ satisfaction} \{A\} = [\![Rt_1 \cdots t_n]\!]_{\mathcal{M}}(A) \wedge \mathcal{M}_A \models \varphi$
     $pour_{Definition\ de [\pi]} \mathcal{M} \models_A Rt_1 \cdots t_n \wedge \mathcal{M} \models_A \varphi$
     $pour \mathcal{M} \models_A (Rt_1 \cdots t_n \wedge \varphi)$

2. $[\mathbf{Rt_1 \cdots t_n}] \varphi \leftrightarrow (Rt_1 \cdots t_n \rightarrow \varphi)$

   - Si tous les états de sortie de cette relation vérifient φ cela revient à exprimer que les états de sortie ne vérifient pas $\neg\varphi$.
   - $\Leftrightarrow_{definition\ [\mathbf{Rt_1 \cdots t_n}]\varphi} \neg\langle \mathbf{Rt_1 \cdots t_n} \rangle \neg\varphi$
     $\Leftrightarrow_{definition\ 1} \neg(Rt_1 \cdots t_n \wedge \neg\varphi)$
     $\Leftrightarrow_{notation} (Rt_1 \cdots t_n \rightarrow \varphi)$

3. $\langle \mathbf{t_1 = t_2} \rangle \varphi \leftrightarrow (t_1 = t_2 \wedge \varphi)$

   L'égalité de deux termes n'est en fait qu'un cas particulier de relation binaire et où la notation de R est infixée. La justification est donc la même.

4. $[\mathbf{t_1 = t_2}] \varphi \leftrightarrow (t_1 = t_2 \rightarrow \varphi)$

   Il s'agit de la même justification que celle réalisée au second point.

6.3.2 Les cas complexes

5. $\langle \pi_1; \pi_2 \rangle \varphi \leftrightarrow \langle \pi_1 \rangle \langle \pi_2 \rangle \varphi$

   - Le premier membre de l'équivalence correspond à la phrase suivante : "Pour un état d'entrée A, le programme $\pi_1; \pi_2$ réussit et a au moins un état de sortie qui vérifie φ". Le second correspond à : "Pour un état A le programme π_1 réussit et produit au moins un état de sortie B. Pour cet état d'entrée B, π_2 réussit et produit au moins un état

de sortie qui vérifie φ"
De par la définition sémantique de l'instruction DPL ";" nous pouvons dire que ces deux phrases sont équivalentes.

- $\mathcal{M} \models_A \langle \pi_1; \pi_2 \rangle \varphi pour \mathcal{M} \models_A \langle \pi_1 \rangle \langle \pi_2 \rangle \varphi$
  $pour_{Definition\ Satisfaction} \{A\} = [\![\pi_1; \pi_2]\!]_{\mathcal{M}}(A) \wedge \mathcal{M} \models_A \varphi$
  $pour_{Definition\ [\![\pi_1;\pi_2]\!]} \{A\} = \bigcup \{[\![\pi_2]\!]_{\mathcal{M}}(u)(u \in [\![\pi_1]\!]_{\mathcal{M}}(A)\} \wedge \models_A \varphi$
  $pour \mathcal{M} \models_A \langle \pi_1 \rangle \langle \pi_2 \rangle \varphi$

6. $[\pi_1; \pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$
   $\Leftrightarrow_{Definition\ [\pi]\varphi} \neg\langle \pi_1; \pi_2 \rangle \neg\varphi$
   $\Leftrightarrow_{Definition\ 5} \neg\langle \pi_1 \rangle \langle \pi_2 \rangle \neg\varphi$
   $\Leftrightarrow_{\neg\neg\varphi \leftrightarrow \varphi} \neg\langle \pi_1 \rangle \neg\neg\langle \pi_2 \rangle \neg\varphi$
   $\Leftrightarrow_{Definition\ [\pi]\varphi} \neg\langle \pi_1 \rangle \neg[\pi_2]\varphi$
   $\Leftrightarrow_{Definition\ [\pi]\varphi} [\pi_1][\pi_2]\varphi$

7. $\langle \neg\pi \rangle \varphi \leftrightarrow (\varphi \wedge [\pi] False)$.

   - Le premier membre de l'équivalence correspond à: "Il existe un état d'entrée A tel que le programme $\neg\pi$ réussit et qu'il produit au moins un état de sortie qui satisfait φ."
     Le second signifie que: "Il existe un état d'entrée qui satisfait φ et pour lequel le programme π échoue."
     La définition sémantique de la négation DPL montre l'équivalence de ces deux propositions.
   - $pour_{Definition [\![\neg\pi]\!]} \mathcal{M} \models_A \varphi \{A\} = [\![\neg\pi]\!]_{\mathcal{M}}(A)$
     $pour_{[\![\pi]\!]=\emptyset} \wedge \mathcal{M} \models_A \varphi \wedge \mathcal{M} \models_A \neg\pi$
     $pour \varphi \wedge [\pi] False$

8. $[\neg\pi]\varphi \leftrightarrow (\langle \pi \rangle True \vee ([\pi] False \wedge \varphi))$.
   $pour_{Definition\ semantique\ de\ []^\epsilon} \notin [\![\neg\pi]\!]_{\mathcal{M}}(s)$(tous les états de sortie de $\neg\pi$ sont propres) et $\forall s' \in [\![\neg\pi]\!]_{\mathcal{M}}(s), \mathcal{M}, s \models \varphi$
   $pour_{Definition\ semantique\ de\ \neg\pi} [\![\pi]\!] \neq \{\epsilon\}$ et (définition sémantique de $\neg\pi$ à nouveau)
   $[\![\pi]\!]_{\mathcal{M}}(s) = \emptyset$ implique que $\mathcal{M}, s \models \varphi$
   $pour_{par\ hypothese\ d'induction} \mathcal{M}, s \models \langle \pi \rangle True \vee [\pi] False$ (les états de sorties sont propres)
   $\mathcal{M}, s \models [\pi] False \rightarrow \varphi$ (définition sémantique de $[\neg\pi]$).
   $\Leftrightarrow (\langle \pi \rangle True \vee [\pi] False) \wedge ([\pi] \rightarrow \varphi)$
   $\Leftrightarrow (\langle \pi \rangle True \wedge \neg([\pi] False \wedge \neg\varphi)) \vee ([\pi] False \wedge \neg([\pi] False \wedge \neg\varphi))$
   $\Leftrightarrow (\langle \pi \rangle True \wedge (\neg[\pi] False \vee ([\pi] False \wedge \varphi)) \vee ([\pi] False \wedge (\neg[\pi] False \vee ([\pi] False \wedge \varphi))$
   $\Leftrightarrow ((\langle \pi \rangle True \wedge \neg[\pi] False) \vee (\langle \pi \rangle True \wedge [\pi] False \wedge \varphi)) \vee (([\pi] False \wedge \neg[\pi] False) \vee ([\pi] False \wedge \varphi))$
   $\Leftrightarrow (\langle \pi \rangle True \vee ([\pi] False \wedge \varphi))$

9. $\langle \pi_1 \Rightarrow \pi_2 \rangle \varphi \leftrightarrow (\varphi \wedge [\pi_1] \langle \pi_2 \rangle True)$

   - Le premier membre de l'équivalence correspond à: "Pour un état d'entrée A, l'exécution du programme $\pi_1 \Rightarrow \pi_2$ réussit et donne au moins un état de sortie qui vérifie

φ."
Le second signifie que: "Pour un état d'entrée A, non seulement φ est vérifiée mais tous les états de sortie de π_1 sont tels que π_2 réussit également."
De nouveau, de par la définition sémantique de l'instruction DPL $\Rightarrow$ nous pouvons dire que ces deux propositions sont équivalentes.

- $pour \mathcal{M} \models_A \langle \pi_1 \Rightarrow \pi_2 \rangle \varphi$
  $pour \{A\} = [\![\pi_1 \Rightarrow \pi_2]\!]_{\mathcal{M}}(A) \wedge \mathcal{M} \models_A \varphi$
  $pour (\forall u \in [\![\pi_1]\!]_{\mathcal{M}}(A) \Rightarrow [\![\pi_2]\!]_{\mathcal{M}}(u) \neq \emptyset) \wedge \mathcal{M} \models_A \varphi$
  $pour \varphi \wedge [\pi_1]\langle \pi_2 \rangle True$

10. $[\pi_1 \Rightarrow \pi_2]\varphi \leftrightarrow (\varphi \rightarrow \langle \pi_1 \rangle [\pi_2] False)$
    $pour_{Definition\ semantique\ de\ []^\epsilon} \notin [\![\pi_1 \Rightarrow \pi_2]\!]_{\mathcal{M}}(s)$(tous les états de sorties de $\pi_1 \Rightarrow \pi_2$ sont propres) et $\forall s' \in [\![\pi_1 \Rightarrow \pi_2]\!]_{\mathcal{M}}, \mathcal{M}, s \models \varphi$
    $pour_{Definition\ semantique\ de\ \pi_1 \Rightarrow \pi_2} [\![\pi_1]\!][\![\pi_2]\!] \neq \{\epsilon\}$ et (définition sémantique de $\pi_1 \Rightarrow \pi_2$ à nouveau) $[\![\pi_1]\!][\![\pi_2]\!](s) \neq \emptyset$ implique que $\mathcal{M}, s \models \varphi$
    $\Leftrightarrow ([\pi_1](\langle \pi_2 \rangle True \vee \langle \pi_2 \rangle False)) \wedge ([\pi_1]\langle \pi_2 \rangle True \rightarrow \varphi))$

6.3.3 Instructions particulières DPL ($\eta \& \iota$)

11. $\langle \eta v : \pi \rangle \varphi \leftrightarrow \exists v \langle \pi \rangle \varphi$.

    - Le membre de gauche de l'équivalence correspond à la proposition suivante:
      "Pour un état d'entrée A, le programme $\eta v : \pi$ réussit et produit au moins un état qui satisfait φ."
      Le membre de droite de l'équivalence signifie que:
      "Pour un état d'entrée A, il existe un élément d dans le domaine du modèle avec la propriété que pour l'état d'entrée $Av = d$, le programme π réussit et produit au moins un état de sortie vérifiant φ."
      La définition sémantique de l'instruction DPL η montre bien que ces deux propositions sont équivalentes.
    - $\langle \eta v : \pi \rangle \varphi pour \mathcal{M} \models_A \langle \eta v : \pi \rangle \varphi$
      $pour \mathcal{M} \models_B \varphi \wedge \{B\} = [\![\eta v : \pi]\!]_{\mathcal{M}}(A)$
      $pour \mathcal{M} \models_B \varphi \wedge \{B\} = \bigcup([\![\pi]\!](A(v/d) | d \in U\}$
      $pour \exists v \langle \pi \rangle \varphi$

12. $[\eta v : \pi]\varphi \leftrightarrow \forall v [\pi] \varphi$
    $\Leftrightarrow_{definition\ [\pi]} \neg \langle \eta v : \pi \rangle \neg \varphi$
    $\Leftrightarrow_{definition\ 11} \neg \exists v \langle \eta v : \pi \rangle \neg \varphi$
    $\Leftrightarrow_{definition\ \forall} \forall \neg \langle \eta v : \pi \rangle \neg \varphi$
    $\Leftrightarrow_{definition\ [\pi]} \forall v [\pi] \varphi$.

13. $\langle \iota v : \pi \rangle \varphi \leftrightarrow (\exists! v \langle \pi \rangle True \wedge \exists v \langle \pi \rangle \varphi)$.

    - Exprimons les deux membres de l'équivalence sous forme de propositions et vérifions leur équivalence.

Le membre de gauche de l'équivalence correspond à la proposition suivante: "Pour un état d'entrée A, le programme $\iota v : \pi$ réussit et donne au moins un état de sortie qui vérifie φ."
"Le membre de droite de l'équivalence correspond à la proposition suivante: "Pour un état d'entrée A, il existe un et un seul d appartenant au domaine du modèle tel que pour $v = d, \pi$ réussisse. De plus, il existe un d' appartenant à ce même domaine tel que $v = d'$ dans A et il produit au moins un état de sortie qui vérifie φ après exécution de π."
d et d' de la seconde proposition sont nécessairement égaux car d est l'unique valeur pour laquelle π réussisse. La définition sémantique de l'instruction DPL ι prouve bien l'équivalence de ces deux propositions.

- $pour \mathcal{M} \models_A \langle \iota v : \pi \rangle \varphi$
  $pour \mathcal{M} \models_B \varphi \wedge \{B\} = [\![\iota v : \pi]\!]_{\mathcal{M}}(A)$
  $pour \mathcal{M} \models_b \varphi \wedge \{B\} = \bigcup([\![\pi]\!]_{\mathcal{M}} A(v/d) | d \in U\} \wedge \exists! d \in U | [\![\pi]\!]_{\mathcal{M}} A(v/d) \neq \emptyset$
  $pour \exists! v \langle \pi \rangle True \wedge \exists v \langle \pi \rangle \varphi)$

14. $[\iota v : \pi]\varphi \leftrightarrow (\exists! v \langle \pi \rangle True \wedge \forall v(\langle \pi \rangle True \rightarrow [\pi]\varphi))$.

- Le membre de gauche de l'équivalence correspond à la proposition suivante: "Pour un état d'entrée A , le programme $\iota v : \pi$ ne produit que des états de sortie satisfaisants φ"
  Le membre de droite de l'équivalence signifie que: "Pour un état d'entrée A, il n'existe qu'un seul d appartenant au domaine du modèle tel que $v = d$ dans A et que π réussisse. Pour toute valeur d' appartenant au domaine du modèle, le programme π avec un état d'entrée A où $v = d'$ réussit et tous les états de sortie vérifient φ."
  Donc nous pouvons dire qu'il n'existe qu'une seule valeur du domaine telle que π réussisse et que ces états de sortie vérifient φ. Par conséquent $d = d'$ et les deux propositions sont bien égales.

- $\neg \langle \iota v : \pi \rangle \neg \varphi$
  $\Leftrightarrow_{Definition \langle \iota v : \pi \rangle} \neg(\exists! v \langle \pi \rangle True \wedge \exists v \langle \pi \rangle \neg \varphi)$
  $\Leftrightarrow_{Definition \wedge} (\exists! v \langle \pi \rangle True \wedge \neg \exists v \langle \pi \rangle \neg \varphi) \vee (\neg \exists! v \langle \pi \rangle True \wedge \wedge \exists v \langle \pi \rangle \neg \varphi)$
  $\Leftrightarrow \exists! v \langle \pi \rangle True \wedge \forall v(\langle \pi \rangle True \rightarrow \neg \langle \pi \rangle \neg \varphi)$
  $\Leftrightarrow \exists! v \langle \pi \rangle True \wedge \forall v(\langle \pi \rangle True \rightarrow [\pi] \varphi)$

6.4 Exemples

Nous disposons maintenant d'un système complet pour dériver les préconditions d'un programme DPL associé à une postcondition dans un langage logique du premier ordre.

Soit la phrase: "$Jean^1$ voit un^2 homme"

La traduction DPL: $\eta\ v_1 : v_1 =$ Jean ; $\eta\ v_2 :$ homme(v_2); voir($v_1\ v_2$)

Grâce aux règles vues dans la section précédente, calculons la précondition existentielle de succès.

$$
\begin{aligned}
&\langle \eta \mathbf{v_1} : \mathbf{v_1} = \mathbf{Jean}; \eta \mathbf{v_2} : \mathbf{homme(v_2)}; \mathbf{voir(v_1 v_2)} \rangle True \\
\leftrightarrow_{Axiome\langle\eta\rangle}\ & \exists\ v_1\ \langle\ \mathbf{v_1} = \text{Jean}\ ;\ \eta\ v_2 : \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ \rangle True \\
\leftrightarrow_{Axiome\langle;\rangle}\ & \exists\ v_1\ \langle\ \mathbf{v_1} = \text{Jean}\ \rangle\langle\ \eta\ v_2 : \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ \rangle True \\
\leftrightarrow_{Axiome\langle\eta\rangle}\ & \exists\ v_1\ \langle\ \mathbf{v_1} = \text{Jean}\ \rangle\ \exists\ v_2\ \langle\ \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ \rangle True \\
\leftrightarrow_{Axiome\langle;\rangle}\ & \exists\ v_1\ \langle\ \mathbf{v_1} = \text{Jean}\ \rangle\ \exists\ v_2\ \langle\ \text{homme(}\ v_2\ \text{)}\ \rangle\langle\ \text{voir(}\ v_1,v_2\ \text{)}\ \rangle True \\
\leftrightarrow_{Axiome\langle Rt_1,\cdots,t_n\rangle}\ & \exists\ v_1\ \langle\ \mathbf{v_1} = \text{Jean}\ \rangle\ \exists\ v_2\ \text{homme(}\ v_2\ \text{)} \wedge \text{voir(}\ v_1,v_2\ \text{)} \\
\leftrightarrow_{Axiome\langle t_1=t_2\rangle}\ & \exists\ v_1\ (v_1 = \text{Jean}) \wedge \exists\ v_2\ (\text{homme(}\ v_2\ \text{)} \wedge \text{voir(}\ v_1,v_2\ \text{))}
\end{aligned}
$$

Cette condition exprime bien qu'un certain Jean dans le modèle (qui est en fait la connaissance) voit quelque chose (v_2) et que ce quelque chose possède la propriété d'être un homme.

Pour cette même phrase, déterminons la précondition universelle d'échec $[\pi]False$, ce qui correspond au fait que tout état d'entrée vérifiant cette condition échoue.

$$
\begin{aligned}
&[\eta\ v_1 : v_1 = \mathbf{Jean}\ ;\ \eta\ v_2 : \mathbf{homme(}\ v_2\ \mathbf{)}\ ;\ \mathbf{voir(}\ v_1\ v_2\ \mathbf{)}\ ] False \\
\leftrightarrow_{Axiome[\eta]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ;\ \eta\ v_2 : \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ ] False \\
\leftrightarrow_{Axiome[;]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ][\ \eta\ v_2 : \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ ] False \\
\leftrightarrow_{Axiome[\eta]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ]\ \forall\ v_2\ [\ \text{homme(}\ v_2\ \text{)}\ ;\ \text{voir(}\ v_1,v_2\ \text{)}\ ] False \\
\leftrightarrow_{Axiome[;]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ]\ \forall\ v_2\ [\ \text{homme(}\ v_2\ \text{)}\ ][\ \text{voir(}\ v_1,v_2\ \text{)}\ ] False \\
\leftrightarrow_{Axiome[Rt_1,\cdots,t_n]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ]\ \forall\ v_2\ [\mathbf{homme(v_2)}]\ (\text{voir(}\ v_1,v_2 \rightarrow False) \\
\leftrightarrow_{Axiome[Rt_1,\cdots,t_n]}\ & \forall\ v_1\ [\ \mathbf{v_1} = \text{Jean}\ ]\ \forall\ v_2\ \text{homme(}\ v_2\ \text{)} \rightarrow (\text{voir(}\ v_1,v_2 \rightarrow False) \\
\leftrightarrow_{Axiome[t_1=t_2]}\ & \forall\ v_1\ v_1 = \text{Jean} \rightarrow (\forall\ v_2\ \text{homme(}\ v_2\ \text{)} \rightarrow (\text{voir(}\ v_1,v_2) \rightarrow False))
\end{aligned}
$$

Cette définition correspond bien à ce que naturellement on pouvait en attendre. En effet, cette condition est vérifiée si Jean ne voit personne ou s'il voit autre chose qu'un homme. La présupposition d'erreur est exprimée de la manière suivante $\neg\langle\pi\rangle True \wedge \neg[\pi]False$. Il est inutile de détailler la formule pour vérifier que cette définition est bien sensée. En effet, si ni la précondition existentielle de succès ni la précondition universelle d'échec ne sont vérifiées, nous sommes alors face à un état d'erreur que l'on notera ϵ. Il se peut tout simplement que la relation ne soit pas définie dans le modèle. Mais le cas le plus intéressant (pour le moment) est principalement celui de l'assignation ι. En effet, s'il existe plusieurs valeurs admissibles et non pas une seule dans un modèle, l'évaluation dans ce modèle va générer une erreur.

- Soit, par exemple la phrase : "Le[1] caractère avec le[2] chapeau est une[3] capitale."
  et un contexte qui consiste en une suite de caractères majuscules ou miniscules avec ou sans accent circonflexe : "a^pPŶ".

  Dans ce contexte, l'évaluation de la précondition de succès va réussir car le seul caractère

avec un accent circonflexe Y est bien une capitale. Nous pouvons dire que ni la précondition universelle d'échec ni la présupposition d'erreur ne sont vérifiées.

- Par contre, si le contexte est le suivant : "a^p̂Kv".

  La précondition existentielle de succès n'est pas vérifiée dans ce modèle. En effet, le seul caractère avec un accent circonflexe n'est pas une capitale. Par contre, la précondition universelle d'échec est vérifiée pour la même raison. Par définition, la présupposition d'erreur n'est pas vérifiée.

- Comme dernier cas, prenons : "a^P̂p̂Kv"

  La précondition existentielle de succès n'est pas vérifiée car il y a plusieurs caractères avec un accent circonflexe. La précondition universelle d'échec ne l'est pas non plus: pour cela, il aurait fallu un seul caractère minuscule avec un accent circonflexe.
  Par définition, la présupposition d'erreur est vérifiée puisque les deux premières ne le sont pas. Ce qui correspond à une ambiguïté du langage car on ne parvient pas à déterminer "le caractère avec un chapeau" puisqu'il y en a deux dans ce contexte. Cet état d'erreur est donc bien défini et fondé. Nous verrons comment étendre cette notion d'erreur à d'autres notions.

6.5 Implémentation

L'implémentation de cette partie s'est faite en trois phases. Un premier module "Axioms" se charge de la dérivation de DPL vers QDL. Ensuite, un petit module "Simplification" simplifie les notations QDL. Le dernier module "Eval" évalue l'expression QDL par rapport à un modèle représenté sous forme d'un base de données logique.

6.5.1 Axiomes

L'implémentation de ce module consiste à réécrire les axiomes développés dans la section précédente en respectant les formats de représentation de DPL et de QDL ainsi que les règles de la programmation logique. Nous donnerons la liste des équivalents logiques (en format Gödel) aux axiomes. Axiom est un prédicat, le premier argument correspond au membre de gauche des équivalences et le second au membre de droite. Les différentes fonctions de représentation de DPL et de QDL sont présentées brièvement au fur et à mesure.

1. `Axiom (Fprecexist (PR (a),phi),Fand ( FR (a),phi)).`
   $pour\langle \mathbf{Rt_1 \cdots t_n} \rangle \varphi \leftrightarrow (Rt_1 \cdots t_n \wedge \varphi)$

Le premier argument est composé de
PR (a) pour $\mathbf{Rt_1 \cdots t_n}$
la variable phi pour φ

Le second argument est composé de
FR (a) pour $Rt_1 \cdots t_n$
la variable phi pour φ
et la fonction Fand pour unir les deux expressions.

2. `Axiom (Fprecuniv(PR(a),phi),Fimplic(FR(a),phi)).`
   pour $[\mathbf{Rt_1 \cdots t_n}]\varphi \leftrightarrow (Rt_1 \cdots t_n \rightarrow \varphi)$

3. `Axiom (Fprecexist (Pequal (t1,t2),phi),Fand (Fequal(t1,t2),phi)).`
   pour $\langle \mathbf{t_1 = t_2} \rangle \varphi \leftrightarrow (t_1 = t_2 \wedge \varphi)$
   On utilise ici, les fonctions Pequal et Fequal respectivement équivalentes à $\mathbf{t_1 = t_2}$ et $t_1 = t_2$.

4. `Axiom (Fprecuniv(Pequal(t1,t2),phi),Fimplic(Fequal(t1,t2),phi)).`
   pour $[\mathbf{t_1 = t_2}]\varphi \leftrightarrow (t_1 = t_2 \rightarrow \varphi)$

5. `Axiom (Fprecexist (Papplic (p1,p2),phi),x)`
   `← Axiom (Fprecexist(p2,phi),y) ∧Axiom (Fprecexist(p1,y),x).`
   pour $\langle \pi_1 ; \pi_2 \rangle \varphi \leftrightarrow \langle \pi_1 \rangle \langle \pi_2 \rangle \varphi$
   La fonction Papplic est l'équivalent Gödel de ;.
   C'est le premier cas où l'on applique la définition axiomatique de manière récursive.

6. `Axiom (Fprecuniv(Papplic(p1,p2),phi),z)`
   `← Axiom(Fprecuniv(p2,phi),z1) ∧ Axiom(Fprecuniv(p1,z1),z).`
   pour $[\pi_1 ; \pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$

7. `Axiom (Fprecexist (Pnot (p),phi),Fand (phi,a))`
   `← Axiom (Fprecuniv(p,FBottom),a).`
   pour $\langle \neg \pi \rangle \varphi \leftrightarrow (\varphi \wedge [\pi] False)$.
   False est une constante Gödel.
   Pnot est la fonction Gödel pour l'instruction DPL ¬

8. `Axiom (Fprecuniv(Pnot(p),phi),For(z,Fand(y,phi))`
   `← Axiom(Fprecexist(p,FTop),z) ∧ Axiom(Fprecuniv(p,FBottom),y).`
   pour $[\neg \pi]\varphi \leftrightarrow ((\langle \pi \rangle True \wedge \neg \varphi) \vee [\pi \rangle False)$

9. `Axiom (Fprecexist (Pimplic(p1,p2),phi),Fand (For(y,z),Fimplic (z2,phi))`
   `← Axiom (Fprecexist (p2,FTop),y) ∧`
   `Axiom (Fprecuniv (p2,FBotom),z) ∧`
   `Axiom (Fprecuniv (p1,y),z2).`
   pour $\langle \pi_1 \Rightarrow \pi_2 \rangle \varphi \leftrightarrow (\varphi \wedge [\pi_1] \langle \pi_2 \rangle True)$
   Pimplic est l'équivalent Gödel de $\Rightarrow$.

10. `Axiom (Fprecuniv(Pimplic(p1,p2),phi),Fand(phi1,Fimplic(phi21,phi)))`
    `← Axiom(Fprecexist(p2,FTop),phi121) ∧`
    `Axiom(Fprecuniv(p2,FBottom),phi122) ∧`
    `Axiom(Fprecuniv(p1,For(phi121,phi122)),phi1) ∧`
    `Axiom(Fprecuniv(p1,phi121),phi21).`

    pour $[\pi_1 \Rightarrow \pi_2]\varphi \leftrightarrow (([\pi_1](\langle\pi_2\rangle True \vee [\pi_2]False)) \wedge ([\pi_1]\langle\pi_2\rangle True \rightarrow \varphi))$

11. `Axiom (Fprecexist (Peta (v,p),phi),Fexist (v,y))`
    `← Axiom (Fprecexist (p,phi),y).`

    pour $\langle \eta \mathbf{v} : \pi\rangle\varphi \leftrightarrow \exists v\langle\pi\rangle\varphi$.

    Peta est la fonction Gödel correspondant à η.

12. `Axiom (Fprecuniv(Peta(v,p),phi),Fall(v,z))`
    `← Axiom(Fprecuniv(p,phi),z).`

    pour $[\eta v : \pi]\varphi \leftrightarrow \forall v[\pi]\varphi$.

13. `Axiom (Fprecexist (Piota (v,p),phi),`
    `Fand (Fonlyone(v,z),Fexist(v,y)))`
    `← Axiom (Fprecexist (p,phi),y) ∧ Axiom (Fprecexist (p,FTop),z).`

    pour $\langle \iota \mathbf{v} : \pi\rangle\varphi \leftrightarrow (\exists! v\langle\pi\rangle True \wedge \exists v\langle\pi\rangle\varphi)$.

14. `Axiom (Precuniv(Piota(x,p),phi),`
    `Fand (Fonlyone(x,y),Fall(x,Fimplic(y,z))`
    `← Axiom (Precexist(p,Ftop),y) ∧ Axiom (Precuniv(p,phi),z).`

    pour $[\iota \mathbf{x} : \pi]\varphi \leftrightarrow \exists! x\langle\pi\rangle True \wedge (\forall x(\langle\pi\rangle True \rightarrow [\pi]\varphi))$

Nous remarquons que la précondition existentielle et la précondition universelle sont définies de manière récursive et que la transcription de ces axiomes en langage logique est directe.

6.5.2 Simplification

Les formules obtenues par la dérivation de programme DPL prennent très vite des proportions importantes. Nous avons mis au point un petit module qui permet de simplifier quelque peu ces formules. Ce module traite les simplifications suivantes :

- $\varphi \wedge True$ pour φ
- $\varphi \wedge False$ pour $False$
- $\varphi \vee True$ pour $True$
- $\varphi \vee False$ pour φ

- $(\varphi \rightarrow True)$ pour $True$
- $(True \rightarrow \varphi)$ pour φ
- $(\varphi \rightarrow False)$ pour $\neg\varphi$
- $(False \rightarrow \varphi)$ pour $True$
- $\neg False$ pour $True$
- $\neg True$ pour $False$
- $(\exists!v\varphi) \wedge (\exists v\varphi)$ pour $(\exists!v\varphi)$

6.5.3 Evaluation

Ce module est le plus intéressant de cette partie mais aussi le plus complexe. Si l'évaluation d'une condition se fait de manière intuitive pour nous, celle-ci est beaucoup plus complexe pour un programme. Nous avons choisi de représenter le modèle (la connaissance) sous forme d'une base de données logiques. Ceci a le double avantage d'une part de tester les capacités de métaprogrammation de Gödel mais aussi de simplifier nettement le problème de représentation et de construction de requêtes et de buts par rapport à une base de données de type relationelle par exemple. Le système est constitué de trois points :

1. La base de données (autrement dit le modèle) est représentée sous forme d'un programme Gödel objet. Cette base de données sera considérée comme un programme objet par le métaprogramme "Eval". Pour simplifier les problèmes de représentation des termes, nous les avons tous déclarés comme constantes sous le type Symbol. Cette simplification n'altère pas la généralité de la méthode utilisée. $True$ est déclaré comme une proposition vraie dans tout modèle. Les autres déclarations dépendent de la connaissance qui nous intéresse.
   Reprenons l'exemple des caractères avec les accents circonflexes.
   Voici un contexte "aâX". Nous allons déclarer comme constante de type "Symbol" : Ap, Blanc, Bp, Xg, Accent et Capitale, et comme prédicats : Avec(Symbol,Symbol), Capitale(Symbol), caractère(Symbol), Is(Symbol), Chapeau(Symbol). Les clauses correspondantes au contexte sont :

   - Capitale(Capitale).
   - Caractère(Xg).
   - Caractère(Bp).
   - Caractère(Ap).
   - Is(Xg,Capitale).
   - Avec (Blanc,Accent).
   - Avec (Xg,Accent).

- Chapeau(Accent).

Nous avons défini le programme objet conforme au contexte. Après compilation, il sera manipulable par métaprogramme "Eval".

2. L'ensemble des instantiations au niveau du programme objet constitue l'état. Cet ensemble est géré par Gödel. Il sera obtenu automatiquement après application du but par le métaprogramme au programme objet (base de données logiques).

3. La condition QDL est traduite sous forme d'un but Gödel qui est appliqué au programme objet(1) par le métaprogramme "Eval".

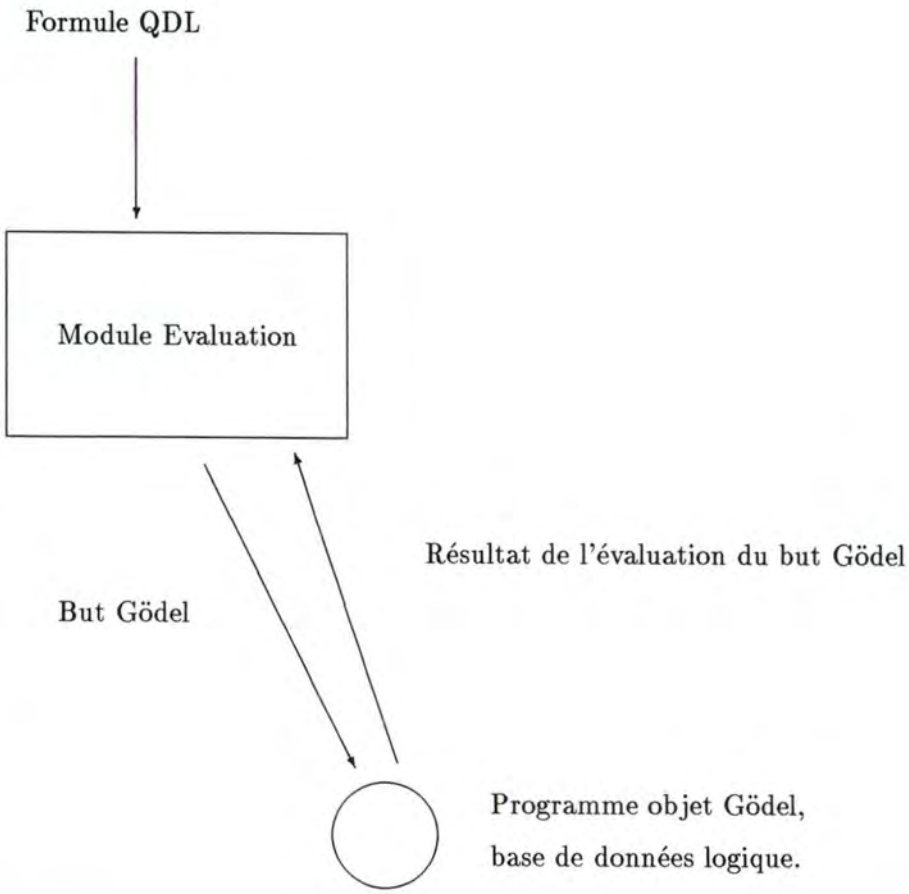


Figure 6.1. Schéma d'évaluation

Si les deux premiers points peuvent être implémentés sans problème, le troisième est quant à lui un peu plus complexe. En principe, QDL étant un langage logique du premier ordre, il est donc possible de le traduire sous forme de but (comme l'illustre la Figure 6.1). Mais les conditions QDL deviennent très vites complexes, tellement complexes qu'elles sont purement et simplement refusées par l'interpréteur Gödel. Nous avons contourné cette difficulté en introduisant de manière temporaire des définitions de prédicats (cfr. Figure 6.2) et des définitions de clauses intermédiaires au sein du programme objet (modèle).

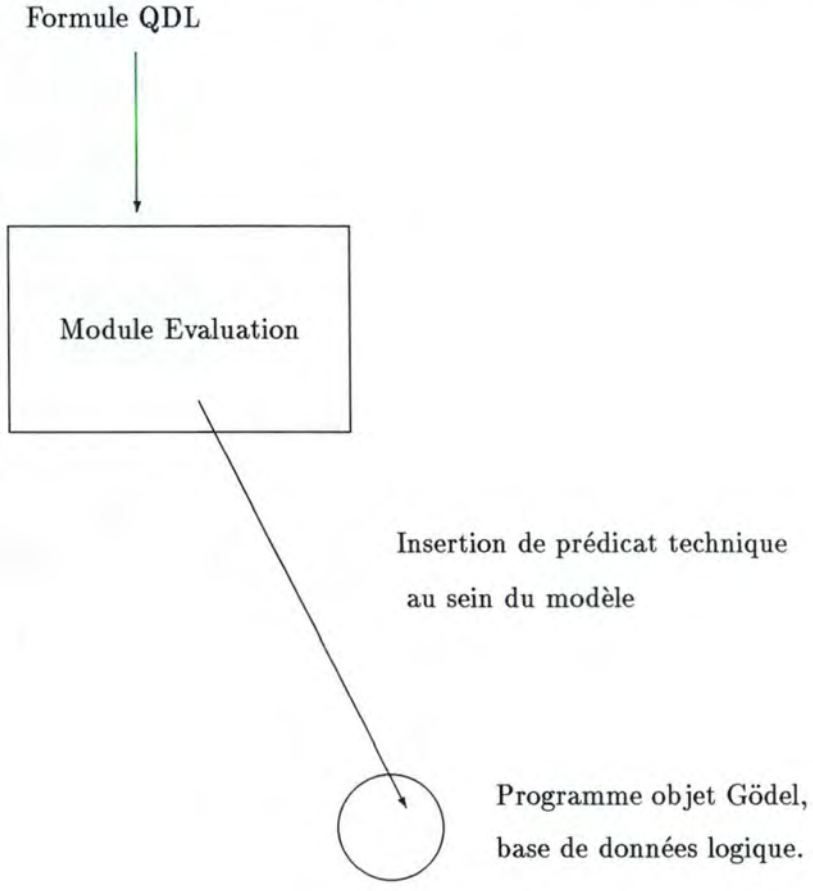


Figure 6.2. Schéma d'insertion d'un prédicat intermédiaire

Nous allons exposerons les différentes traductions avec leurs effets sur les arguments du prédicat. Nous présenterons brièvement les prédicats qu'offre Gödel pour faciliter la métapro-

grammation.
Nous expliquons ci-dessous les arguments du prédicat qui se charge de la traduction d'une expression QDL en un langage conforme à un programme Gödel objet.

- le programme objet en entrée;
- le nom du module du programme objet concerné;
- le numéro du dernier prédicat technique créé ou de la dernière proposition technique créée en entrée;
- la formule QDL en entrée;
- le but correspondant à la formule QDL exprimé dans un langage conforme au module du programme objet concerné et exécutable par l'interpréteur Gödel;
- le programme objet en sortie. Il est utile d'obtenir ce résultat en sortie car il se peut que l'on introduise des clauses intermédiaires au sein même du programme objet. Le programme objet résultat peut donc être différent de celui en entrée;
- le numéro du dernier prédicat créé en sortie;
- la liste des variables libres de l'expression QDL à traduire. Lors de la création d'un prédicat technique, cette liste sert à déterminer les arguments du prédicat créé.

Pour détailler la traduction des formules QDL en buts Gödel, nous avons déterminé trois niveaux de complexité :

1. Les cas de base: traduction pratiquement immédiate.
2. Les cas complexes: composition de cas de base.
3. Les quantificateurs QDL: certains ont un équivalent Gödel mais pas tous.

Les cas de base

1. *True*

   Nous utilisons un prédicat mis à disposition par le module Programs de Gödel qui traduit un string en une représentation "flat" dans un langage conforme au module du programme concerné.

   - La représentation du programme objet est inchangée.

- Le numéro du dernier prédicat technique créé au niveau du programme objet est inchangé.

2. $Rt_1, \cdots, t_n$

   Il suffit de changer la représentation de la relation, sous forme de prédicat conforme à la définition qui se trouve dans le programme objet. Tout ceci se fait très aisément grâce aux facilités offertes par Gödel. En effet, il suffit de traduire la liste de termes et la relation sous forme d'un string de type "$Rt_1, \cdots, t_n$". Après quoi, nous utilisons le même prédicat que pour $True$.

   - La représentation du programme objet est inchangée.
   - Le numéro du dernier prédicat technique créé au niveau du programme objet est inchangé.
   - La liste des variables libres est $t_1, \cdots, t_n$.

3. $t_1 = t_2$

   Il est nécessaire de différencier les cas suivants :

   - $v_1 = Constante$
   - $v_1 = v_2$

   En effet, les variables libres à introduire dans la liste sont soit v_1, soit v_1 et v_2.

   - La représentation du programme objet est inchangée.
   - Le numéro du dernier prédicat technique créé au niveau du programme objet est inchangé.
   - La liste des variables libres est v_1 dans le premier cas et v_1, v_2 dans le second.

Les cas complexes

4. $\varphi \wedge \psi$ (respectivement [$\varphi \vee \psi$; $\varphi \leftarrow \psi$; $\neg\varphi$])

   Par hypothèse de récurrence, on suppose que φ et ψ (pas pour $\neg\varphi$) sont traduisibles et ont pour traduction en Gödel respectivement φ' et ψ'. Grâce aux facilités offertes par Gödel, il est facile de lier ces deux propositions conformes au programme objet par un "et" (respectivement [ou; implication et négation]) logique pour ne plus en former qu'une seule, le résultat sera : $\varphi' \wedge \psi'$ (respectivement [$\varphi \vee \psi$; $\varphi \leftarrow \psi$; $\neg\varphi$]).

   Pour lier ces deux traductions, on utilise le prédicat And (respectivement [Or; Implies; Not]) défini dans le module Programs de Gödel qui donne comme résultat une troisième expression équivalente à $\varphi' \wedge \psi'$ (respectivement [$\varphi' \vee \psi'$; $\varphi' \leftarrow \psi'$; $\neg\varphi'$]).

- Dans un premier temps, la représentation du programme objet peut avoir été modifiée au cours de la traduction de φ. Cette représentation du programme objet éventuellement modifiée peut de nouveau l'être lors de la traduction de ψ (sauf pour $\neg\ \varphi$). Au niveau de la traduction du et (respectivement ou, implication, négation) logique en lui(elle)-même, la représentation du programme objet n'est pas modifiée.
- Le numéro du dernier prédicat technique créé au niveau du programme objet peut avoir été modifié au cours de la traduction de φ, dans un premier temps. Ce numéro éventuellement modifié peut de nouveau l'être lors de la traduction de ψ (sauf pour $\neg\ \varphi$). Au niveau de la traduction du "et" (respectivement [ou,implication,négation]) logique en lui(elle)-même, le numéro du dernier prédicat créé est inchangé.
- La liste des variables libres de l'expression $\varphi \wedge \psi$ (respectivement [$\varphi \vee \psi$; $\varphi \leftarrow \psi$; $\neg\varphi$]) est l'union des listes de variables libres de φ et de ψ (sauf pour $\neg\varphi$).

Les quantifcateurs QDL

5. $\exists v : \varphi$ (respectivement $[\forall v : \varphi]$)

Par hypothèse de récurrence on suppose φ traduisible en φ' conforme au programme objet. Il est possible de construire grâce aux possibilités de Gödel une expression équivalente à $\exists\varphi$ (respectivement [$\forall v : \varphi$]) mais celle-ci est trop complexe pour être admise dans un but. Nous introduisons une nouvelle définition de prédicat dans le programme. Le nom de ce prédicat est purement technique (concaténation d'une lettre arbitrairement choisie auparavant et d'un compteur). Le nombre d'arguments du prédicat est déterminé par la liste des variables non-quantifiées de φ de laquelle on retire la variable v. La définition de la clause de ce nouveau prédicat est la traduction de $\exists v : \varphi$ (respectivement [$\forall v : \varphi$]) en Gödel. Le résultat de la traduction de $\exists v : \varphi$ (respectivement [$\forall v : \varphi$]) est le nouveau prédicat qui lui est assez simple pour faire partie d'un but. Le prédicat utilisé pour construire l'expression $\exists v : \varphi'$ (respectivement [$\forall v : \varphi$]) est Some (respectivement All), qui est défini dans le module Programs de Gödel.

- Dans une première étape, la représentation du programme objet peut avoir été changée au cours de la traduction de φ. Cette représentation éventuellement modifiée l'est de toute manière au cours de la traduction de l'expression $\exists v\varphi$ (respectivement [$\forall v : \varphi$]) en elle-même. En effet, l'introduction du prédicat technique (ou de la proposition) modifie évidemment la représentation du programme objet.
- Dans un premier temps le numéro du dernier prédicat technique créé au niveau du programme objet peut avoir été modifié au cours de la traduction de φ. Le numéro du dernier prédicat technique créé obtenu après traduction de $\exists v : \varphi$ (respectivement [$\forall v : \varphi$]) est incrémenté de 1. Cette incrémentation correspond à l'introduction dans le programme objet du prédicat (ou de la proposition) nécessaire pour traduire l'expression $\exists v\varphi$ (respectivement [$\forall v : \varphi$]).
- La liste des variables libres de l'expression $\exists v\varphi$
(respectivement [$\forall v : \varphi$]) est la liste des variables libres de l'expression φ de laquelle nous retirons v.

6. $\exists!v : \varphi$

Ceci est le cas le plus compliqué car non seulement il nous faut définir un prédicat intermédiaire mais de plus, il n'existe pas d'équivalent Gödel à ce type de quantificateur. Le principe pour construire la définition du prédicat intermédiaire est toujours le même. En ce qui concerne la traduction de $\exists!v\varphi$, nous avons choisi de rechercher l'ensemble des v qui vérifient φ et de tester si le cardinal de cet ensemble est égal à un. C'est ainsi qu'a été définie la clause du prédicat intermédiaire.

- Dans un premier temps, la représentation du programme objet peut avoir été modifiée au cours de la traduction de φ. Cette représentation éventuellement modifiée l'est de toute manière au cours de la traduction de l'expression $\exists!v\varphi$ en elle-même. En effet, l'introduction du prédicat technique (ou de la proposition) modifie évidemment la représentation du programme objet.
- Dans un premier temps, le numéro du dernier prédicat technique créé au niveau du programme objet peut avoir été modifié au cours de la traduction de φ. Au niveau de la traduction de l'expression $\exists!v\varphi$ en elle-même, le numéro du dernier prédicat technique créé éventuellement modifié est de toute manière incrémenté de 1. Cette incrémentation correspond à l'introduction dans le programme objet du prédicat (ou de la proposition) nécessaire pour traduire l'expression $\exists!v\varphi$.
- La liste des variables libres de l'expression $\exists!v\varphi$ est la liste des variables libres de l'expression φ de laquelle nous retirons v.

Nous sommes maintenant en mesure de traduire toutes les expressions QDL. L'évaluation d'une expression QDL φ se fait aisément, il suffit d'appliquer comme but la traduction φ' de φ au programme objet qui aura éventuellement été modifié au cours de la traduction de φ. Si le but réussit, la condition φ est vérifiée dans le modèle, sinon elle ne l'est pas.

6.6 Les présuppositions

Comme nous venons de le voir (Cfr. Section 6.4), l'état d'erreur indique un échec de l'opérateur ι de DPL. Dans les phrase du type "le caractère avec un chapeau $\cdots$" par exemple, on présuppose que le "caractère" qui remplit la condition "avec un chapeau" est unique. Il serait intéressant d'étendre cette notion d'erreur à d'autres domaines. Par exemple, le traitement du phénomène de début d'une action et de fin d'une action ou le traitement de la présupposition lexicale. Dans cette section nous évoquerons tout d'abord le traitement du premier phénomène et nous détaillerons ensuite la solution implémentée pour traiter le second phénomène.

6.6.1 Le phénomène de début et de fin d'une action

Voyons par l'analyse détaillée d'un exemple en quoi consiste ce phénomène. Soit une personne que nous appellerons Jean qui est en train de marcher. A un moment donné Jean s'arrête pour une raison ou pour une autre. Pour que le type de phrase "Jean s'arrête de marcher" soit correctement interpreté, il est nécessaire de vérifier que la personne était en train de marcher. Par "était en train de marcher", on entend que cette personne à commencer à marcher à moment t antérieur au moment où il s'est arrêté (disons $t + i$) et qu'il ne s'est pas arrêté entre temps. Le traitement de ce genre de phénomène nécessite l'ajout d'une variable temps. C'est sur base de cette variable que l'on peut traiter l'antériorité d'une action par rapport à une autre. Ainsi, dans notre cas, si le contexte est tel que Jean n'était pas en train de marcher, l'interprétation de la phrase "Jean s'arrête de marcher" doit générer une erreur. Nous ne traitons pas ce type de phénomène dans notre programme. Mais un membre du C.W.I. étend notre système de manière à pouvoir l'interpréter. A partir de l'introduction de la variable temps comme paramètre, on peut envisager de traiter la conjugaison des verbes (futur,passé,présent...) [Muskens 1992].

6.6.2 La présupposition lexicale

Pour expliciter cette notion de présupposition lexicale, nous allons nous baser sur un exemple. Quand on parle de quelqu'un comme étant "le mari de y", ce quelqu'un est forcément de sexe masculin et marié à y. Le fait que cette personne est marié et de sexe masculin constitue la présupposition lexicale de la relation $maride(x, y)$. Définissons une fonction (Lp) qui à une relation (ou une égalité de termes) fait correspondre un programme DPL qui exprime la présupposition lexicale de cette relation (égalité de termes). Si la relation n'a pas de présupposition lexicale, Lp pointera vers le programme $\top$. Seuls les axiomes portant sur la relation ou l'égalité de termes ont été modifiés. Voici les nouvelles définitions avec leur équivalent en programmation logique. La fonction Lp a été définie sous forme d'un prédicat qui a pour arguments des programmes DPL:

```
Lp(PR(Rel("maride",[i,j])),Papplic(PR(Rel("homme",i)),PR(Rel("marie",[i,j])))).
Lp (_,PTop).
```

L'argument de droite constitue la relation ou l'égalité de termes pouvant faire l'objet d'une présupposition et le membre de gauche la présuposition en elle-même. Si la relation ou l'égalité de termes n'a pas de présupposition, le programme DPL associé est $\top$ (PTop dans le programme logique ci-dessous), ce qui corresspond bien au fait que la présupposition est toujours vérifiée.

1. $\langle \mathbf{Rt_1 \cdots t_n} \rangle \varphi \leftrightarrow (Rt_1 \cdots t_n \wedge \varphi \wedge \langle Lp\mathbf{Rt_1 \cdots t_n} \rangle True)$

   La présupposition doit être vérifiée au même titre que la relation. Ceci a été implémenté de la manière suivante :

```
Axiom (Fprecexist (PR (a),phi),Fand(Fand ( FR (a),phi),c))
                                    →Lp (PR(a),b) ∧
                                      Axiom(Fprecexist (b,FTop),c).
```

2. $[\mathbf{Rt_1 \cdots t_n}]\varphi \leftrightarrow (Rt_1 \cdots t_n \rightarrow \varphi) \wedge \langle Lp\mathbf{Rt_1 \cdots t_n}\rangle True$

   De prime abord il peut paraître étrange que la présupposition soit traitée de la même manière pour une précondition universelle que pour une précondition existentielle. Le but est de vérifier que la présupposition est vérifiée. Si elle l'est alors on traite la relation présupposée en fonction de la précondition en vigueur. Si par contre la présupposition n'est pas vérifiée alors l'état obtenu doit être un état d'erreur. Ce sera bien le cas car comme la présupposition est traitée de la même façon dans les deux types de préconditions. Si la présupposition n'est pas vérifiée pour l'une, elle ne le sera pas non plus pour l'autre.

```
Axiom (Fprecexist (Pequal (t1,t2),phi),Fand(Fand (Fequal(t1,t2),phi),c))
                                          →Lp (Pequal(t1,t2),b) ∧
                                            Axiom(Fprecexist(b,FTop),c).
```

3. $\langle \mathbf{t_1} = \mathbf{t_2}\rangle\varphi \leftrightarrow (t_1 = t_2 \wedge \varphi) \wedge \langle Lp\mathbf{t_1} = \mathbf{t_2}\rangle True$

   Le cas des égalités de termes est similaire à la relation.

```
Axiom (Fprecuniv(PR(a),phi),Fand(Fimplic(FR(a),phi),c))
                                →Lp (PR(a),b) ∧
                                  Axiom(Fprecexist (b,FTop),c).
```

4. $[\mathbf{t_1} = \mathbf{t_2}]\varphi \leftrightarrow (t_1 = t_2 \rightarrow \varphi) \wedge \langle Lp\mathbf{t_1} = \mathbf{t_2}\rangle True$

```
Axiom (Fprecuniv(Pequal(t1,t2),phi),Fand(Fimplic(Fequal(t1,t2),phi),c))
                                      → Lp (Pequal(t1,t2),b) ∧
                                        Axiom(Fprecexist(b,FTop),c).
```

6.7 Conclusion

Nous avons présenté dans ce chapitre la sémantique axiomatique en général. Ensuite, nous avons introduit un langage logique du premier ordre (QDL) et une méthode de dérivation automatique de préconditions sur un programme DPL. Quelques exemples ont illustré l'utilité de cette méthode. La sous section suivante a présenté l'implémentation de ce chapitre et plus particulièrement la représentation du modèle comme un programme objet et du module évaluation comme un méta programme. Finalement, nous avons étendu le concept d'erreur à la présupposition lexicale.

Conclusion

Gödel

Le langage Gödel est toujours en cours de développement à l'Université de Bristol. Les principaux auteurs en sont Lloyd et Hill. La version que nous avons utilisée est la version 1.2. Celle-ci ne contient qu'un noyau du langage.

Dans un premier temps, l'approche déclarative et la notion typée nous ont paru plutôt lourdes à utiliser et Gödel nous semblait un langage destiné aux théoriciens et aux puristes. Après quelques temps, nous avons quitté la vision Prolog de la programmation logique et l'approche Gödel nous est apparue moins rébarbative. La notion de type s'est avérée très utile lors des compilations et nous a forcés à définir correctement et proprement les prédicats. Certaines erreurs auraient pris un certain temps pour être détectées en Prolog alors que Gödel les détecte automatiquement.

Un autre avantage, nous semble-t-il, d'un langage typé est de faciliter l'efficacité du code objet. Pour le moment, Gödel est écrit en partie en Prolog et en partie en C. La version actuelle ne prévoit aucune optimisation.

Un autre aspect intéressant de Gödel est la méta programmation. Dans le chapitre 6, nous avons implémenté de manière élégante l'évaluation d'une précondition par rapport à un modèle. En effet, le modèle et la précondition constituent des données. Le module d'évaluation traduit la précondition en un but et évalue ce but par rapport au modèle qui est défini sous forme de programme Gödel. En Prolog, nous aurions dû, dans un premier temps, ajouter les clauses du modèle, au module d'évaluation et, dans un second temps, évaluer la précondition par rapport au programme lui-même.

Finalement, les auteurs de Gödel insistent sur le fait que ce langage est plus déclaratif que Prolog. Gödel n'a plus de prédicat non logique. Ils sont, cependant, remplacés par d'autres mécanismes qui n'existent pas dans la logique des prédicats du premier ordre.

Le langage naturel

Un ordinateur est une machine polyvalente puisqu'il peut effectuer de nombreuses tâches différentes en fonction du programme qu'il exécute. Il peut, par exemple, rechercher l'adresse d'un client dans un fichier, produire une facture sur papier, effectuer des calculs scientifiques,... Le programme contient un ensemble d'instructions que l'ordinateur exécute pas à pas et qui lui indiquent ce qu'il doit faire. Ces instructions sont écrites en langage machine que l'ordinateur comprend. Elles sont du type: copier la valeur d'un registre vers un autre registre, additionner le contenu du registre A avec celui du registre B, ... L'ordinateur ne comprend que ce langage et y est adapté.

L'être humain, par contre, éprouve de grandes difficultés à le comprendre et utilise des compilateurs pour traduire un langage de programmation évolué dans le code machine. La construction d'un compilateur impose que le langage de programmation soit complètement défini et sans ambiguïté. En effet, le processus de traduction doit être décrit par un algorithme.

Le langage naturel n'est pas compris directement par l'ordinateur. Il est donc nécessaire de le traduire en un programme formel compris par l'ordinateur. Le langage naturel n'est évidemment pas un langage de programmation et est bien plus complexe. De plus, contrairement à un langage de programmation, il n'a aucun fondement formel car il ne correspond pas à des concepts bien définis. Il fait référence à la personnalité de chaque être humain. Comment définir l'âme ou l'amour en termes mathématiques ? Quel sens donner à la phrase "ce poème est beau" ? Certes une traduction complète du langage naturel en un langage formel n'est pas possible. Cependant, elle peut être intéressante pour une approche différente. Une phrase telle que "Quelle est l'adresse de Jean ?" peut être interpétée par l'ordinateur comme une requête à une base de données. Une autre phrase comme "Jean habite à Namur" peut être interprétée par l'ordinateur comme une affirmation à conserver dans la base de données.

Tout comme un langage de programmation qui est traduit en code machine, notre approche consiste en deux phases. Elle analyse d'abord la structure du texte source et produit ensuite un code objet.

Pour effectuer l'analyse du texte source, il est nécessaire de définir les règles syntaxiques. Un procédé consiste à donner un ensemble de couples de mots et catégories et de donner les règles de composition des catégories pour former une catégorie plus complexe. Ce procédé correspond à la notion de grammaire formelle. Dans le cadre de ce mémoire, nous avons présenté les grammaires catégorielles (cfr chapitre 3). La définition d'une grammaire catégorielle pour le langage naturel n'est pas une tâche aisée. Les phrases peuvent prendre plusieurs formes (active, passive, interrogative, négative, ...); les règles d'accord sont nombreuses et contiennent des exceptions, il y a de nombreux types de compléments, des éléments sont facultatifs ou n'ont pas de places fixes, ... Pour simplifier la tâche, nous nous sommes limités à un vocabulaire restreint et à des structures de phrases simples. Une des caractéristiques d'une bonne grammaire selon [Allen 1987] est sa capacité d'être facilement étendue. Dans ce mémoire, nous avons défini un premier fragment du français que nous avons par la suite complété pour introduire la gestion des accords. Le programme d'analyse syntaxique n'a subi aucune modification. La grammaire

constitue une donnée du programme d'analyse, tout comme la phrase à analyser. Pour preuve, le programme peut analyser une phrase française ou anglaise en fonction de la grammaire "linkée".

La seconde étape consiste à traduire une phrase dans un code compréhensible par la machine. La logique des prédicats dynamiques permet de manipuler des états. Elle permet soit d'ajouter des variables à cet état, soit de vérifier si un état vérifie une condition. La logique des prédicats dynamiques avec état d'erreur défini dans [van Eijck 1991] permet de traiter de manière élégante le problème de l'assignation définie du langage. L'introduction d'un état d'erreur permet en outre de gérer le cas où l'unicité n'est pas vérifiée.

Le principe de la traduction repose sur le λ-calcul qui est couramment utilisé en linguistique automatique. Il offre une capacité d'extension du fragment. En effet, pour ajouter de nouveaux mots, il suffit d'introduire le mot, sa catégorie et la λ-expression associée. Le programme construit dans le cadre du mémoire peut, en fonction du lexique, traduire des phrases anglaises ou françaises.

L'étape suivante consiste à dériver de ce programme deux types de préconditions en langage logique du premier ordre. La première exprime la condition qu'un état doit vérifier pour que la phrase soit juste. Par "juste" nous entendons, que la phrase ait un sens, et que l'exécution du programme avec un tel état d'entrée réussisse. La seconde exprime la condition qu'un état doit vérifier pour que la phrase soit fausse. Par "fausse" nous entendons que la phrase ait un sens et que l'exécution du programme avec un tel état d'entrée échoue. La composition des négations de ces deux expressions est la condition d'état d'erreur. En effet, si un état ne vérifie ni l'échec ni la réussite du programme, la phrase n'a pas de sens. C'est ce qu'exprime un état d'erreur.

La dernière étape consiste à évaluer ces conditions par rapport à un modèle. La solution que nous avons choisie est de représenter le programme d'évaluation comme une méta programme qui manipule un programme Gödel qui lui représente le modèle. L'évaluation consiste d'abord à traduire la condition exprimée en Quantified Dynamic Logic en un but Gödel. Ensuite, ce but est évalué au sein du programme objet dit "modèle".

Ces deux dernières étapes, mis à part la représentation du modèle, sont indépendantes du langage naturel utilisé. L'interprétation a été affinée par la suite en utilisant le concept de présupposition lexicale (Cfr. sous-section 6.6.2).

Perspective

Le mémoire actuel offre de nombreuses perspectives de développements ultérieurs. La grammaire peut être largement étendue. Celle-ci illustre simplement les aspects théoriques des grammaires catégorielles. De nombreux aspects peuvent être développés et notamment l'introduction des temps pour les verbes. Les aspects syntaxiques ne constituent pas le centre du mémoire.

L'aspect original et novateur, du programme que nous avons développé au C.W.I. réside dans le traitement sémantique. Les perspectives de développements se situent au niveau du traitement

des présuppositions. Le concept peut notamment être étendu à la notion de séquences d'actions. Cela consiste à vérifier qu'une action se déroule avant une autre. Par exemple, l'action "Jean commence à marcher" se déroule avant "Jean arrête de marcher".

Bibliographie

[Allen 1987] Allen, J. *Natural Language Understanding*

[Chambreuil 1989] Chambreuil, M. (1989), *Grammaire de Montague: Langage, Traduction, Interprétation*, Adosa, Clermont-Ferrand (France)

[Dahl et Saint-Dizier 1984] Dahl, V. et Saint-Dizier, P. (1984), *Natural Language Understanding and Logic Programming*, North Holland, Rennes (France)

[EACL 1993] Bouchez, O., van Eijck, D.J.N., Istace, O. (1993), "A Strategy for Dynamic Interpretation: A Fragment and an Implementation", *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht (Pays-Bas)

[Grevisse] Grevisse, M. (1969), *Précis de grammaire française*, DUCULOT, Gembloux (Belgique)

[Grosz, Sparck et Webber 1986] Grosz, Sparck et Webber (1986), *Readings in Natural Language Processing*, Morgan Kaufmann Publishers, U.S.A.

[King 1983] King, M. (1983), *Parsing Natural Language*, Academic Press, Lugano, Switzerland

[Lloyd 1987] Lloyd, J.W. (1987), *The Foundations of Logic Programming*, Springer-Verlag

[Hill & Lloyd 1992] Hill, P.M. , Lloyd, J.W. (1992), *The Gödel Programming Language*, University of Bristol, (Royaume-Uni)

[Muskens 1990] Muskens, R. (1990), "Anaphora and the logic of change.", In J.van Eijck, editor, *Logics in AI / European Workshop JELIA '90 / Amsterdam, The Netherlands, September 1990 / Proceedings*, Lecture Notes in Artificial Intelligence 478, pages 412–427. Springer Verlag, 1991

[Muskens 1992] Muskens, R. (1992), "Tense and the logic of change.", Manuscript, University of Tilburg

[Pereira 1987] Pereira, F. C. N. (1987), *Prolog and Natural Language Analysis*, CSLI, Stanford University (Etats-Unis)

[Pratt 1976] Pratt, V. (1976), *Semantical considerations on Floyd–Hoare logic*, Proceedings 17th IEEE Symposium on Foundations of Computer Science, pages 109–121

[Shapiro 1986] Sterling, L., Shapiro, E. (1986), *The Art of Prolog*, MIT Press, Cambridge (Etats-Unis)

[Stoy 1977] Stoy, J. (1977), *Denotational Semantics: chapitre 5 The λ-calculus*, MIT press

[van Eijck 1991] van Eijck, D.J.N. (1991), "The dynamics of description", *CWI Report*, CWI, Amsterdam (Pays-Bas)

[van Eijck 1992] van Eijck, D. J. N. and de Vries, F. J. (1992), "Dynamic interpretation and Hoare deduction", *Journal of Logic, Language, and Information*, 1:1–44

[van Eijck 1993] van Eijck, J. (1993), "Presupposition failure — a comedy of errors." Manuscript, CWI, Amsterdam

[Yoo & Lee] Yoo, S., Lee, K. "Extended Categorial Grammar", CSLI, Stanford University (Etats-Unis)

[Weizenbaum 1981] Weizenbaum, J. (1981), *Puissance de l'ordinateur et raison de l'homme : du jugement au calcul*, Ed. d'informatique, Boulogne (France)

Annexe A

Description de l'implémentation

A.1 Introduction

Le programme développé au CWI à Amsterdam est écrit en Gödel et met en oeuvre un traitement du langage naturel tel que présenté dans le mémoire. L'objectif de cet implémentation est d'une part de mettre en pratique les théories exposées et d'autre part d'expérimenter le langage Gödel.

Le listing que nous donnons en annexe constitue le programme dans sa version final brute. Pour aider à la compréhension, nous allons tout d'abord introduire les éléments importants de Gödel. Ensuite nous présenterons la structure générale du programme.

A.2 Gödel

Gödel est un langage modulaire. Un programme est constitué d'un ensemble de modules. Un module est divisé en deux parties. Une première partie dont l'entête est `MODULE` suivi du nom du module contient les déclarations internes au module et la définition des prédicats. La deuxième partie, facultative, dont l'entête est `EXPORT` contient les déclarations utilisables à l'extérieur du module. L'entête correspond à la première ligne de code du programme. Les lignes commançant par % sont des commentaires.

Un module est divisé en section.

- La section `IMPORT` contient le nom des modules importés par ce module.
- La section `BASE` contient la déclaration des types utilisés dans le module.
- La section `CONSTANT` contient la déclaration des constantes et de leur type.

- La section `FUNCTION` contient la déclaration des fonctions, du type de ces arguments, de la priorité , de la position de l'opérateur (pré, post, ou infixé), du type du résultat.
- La section `PREDICATE` contient la déclaration des prédicats et du type des arguments.

Inversément pour les habitués de Prolog, les variables commencent par une minuscule. Les types, les constantes, les fonctions et les prédicats commencent par une majuscule.

Le symbole & représente une conjonction et le symbole $<-$ représente l'implication.

A.3 Structure

Le programme est structuré en modules.

A.3.1 Main

`Main` est le module principal. Il contient la définition de trois prédicats:

- `Translation` produit à partir d'une liste de mots, une liste de catégories correspondant à la catégorie de la liste de mots, une liste d'arbres qui correspond à l'analyse de la liste de mots, et un program DPL qui correspond à la représentation de la sémantique de la suite de mots. La catégorie d'un texte est une liste de S (S est la catégorie d'une phrase) et l'analyse d'un texte est une forêt d'arbre d'analyse de chacune des phrases.
- `Traduction` produit à partir d'un nom de fichier et d'une liste de mots, un document latex qui contient les résultats du prédicat `Translation`
- `Compilation` produit à partir d'un nom de fichier, d'un nom de modèle et d'une liste de mots, un fichier latex qui contient en plus de `Traduction` les préconditions et l'évaluation de ces précondition par rapport à un modèle.

A.3.2 Categor

Le module `Categor` gère l'analyse d'un arbre par rapport à n'importe quelle grammaire catégorielle contenue dans le dictionnaire.

Il définit la représentation d'un arbre au moyen de fonctions et de constantes.

`Info` est un type qui représente une information. `I1` est une fonction qui reçoit comme argument un nom de catégorie et produit une information. `I2` est une fonction qui reçoit comme argument un nom de catégorie et une liste de mots et qui produit une information.

`Tree` est un type de donnée qui correspond à un arbre. `Empty` représente un arbre vide. `A` est une fonction qui reçoit trois arguments une information, un arbre gauche et un arbre droit et qui produit un arbre.

Le prédicat `Category` permet d'analyser une liste de mots et de donner sa catégorie et son arbre d'analyse. Les arguments sont une liste de mots, une liste de catégories et une liste d'arbres.

A.3.3 Lambda

Le module `Lambda` définit la traduction d'une liste d'arbre en un programme au moyen du prédicat `Trans` . Il contient un réducteur de λ-expression.

A.3.4 Axioms

Le module `Axioms` permet de dériver les préconditions d'un programme DPL. Il définit la représentation des fonctions QDL.

Le prédicat `PureProgram` permet de vérifier si un programme ne contient aucune λ-expression résiduelle de la traduction. Le prédicat `TermVL` vérifie si un terme est une variable λ. Le prédicat `Axiom` permet à partir d'une formule décrivant une précondition, de calculer cette précondition.

A.3.5 Simple

Le prédicat `Simplification` permet de simplifier quelque peu une formule en utilisant des équivalences logiques triviales (cfr. 6.5.2)

A.3.6 Eval

Le module `Eval` permet de vérifier une formule dans un modèle. Le premier argument du prédicat `Test` est le nom du modèle. Le second argument est la formule QDL. Le troisième argument est soit YES ou NO et répond à question "La formule est-elle vérifiée dans le modèle ?"

A.3.7 IOPrgs

Le module `IOPrgs` gère les sorties dans un fichier texte en format latex.

Le prédicat `WriteStringslash` permet d'écrire une chaîne de caractère commançant par le symbole \.

Le prédicat `OutputListWord` permet d'écrire un liste de mots dans un fichier.

Le prédicat `OutputFormula` permet d'écrire une formule QDL dans un fichier.

Le prédicat `OutputProgram` permet d'écrire un programme DPL dans un fichier.

Le prédicat `Depth` calcule la profondeur d'un arbre.

Le prédicat `Breadth` calcule la largeur d'un arbre.

Le prédicat `Writenumbers` permet d'écrire des nombres dans un fichier.

Le prédicat `OutputTree` permet d'écrire un arbre dans un fichier.

Le prédicat `OutputTrees` permet d'écrire une suite d'arbres dans un fichier.

A.3.8 Dico

Le module `Dico` contient tous les éléments relatifs au langage. Nous l'avons également utilisé pour y placer tous les types utilisés par plusieurs modules. C'est pourquoi il contient la déclaration du type `Program` qui représente un programme DPL et `Expr` qui représente une λ-expression.

Le prédicat `Dicl` définit le lexique. Son premier argument est une liste de mots, le second la catégorie de cette liste de mots et le dernier la λ-expression correspondante. Nous avons introduit des listes de mots dans le lexique pour gérer le problème des particules. Par exemple le verbe "sit down" est défini. Par contre "sit up" n'a aucun sens.

Le prédicat `Lp` définit les préssupositions lexicales.

Le prédicat `Syncat` définit les catégories en terme de catégories de base.

Le prédicat `Text` vérifie qu'une liste de catégories est une suite de phrases.

A.4 Main

```
% Implementation of a little fragment of natural language
%
% Author : Olivier Bouchez & Olivier Istace.
% Date   : October 1992
%
% The goal of this program is to translate simple sentences
% of English into dynamic assignment logic.
%
% -Declaration Part ------------------------------------------------------

MODULE Main.

IMPORT
Lists,Strings,IO,Dico,IOPrgs,Axioms,Lambda,Categor,Simple,Eval.
% ---------------------------------------------


PREDICATE
              Translation : List(Word) * List(Cat) *
                            List(Tree) * Program;

              Traduction  : String * List(Word).



PREDICATE Compilation     : String       *String * List(Word).


% - Definition Part ------------------------------------------------------

% --------------------T R A N S L A T I O N--------------------------

Translation (a,b,c,d) <- Category (a,b,c) & Trans (c,d).


% -------------------- T R A D U C T I O N ---------------------------

Traduction(n,s) <- FindOutput (n,Out(stream)) &
                   WriteStringslash(stream,
                          "documentstyle[11pt]{article}") &
                   NewLine(stream) & NewLine(stream)  &
                   WriteStringslash(stream,
```

```
                    "begin{document}") &
                NewLine(stream) & NewLine(stream)  &
                WriteString (stream,"Sentence:") &
                OutputListWord(stream,s) &
                NewLine(stream) & NewLine(stream)  &
                Translation (s,_,a,p) &
                OutputTrees (stream,a) &
                NewLine(stream)  & NewLine(stream)  &
                WriteString (stream,"Program: ") &
                NewLine(stream) & NewLine(stream)  &
                OutputProgram (stream,p) &
                WriteStringslash (stream,"end{document}") &
                NewLine(stream) & NewLine(stream)  &
                EndOutput(stream) .

%-------------------------------------------------
Compilation(n,model,s) <- FindOutput (n,Out(stream)) &
                WriteStringslash(stream,
                        "documentstyle[11pt]{article}") &
                NewLine(stream) & NewLine(stream)  &
                WriteStringslash(stream,"begin{document}") &
                NewLine(stream) & NewLine(stream)  &
                WriteString (stream,"Sentence:") &
                OutputListWord(stream,s) &
                NewLine(stream) & NewLine(stream)  &
                Translation (s,_,a,p) &
                OutputTrees (stream,a) &
                NewLine(stream)  & NewLine(stream)  &
                WriteString (stream,"Program: ") &
                NewLine(stream) & NewLine(stream)  &
                OutputProgram (stream,p) &
                NewLine(stream) & NewLine(stream)  &
                Axiom (Fprecexist(p,FTop),pe) &
                WriteString (stream,
                        "Existential precondition of success:") &
                NewLine(stream) &
                NewLine(stream) &
  WriteString(stream,"without simplification") &
                NewLine(stream) & NewLine(stream) &
                OutputFormula(stream,pe) &
                NewLine(stream) & NewLine(stream) &
  WriteString(stream,"with simplification") &
                NewLine(stream) & NewLine(stream)  &
                Simplification(pe,pes) &
                OutputFormula (stream,pes) &
```

```
                NewLine(stream) & NewLine(stream)  &
                Test(model,pes,respes) &
                Axiom (Fprecuniv(p,FBottom),pu) &
                WriteString (stream,
                       "Universal precondition of failure:") &
                NewLine(stream) & NewLine(stream) &
 WriteString(stream,"without simplification") &
                NewLine(stream) & NewLine(stream) &
                OutputFormula(stream,pu) &
                NewLine(stream) & NewLine(stream) &
 WriteString(stream,"with simplification") &
                NewLine(stream) & NewLine(stream)  &
                Simplification(pu,pus) &
                OutputFormula (stream,pus) &
                NewLine(stream) & NewLine(stream)  &
                Test(model,pus,respus) &
                Axiom (Fpar(p,FE),per) &
                WriteString (stream,"Precondition of error:") &
                NewLine(stream) & NewLine(stream) &
 WriteString(stream,"without simplification") &
                NewLine(stream) & NewLine(stream) &
                OutputFormula(stream,per) &
                NewLine(stream) & NewLine(stream) &
 WriteString(stream,"with simplification") &
                NewLine(stream) & NewLine(stream)  &
                Simplification(per,pers) &
                OutputFormula (stream,pers) &
                NewLine(stream) & NewLine(stream)  &
                WriteString(stream,"Result of the evaluation of ") &
                WriteString(stream,model) &
                WriteString(stream,": ") &
                (IF (respes = "YES") THEN
                (WriteString(stream,"Success."))
                ELSE
                ((IF ((respus = "YES")) THEN
                 (WriteString(stream,"Failure."))
                ELSE (WriteString(stream,"Error."))))) &
                WriteStringslash (stream,"end{document}") &
                NewLine(stream) & NewLine(stream)  &
                EndOutput(stream) .



Compilation(n,_,s) <- FindOutput (n,Out(stream)) &
```

```
WriteStringslash(stream,
        "documentstyle[11pt]{article}") &
NewLine(stream) & NewLine(stream)  &
WriteStringslash(stream,"begin{document}") &
NewLine(stream) & NewLine(stream)  &
WriteString (stream,"Sentence:") &
OutputListWord(stream,s) &
NewLine(stream) & NewLine(stream)  &
~Category (s,_,_) &
WriteString (stream,
       "This sentence seems not correct") &
NewLine(stream) & NewLine(stream)  &
WriteStringslash (stream,"end{document}") &
EndOutput(stream).
```

A.5 Categor

Export

```
EXPORT Categor.

IMPORT Dico.

BASE
              Tree,Info.


% Empty represents an empty tree.

CONSTANT
             Empty : Tree.


% Function to represent Trees
% A tree contains two kind of information
%   I2  -  A Word and its category
%   I1  -  The category of two descendant trees
% A tree is defined by an information, a left subtree and a right subtree.

FUNCTION      A  : Info * Tree * Tree -> Tree;
```

```
            I1 : Cat                -> Info;
            I2 : Cat * List (Word)  -> Info.

PREDICATE
         Category : List (Word) * List (Cat) * List(Tree).
```

Local

```
LOCAL Categor.

PREDICATE
              Red    : List(Cat) * List(Cat).


% The old predicates are the same as the new ones but they
% don't build a parsing tree.

PREDICATE

              Lcatold      : List (Word) * List (Cat);
              Categoryold  : List (Word) * List (Cat);
              Redallold    : List(Cat)   * List(Cat);
              Red2old      : List(Cat)   * List(Cat);
              Catappendold : List(Word)  * Cat.

% The predicate Lcat creates the list of categories corresponding
% to each word of a sentence and the list of trees containing the
% I2(word , category).
%
% The predicate Category computes the category of a sentence
% and builds its parsed tree with a bottom-up method.
%
% The predicate Red2
%              - finds in a list of categories two categories
%                that may be reduce
%              - returns  the list of the reduction
%              - modifies the list of trees with the I1 of the reduction.
%
% The predicate Redall reduces until no possible a list of categories.
%
% The predicate cattappend computes like Category the category and the
% parsed tree of a sentence, but it uses a top-down method.
```

```
%
PREDICATE
        Lcat      : List (Word) * List (Cat) * List(Tree);
        Red2      : List(Cat)   * List(Cat)  * List(Tree) * List(Tree);
        Redall    : List(Cat)   * List(Cat)  * List(Tree) * List(Tree);
        Catappend : List(Word)  * Cat        * Tree.



% -------------------------R E D-------------------------------------
Red ([a/b,b],[a]).
Red ([a,a\b],[b]).
% ----------------------R E D 2 0 L D-------------------------------
Red2old ([m1,m2|r],[m|r]) <- Red ([m1,m2],[m]).
Red2old ([m1|r],[m1|q]) <- Red2old (r,q).

% -------------------------R E D 2-------------------------------------
Red2 ([m1,m2|r],[m|r],[d1,d2|s],[A(I1(sm),d1,d2)|s]) <-
                                Red ([m1,m2],[m])
                                & Syncat (sm,m).

Red2 ([m1|r],[m1|q],[d|s],[d|t]) <-  Red2 (r,q,s,t).

% ----------------------R E D A L L 0 L D---------------------------
Redallold (m,r) <-  Red2old (m,q) & Redallold (q,r).
Redallold (m,m) <-  ~Red2old (m,_).

% -------------------------R E D A L L-------------------------------
Redall (m,r,d,f) <-   Red2 (m,q,d,g) & Redall (q,r,g,f).
Redall (m,m,d,d) <-   ~Red2(m,_,d,_).
% -----------------C A T E G 0 R Y 0 L D-----------------------------
Categoryold (l,c) <- Lcatold (l,b) & Redallold (b,c).
Category (l,c,f) <- Lcat (l,b,d) &  Redall (b,c,d,f) & Text (c).
% ----------------------L C A T 0 L D-------------------------------
Lcatold (p,[c]) <- Dicl(p,c1,_) & Syncat(c1,c).

Lcatold (p,[a|b]) <- Append (d,f,p) &
             d ~= [] &
             f ~= [] &
             Dicl (d,a1,_) &
             Syncat (a1,a) &
             Lcatold (f,b).

% --------------------------L C A T-------------------------------------
Lcat(p,[c],[A(I2(c1,p),Empty,Empty)]) <- Dicl(p,c1,_) &
```

```
                                        Syncat(c1,c).

Lcat (p,[a|b],[A(I2(a1,d),Empty,Empty)|g]) <-
                Append (d,f,p) &
                d ~= [] &
                f ~= [] &
                Dicl (d,a1,_) &
                Syncat (a1,a) &
                Lcat (f,b,g).


%----------------------C A T A P P E N D O L D-------------------------

Catappendold(a,b) <- Dicl (a,c,_) & Syncat (c,b).

Catappendold(p,c) <-  Append (d,f,p) &
                      d ~= [] &
                      f ~= [] &
                      Catappendold (d,a) &
                      Catappendold (f,b) &
                      Red ([a,b],[c]) .
%                      Syncat (e,c).
%----------------------C A T A P P E N D-----------------------------

Catappend(a,b,A(I2(c,a),Empty,Empty)) <- Dicl (a,c,_) &
                                         Syncat (c,b).

Catappend(p,c,A(I1(e),t_a,t_b)) <-  Append (d,f,p) &
                                    d ~= [] &
                                    f ~= [] &
                                    Catappend (d,a,t_a) &
                                    Catappend (f,b,t_b) &
                                    Red ([a,b],[c]) &
                                    Syncat (e,c).
```

A.6 Lambda

Export

```
EXPORT Lambda.
```

```
IMPORT Dico,Lists,Categor,Axioms.

PREDICATE
              Trans       : List(Tree) * Program.
```

Local

```
LOCAL Lambda.

% The predicate Trad translates a parsed tree into a Lambda expression
% that represents a DAL program.
%
% The predicate Translation is the heart of the program. It
$               - translates a representation of a sentence into
%                 the representation of its DPL program
%               - builds the parsed tree
%               - computes its category.
%
% The predicate Listtoappl transforms a list of lambda expression into
% a list containing only one element which has the same meaning.


PREDICATE
       Trad         : Tree       * Expr;
              Listtoappl  : List(Expr) * List(Expr).

% The lambda reduction predicates.
% Predicate Occfree tests wether a variable is free in an expression.
% Predicate Subst applies the substitution of a variable by an expression
% into an expression.
% Predicate Nextfree looks for the next free variable into an expression.
% Predicate Redl defines the reduction rules of lambda expression.
% Canred finds the most reducible form of a lambda expression.

PREDICATE
              TermVle  : Term    * Integer;
              Occfree  : Integer * Expr;
              Subst    : Integer * Expr * Expr * Expr;
              LSubst   : Integer * Expr * List(Term) * List(Term);
              Nextnotfree: Integer * Expr * Integer;
              Redl     : Expr    * Expr;
              Canred   : Expr    * Expr.
```

```
% -------------------------- T R A N S -------------------------------

Trans ([c],p) <- Trad (c,P(p)).
Trans ([h|t],Papplic(p1,p2)) <- Trad (h,P(p1)) & Trans (t,p2).
% --------------------------T R A D---------------------------------

Trad (A(I2(b,a),Empty,Empty),l) <- Dicl (a,b,l).


Trad (A(I1(ca),A(I1(csag),sag1,sad1),
             A(I1(csad),sag2,sad2)),l)
                      <-    Syncat(ca,a) &
                            Syncat(csag,a/b) &
                            Syncat(csad,b) &
      Trad (A(I1(csag),sag1,sad1),la) &
                     Trad (A(I1(csad),sag2,sad2),lb) &
                            Canred (A(la,lb),l).

Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
             A(I2(csad,nsad),sag2,sad2)),l)
                      <-    Syncat(ca,a) &
                            Syncat(csag,a/b) &
                            Syncat(csad,b) &
                            Trad (A(I2(csag,nsag),sag1,sad1),la) &
                     Trad (A(I2(csad,nsad),sag2,sad2),lb) &
                            Canred (A(la,lb),l).

Trad (A(I1(ca),A(I1(csag),sag1,sad1),
             A(I2(csad,nsad),sag2,sad2)),l)
                      <-    Syncat(ca,a) &
                            Syncat(csag,a/b) &
                            Syncat(csad,b) &
      Trad (A(I1(csag),sag1,sad1),la) &
                     Trad (A(I2(csad,nsad),sag2,sad2),lb) &
                            Canred (A(la,lb),l).

Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
             A(I1(csad),sag2,sad2)),l)
                      <-    Syncat(ca,a) &
                            Syncat(csag,a/b) &
                            Syncat(csad,b) &
                            Trad (A(I2(csag,nsag),sag1,sad1),la) &
                     Trad (A(I1(csad),sag2,sad2),lb) &
```

```
                              Canred (A(la,lb),l).

Trad (A(I1(ca),A(I1(csag),sag1,sad1),
             A(I1(csad),sag2,sad2)),l)
                        <-    Syncat(ca,a) &
                              Syncat(csag,b) &
                              Syncat(csad,b\a) &
      Trad (A(I1(csag),sag1,sad1),la) &
                        Trad (A(I1(csad),sag2,sad2),lb) &
                              Canred (A(lb,la),l).

Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
             A(I2(csad,nsad),sag2,sad2)),l)
                        <-    Syncat(ca,a) &
                              Syncat(csag,b) &
                              Syncat(csad,b\a) &
                              Trad (A(I2(csag,nsag),sag1,sad1),la) &
                        Trad (A(I2(csad,nsad),sag2,sad2),lb) &
                              Canred (A(lb,la),l).

Trad (A(I1(ca),A(I1(csag),sag1,sad1),
             A(I2(csad,nsad),sag2,sad2)),l)
                        <-    Syncat(ca,a) &
                              Syncat(csag,b) &
                              Syncat(csad,b\a) &
      Trad (A(I1(csag),sag1,sad1),la) &
                        Trad (A(I2(csad,nsad),sag2,sad2),lb) &
                              Canred (A(lb,la),l).

Trad (A(I1(ca),A(I2(csag,nsag),sag1,sad1),
             A(I1(csad),sag2,sad2)),l)
                        <-    Syncat(ca,a) &
                              Syncat(csag,b) &
                              Syncat(csad,b\a) &
                              Trad (A(I2(csag,nsag),sag1,sad1),la) &
                        Trad (A(I1(csad),sag2,sad2),lb) &
                              Canred (A(lb,la),l).

% ----------------------L I S T T O A P P L----------------------------
Listtoappl ([x],[x]).
Listtoappl ([a,b],[A(a,b)]).
Listtoappl ([a,b|r],[A(A(a,b),s)]) <- Listtoappl(r,[s]).
% ------------------------O C C F R E E ----------------------------

Occfree (x,VE(Vl(x))).
```

```
Occfree (x,A(y,_)) <- Occfree (x,y).
Occfree (x,A(_,z)) <- Occfree (x,z).
Occfree (x,L(y,z)) <- x ~= y &
                      Occfree (x,z).

Occfree (x,P(PR(Rel(_,[Var(Vl(x))])))).
Occfree (x,P(PR(Rel(_,[Var(Vl(x))|_])))).
Occfree (x,P(PR(Rel(n,[_|t])))) <- Occfree (x,P(PR(Rel(n,t)))).

Occfree (x,P(Pequal(Var(Vl(x)),_))).
Occfree (x,P(Pequal(Var(_),Var(Vl(x))))).

Occfree (x,P(Pnot(p))) <- Occfree (x,P(p)).
Occfree (x,P(Pimplic(p1,_))) <- Occfree (x,P(p1)).
Occfree (x,P(Pimplic(_,p2))) <- Occfree (x,P(p2)).
Occfree (x,P(Papplic(p1,_))) <- Occfree (x,P(p1)).
Occfree (x,P(Papplic(_,p2))) <- Occfree (x,P(p2)).
Occfree (x,P(Piota(_,p))) <- Occfree (x,P(p)).
Occfree (x,P(Piota(Vl(x),_))).
Occfree (x,P(Peta(Vl(x),_))).
Occfree (x,P(Peta(_,p))) <- Occfree (x,P(p)).
Occfree (x,P(LE(e))) <- Occfree (x,e).
% -------------------------S U B S T---------------------------------

Subst (px,gm,VE(Vl(px)),gm).
Subst (px,_,VE(Vl(gxp)),VE(Vl(gxp))) <- px ~= gxp.
Subst (px,gm,A(gy,gz),A(gxp1,gxp2)) <-
                          Subst (px,gm,gy,gxp1) &
                          Subst (px,gm,gz,gxp2).
Subst (px,_,L (px,gxp),L(px,gxp)).
Subst (px,gm,L (py,gy),L(py,gxp)) <-
                           px ~= py &
                           ~Occfree(px,gy) &
                           Subst (px,gm,gy,gxp).
Subst (px,gm,L (py,gy),L (py,gxp)) <-
                            px~= py &
                            ~Occfree(py,gm) &
                            Subst (px,gm,gy,gxp).
Subst (px,gm,L (py,gy),gxp) <-
                             Occfree (px,gy) &
                             Occfree(py,gm) &
                             Nextnotfree (px,gm,pz) &
                             gxp = L (pz,c) &
                             Subst (px,gm,b,c) &
                             Subst (py,VE(Vl(pz)),gy,b).
```

```
Subst (_,_,VE(V(gxp)),VE(V(gxp))).
Subst (_,_,P(PBottom),P(PBottom)).
Subst (_,_,P(PTop),P(PTop)).
Subst (px,gm,P(Pnot(p)), P(Pnot(pp))) <- Subst (px,gm,P(p),P(pp)).
Subst (px,gm,P(Papplic(p1,p2)), P(Papplic(p1p,p2p)))
                <- Subst (px,gm,P(p1),P(p1p))
                   & Subst (px,gm,P(p2),P(p2p)).
Subst (px,gm,P(Pimplic(p1,p2)), P(Pimplic(p1p,p2p)))
                <- Subst (px,gm,P(p1),P(p1p))
                   & Subst (px,gm,P(p2),P(p2p)).
Subst (px,gm,P(PR(Rel(m,l))),P(PR(Rel(m,lp))))
                <- LSubst (px,gm,l,lp).

Subst (px,VE(gm),P(Pequal(Var(Vl(px)),t2)), P(Pequal(Var(gm),t2))).

Subst (px,VE(gm),P(Pequal(t1,Var(Vl(px)))), P(Pequal(t1,Var(gm)))).


Subst (px,_,P(Pequal(t1,t2)),
       P(Pequal(t1,t2)))
                <- ~TermVle(t1,px) & ~TermVle(t2,px).

Subst (_,_,P(Pequal(t1,t2)),P(Pequal(t1,t2)))
             <- ~TermVl(t1) & ~TermVl(t2).

Subst (px,VE(gm),P(Peta(Vl(px),p)),P(Peta(gm,pp)))
             <- Subst (px,VE(gm),P(p),P(pp)).

Subst (px,gm,P(Peta(Vl(py),p)),P(Peta(Vl(py),pp)))
             <- py ~= px & Subst (px,gm,P(p),P(pp)).
Subst (px,gm,P(Peta(V(py),p)),P(Peta(V(py),pp)))
             <- Subst (px,gm,P(p),P(pp)).

Subst (px,VE(gm),P(Piota(Vl(px),p)),P(Piota(gm,pp)))
             <- Subst (px,VE(gm),P(p),P(pp)).

Subst (px,gm,P(Piota(Vl(py),p)),P(Piota(Vl(py),pp)))
             <- py ~= px & Subst (px,gm,P(p),P(pp)).

Subst (px,gm,P(Piota(V(py),p)),P(Piota(V(py),pp)))
             <- Subst (px,gm,P(p),P(pp)).

Subst (px,gm,P(LE(x)),P(LE(xp))) <- Subst (px,gm,x,xp).
```

```
% ---------------- L S U B S T -------------------------------------

LSubst (_,_,[],[]).
LSubst (px,VE(gm),[Var(Vl(px))|t],[Var(gm)|tp])
                        <- LSubst (px,VE(gm),t,tp).
LSubst (px,gm,[Var(Vl(py))|t],[Var(Vl(py))|tp])
                        <- py ~= px & LSubst (px,gm,t,tp).

LSubst (px,gm,[t1|t],[t1|tp])
                        <- ~TermVl(t1) & LSubst (px,gm,t,tp).
% ------------------------- T E R M V L ----------------------------

TermVle (Var(Vl(x)),x).

% -------------------------N E X T F R E E-------------------------

Nextnotfree (i,e,i) <- ~Occfree (i,e).
Nextnotfree (i,e,j) <- Occfree (i,e) &
                       k = i + 1 &
                       Nextnotfree (k,e,j).

% ----------------------------R E D L--------------------------------
Redl (A(L(px,gm),A(g1,g2)), A(res,g2)) <- Subst (px,g1,gm,res).
Redl (A(L(px,gm),gn),res) <- Subst (px,gn,gm,res).
Redl (L(px,A(gm,VE(Vl(px)))),gm) <- ~Occfree (px,gm).
%Redl (A(A(a,b),c),A(res,c)) <- Redl(A(a,b),res).
%Redl (A(c,A(a,b)),A(c,res)) <- Redl(A(a,b),res).
Redl (L(x,l),L(x,lp)) <- Redl (l,lp).
Redl (A(A(c,L(a,b)),m),A(c,mp)) <- Redl(A(L(a,b),m),mp).
Redl (A(A(c,A(a,b)),m),A(c,mp)) <- Redl(A(A(a,b),m),mp).

Redl (P(Pnot(p)),P(Pnot(pp))) <- Redl(P(p),P(pp)).
Redl (P(Papplic(p1,p2)),P(Papplic(p1p,p2))) <- Redl(P(p1),P(p1p)).
Redl (P(Pimplic(p1,p2)),P(Pimplic(p1p,p2))) <- Redl(P(p1),P(p1p)).
Redl (P(Papplic(p1,p2)),P(Papplic(p1,p2p))) <- Redl(P(p2),P(p2p)).
Redl (P(Pimplic(p1,p2)),P(Pimplic(p1,p2p))) <- Redl(P(p2),P(p2p)).
Redl (P(Peta(v,p)),P(Peta(v,pp))) <- Redl (P(p),P(pp)).
Redl (P(Piota(v,p)),P(Piota(v,pp))) <- Redl (P(p),P(pp)).
Redl (P(LE(P(x))),P(x)).
Redl (P(LE(e)),P(LE(ep))) <- Redl (e,ep).

Redl (A(e1,e2), A(e1p,e2)) <- Redl (e1,e1p).
Redl (A(e1,e2), A(e1,e2p)) <- Redl (e2,e2p).

Redl (A(P(Papplic(a,LE(x))),n),P(Papplic(a,LE(res))))
```

```
                <- Redl (A(x,n),res).
Redl (A(P(Pimplic(a,LE(x))),n),P(Pimplic(a,LE(res))))
                <- Redl (A(x,n),res).
Redl (A(P(Piota(v,LE(x))),n),P(Piota(v,LE(res))))
                <- Redl (A(x,n),res).
Redl (A(P(Peta(v,LE(x))),n),P(Peta(v,LE(res))))
                <- Redl (A(x,n),res).
Redl (A(P(Pnot(LE(x))),n),P(Pnot(LE(res))))
                <- Redl (A(x,n),res).

Redl (A(P(Papplic(a,p)),n), P(Papplic(a,LE(res))))
                <- Redl (A(P(p),n),res).
Redl (A(P(Piota(v,p)),n), P(Piota(v,LE(res))))
                <- Redl (A(P(p),n),res).
Redl (A(P(Peta(v,p)),n), P(Peta(v,LE(res))))
                <- Redl (A(P(p),n),res).
Redl (A(P(Pnot(p)),n), P(Pnot(LE(res))))
                <- Redl (A(P(p),n),res).
Redl (A(P(Pimplic(a,p)),n), P(Pimplic(a,LE(res))))
                <- Redl (A(P(p),n),res).

% -------------------------C A N R E D-------------------------------
Canred (exp,canexp) <- Redl (exp,x) &
                       Canred (x,canexp).
Canred (exp,exp) <- ~Redl (exp,_).
```

A.7 Axioms

Export

```
EXPORT Axioms.

IMPORT Dico,Strings,Lists.

BASE   Formula.



CONSTANT      FBottom, FTop,FE: Formula.


FUNCTION

      FR          : Relation              -> Formula;
      Fequal      : Term     * Term       -> Formula;
      Fand        : Formula  * Formula    -> Formula;
      Fnot        : Formula               -> Formula;
      Fexist      : Variable * Formula    -> Formula;
      Fprecexist  : Program  * Formula    -> Formula;
      Fpar        : Program  * Formula    -> Formula;
      For         : Formula  * Formula    -> Formula;
      Fimplic     : Formula  * Formula    -> Formula;
      Fequiv      : Formula  * Formula    -> Formula;
      Fprecuniv   : Program  * Formula    -> Formula;
      Fall        : Variable * Formula    -> Formula;
      Fonlyone    : Variable * Formula    -> Formula.


PREDICATE
      PureProgram : Program;
      Axiom       : Formula  * Formula;
      TermVl   : Term.
```

Local

```
% Module developed by Olivier Bouchez and Olivier Istace in December 1992
```

```
LOCAL Axioms.

%- Definition part ----------------------------------------------------

% - PureProgram ------------------------------------------------------
% This predicate tests whether the representation is a real DPL program.

PureProgram (PBottom).
PureProgram (PTop).
PureProgram (PR(Rel(_,[]))).
PureProgram (PR(Rel(n,[h|t]))) <- ~TermVl(h) & PureProgram (PR(Rel(n,t))).
PureProgram (Pequal(t1,t2)) <- ~TermVl (t1) & ~TermVl (t2).
PureProgram (Pnot(p)) <- PureProgram(p).
PureProgram (Papplic(p1,p2)) <- PureProgram (p1) & PureProgram (p2).
PureProgram (Pimplic(p1,p2)) <- PureProgram (p1) & PureProgram (p2).
PureProgram (Piota(V(_),p)) <- PureProgram(p).
PureProgram (Peta(V(_),p)) <-  PureProgram(p).

% - Axiom ------------------------------------------------------------

Axiom (Fprecexist(PTop,FTop),FTop).

% The existential precondition of success of a program PTop is Top.

Axiom (Fprecexist (PR (a),phi), Fand(Fand ( FR (a),phi),c))
                                <-Lp (PR(a),b) &
                                  Axiom(Fprecexist (b,FTop),c).

% The existential precondition of a relation.
% The predicate computes the lexical presupposition of this relation and
% The predicate computes the existential precondition of success
% of this presupposition.
% The final result is the conjunction of phi, the relation and the result
% of the existential precondition of success of the presuppotision.

Axiom (Fprecexist (Pequal (t1,t2),phi),Fand(Fand (Fequal(t1,t2),phi),c))
                                <- Lp (Pequal(t1,t2),b) &
                                   Axiom(Fprecexist(b,FTop),c).

% Same system as the precedent definition but for the equality of terms.

Axiom (Fprecexist (Papplic (p1,p2),phi),x) <-
                    Axiom (Fprecexist(p2,phi),y) &
                    Axiom (Fprecexist(p1,y),x).
```

```
% The existential precondition of an application.
% The existential precondition of the right part of the application gives
% a formula and the final result is the existential precondition of the
% left part of the application with this formula.

Axiom (Fprecexist (Pnot (p),phi),Fand (phi,a)) <-
                        Axiom (Fprecuniv(p,FBottom),a).

% The existential precondition of a negation is in fact
% the universal precondition of failure of the expression composed with
% the formula phi.

Axiom (Fprecexist (Pimplic(p1,p2),phi),Fand (For(y,z),Fimplic(z2,phi)) <-
                                    Axiom (Fprecexist (p2,FTop),y) &
                                    Axiom (Fprecuniv (p2,FBottom),z) &
                                    Axiom (Fprecuniv(p1,y),z2).

% The existential precondition of an implication.
% The predicate computes the existential precondition of success
% of the right part of the implication, it gives a formula.
% The predicate computes the universal precondition of failure
% of the right part of the implication, it gives a formula.
% The predicate computes the universal precondition
% of the first obtained  formula
% of the left part of the implication, it gives a formula.
% The conjunction of the disjunction
% of the two first formulas with the implication
% from the last formula to phi is the final result.

Axiom (Fprecexist (Peta (v,p),phi),Fexist (v,y)) <-
                                        Axiom (Fprecexist (p,phi),y).

% The existential precondition of an eta assignment.
% The predicate computes the existential preconditon of the same formula
% of the program. The existential quantification of the result
% formula is the final result.

Axiom (Fprecexist (Piota (v,p),phi), Fand (Fonlyone(v,z),Fexist(v,y)))
                            <-
                                    Axiom (Fprecexist (p,phi),y) &
                                    Axiom (Fprecexist (p,FTop),z).

% The existential precondition of an iota assignment.
% The predicate computes the existential precondition of the program
% with phi (b) then we compute the existential precondition of success
```

```
% of the program (a).  The final result is the conjunction
% between the unique quantification of (a) and the existential
% quantification of (b).

%-----------------------------------------------------------------
Axiom(Fpar(x,FE),Fand(Fnot(y),Fnot(z))) <-
       Axiom(Fprecexist(x,FTop),y) &
       Axiom(Fprecuniv(x,FBottom),z).

% The precondition of error is the conjunction of the negations
% of the existential precondition of success and the universal
% precondition of failure.
%-----------------------------------------------------------------

Axiom (Fprecuniv(Papplic(p1,p2),phi),z) <-
       Axiom(Fprecuniv(p2,phi),z1) &
       Axiom(Fprecuniv(p1,z1),z).

% The universal precondition of an application is
% the universal precondition of the components in the order.

Axiom (Fprecuniv(Pnot(p),phi),For(z,Fand(y,phi))) <-
        Axiom(Fprecexist(p,FTop),z) &
        Axiom(Fprecuniv(p,FBottom),y).

% the universal precondition of a negation.
% The predicate computes the existential precondition of success
% and the universal precondition of failure .
% The final result is the disjunction of the success condition
% with the conjunction of  the failure condition and phi itself.


Axiom (Fprecuniv(Pimplic(p1,p2),phi),Fand(phi1,Fimplic(phi21,phi))) <-
        Axiom(Fprecexist(p2,FTop),phi121) &
        Axiom(Fprecuniv(p2,FBottom),phi122) &
        Axiom(Fprecuniv(p1,For(phi121,phi122)),phi1) &
        Axiom(Fprecuniv(p1,phi121),phi21).

% The universal precondition of an implication.
% The predicate computes the existential precondition of success
% of the right part (a).
% The predicate computes the universal precondition of failure
% of the right part (b).
% The predicate computes the universal precondition
% of the disjuntcion of (a) and (b)  of the left part,  it gives (c).
```

```
% the predicate computes the universal precondition of (a)
% of the left part, it gives (d).
% The conjunction of (c) and the implication from (d) to phi
% gives the final result.

Axiom (Fprecuniv(Peta(v,p),phi),Fall(v,z)) <- Axiom(Fprecuniv(p,phi),z).

% The universal precondition of an eta assignement.
% The predicate computes the universal precondition of
% the program with phi.
% The universal quantification of this formula gives the final result.

Axiom
(Fprecuniv(Piota(v,p),phi),Fand(Fonlyone(v,y),Fall(v,Fimplic(y,w)))) <-
        Axiom (Fprecexist(p,FTop),y) &
        Axiom (Fprecuniv (p,phi),w).

% The universal precondition of an iota assignement.
% The predicates computes the existential precondition
% of success of the program(a).
% The predicate computes the universal precondition of phi
% of this program.(b)
% The final result is the conjunction of the unique quantification of (a)
% and the universal quantification of the implication of (a) to (b).

Axiom (Fprecuniv(PR(a),phi),Fand(Fimplic(FR(a),phi),c)) <-
         Lp (PR(a),b) &
         Axiom(Fprecexist (b,FTop),c).

% The universal precondition of a relation.
% The predicate computes the lexical presuppositon of the relation (a).
% The predicate computes the existential precondition of success of (a).
% The final result is the conjunction of the implication from
% the relation to phi and with (c).

Axiom (Fprecuniv(Pequal(t1,t2),phi),
              Fand(Fimplic(Fequal(t1,t2),phi),c)) <-
         Lp (Pequal(t1,t2),b) &
         Axiom(Fprecexist(b,FTop),c).

% The universal precondition of an equality of terms.
% Same system as the precedent definition.

% - TermVl ------------------------------------------------------------
% This predicate tests if a representation is a variable.
```

```
TermVl (Var(Vl(_))).
```

A.8 Simple

Export

```
EXPORT Simple.

IMPORT Axioms.

PREDICATE Simplification: Formula * Formula.
```

Local

```
%
% Module developed by Olivier Bouchez and Olivier Istace
% in December 1992
% This module is in charge of simplifying some expressions.
% It simplifies the
% most frequent repetitions obtained by the Module Axioms.

LOCAL Simple.

IMPORT Axioms.

%-----------------------------------------------------------------------

PREDICATE Simple: Formula * Formula.

%-----------------------------------------------------------------------

Simple (Fand(x,FTop),x).
Simple (Fand(FTop,x),x).
Simple (Fand(_,FBottom),FBottom).
Simple (Fand(FBottom,_),FBottom).
Simple (For(_,FTop),FTop).
Simple (For(FTop,_),FTop).
Simple (For(x,FBottom),x).
Simple (For(FBottom,x),x).
```

```
Simple (Fimplic(_,FTop),FTop).
Simple (Fimplic(FTop,x),x).
Simple (Fimplic(x,FBottom),Fnot(x)).
Simple (Fimplic(FBottom,_),FTop).
Simple (Fnot(FBottom),FTop).
Simple (Fnot(FTop),FBottom).
Simple (Fand(Fonlyone(v,x),Fexist(v,x)),Fonlyone(v,x)).

% Logical simplification.

Simple (x,x).

% There is no more simplification.

Simplification (Fand(x,y),z) <-
                              Simplification (x,x2) &
                              Simplification (y,y2) &
                              Simple (Fand(x2,y2),z).

% The simplification of a conjuction is the conjunction
% of the simplifications of its components.

Simplification (For(x,y),z) <-
                              Simplification (x,x2) &
                              Simplification (y,y2) &
                              Simple (For(x2,y2),z).

% The simplification of a disjunction is the disjunction
% of the simplifications of its components.

Simplification (Fimplic(x,y),z) <-
                              Simplification (x,x2) &
                              Simplification (y,y2) &
                              Simple (Fimplic(x2,y2),z).

% The simplification of an implication is the implication
% of the simplifications of the components.

Simplification (Fnot(x),Fnot(z)) <- Simplification (x,z).

% The simplification of a negation is the negation
% of the simplification of the expression.

Simplification (Fexist(v,x),Fexist(v,z)) <- Simplification (x,z).
Simplification (Fall(v,x),Fall(v,z)) <- Simplification (x,z).
```

```
Simplification (Fonlyone(v,x),Fonlyone(v,z)) <- Simplification (x,z).

% The simplification of a quantification is the quantification
% of the simplication of the expression.

Simplification (x,x).

% There is no simplification for any other representation.
```

A.9 Eval

Export

```
EXPORT Eval.
IMPORT Strings.
PREDICATE
 Test : String * Formula * String.
```

Local

```
%
% Module developed by Olivier Bouchez and Olivier Istace in December 1992.
%
% This moduled is in charge of evaluating a formula
% with respect to a model.
% The model is represented by a godel program (First argument of
% predicate Test).  The formula is the second argument
% of the predicate test
% and the result of the evaluation is the third argument.
% From the formula given in the second argument, this module builds
% a complex goal and evaluates this goal
% into the object program (the model).
%

%-Declaration part ------------------------------------------------------


LOCAL Eval.

IMPORT Axioms,Syntax,Programs,Lists,ProgramsIO.
```

```
%-----------------------------------------------------------------------

PREDICATE Translate:

  OProgram     %input program
* String       %Name of the module
* Integer      %Counter of created predicates input
* Formula      %Formula to be translated
* OFormula     %Formula translated
* OProgram     %output program
* Integer      %Counter of created predicates output
* List(Integer) %List of free variables
;

% This predicate is in charge of translating a formula into a correct
% goal of the module of the program (OProgram).  As it was explained
% before we need to introduce some intermediates statements. Hence one
% of the paramater is the number of the last statement introduced.
% That is also why we have a resulting program with the introduced
% statements and clauses.  The list of integers is the list of numbers
% of the DPL variables used and not yet quantified.

          TradTerm: List(Term) * String * List(Integer);

% This predicate translates a list of integers into a list of terms and a
% string composed by the list of variables v indiced by an integer,
% separeted by a ",".

          Inttostring      : Integer      * String;

% Transform a signed integer into a string.

          Clean: List(Integer) * List(Integer);

% Delete repeating occurences of an integer in a ordered list.

          Typo: List(Integer) * String.

% This predicate constructs a string for the definition of a new
% object predicate.  The type choosed is Symbol.


% -Definition Part-----------------------------------------------
```

```
% ----------------------Translate --------------------------------

Translate (program,module,i,FTop,x,program,i,[]) <-
            StringToProgramFormula(program,module,"Top",[x]).

Translate (program,module,i,FBottom,z,program,i,v) <-
                    Translate (program,module,i,FTop,x,program,i,v) &
                    Not(x,z).

Translate (program,module,i,FR(Rel(s,v)),y,program,i,sv) <-
                            TradTerm (v,xv,sv) &
                            ListIntAsString ([his+32|tis],s) &
                            ListIntAsString ([his|tis],sp) &
 StringToProgramFormula(program,module,sp ++ "(" ++ xv ++")",[y]).

% translation of the functional representation of a relation
% into a string which will be translated into a functional
% representation of the object program (y).

Translate
     (program,module,i,Fequal(Var(V(j)),Var(V(k))),z,program,i,[j,k]) <-
                          Inttostring (j,cj) &
                          Inttostring (k,ck) &
                          StringToProgramFormula
                           (program,module,"v"++cj++"="++"v"++ck,[z]).

% Same system as the precedent definition
% but for the equality of two variables.

Translate
      (program,module,i,Fequal(Var(V(j)),Cst(x)),z,program,i,[j]) <-
                         Inttostring (j,cj) &
                         StringToProgramFormula
                          (program,module,"v"++cj++"="++x,[z]).

% Same system as the precedent definition
% but for the equality of a variable and a constant.


Translate (program,module,i,Fequal(Cst(x),Var(V(j))),z,program,i,[j]) <-
                            Inttostring (j,cj) &
                            StringToProgramFormula
                             (program,module,"v"++cj++"="++x,[z]).
```

```
% Same system as the previous definition
% but for the equality of a constant and a variable.

Translate (program,module,i,Fand(x,y),z,programp,j,v) <-
          Translate (program,module,i,x,x2,programx,ix,vx) &
          Translate (programx,module,ix,y,y2,programp,j,vy) &
          Merge (vx,vy,v) &
          And (x2,y2,z).

% The translation of composition consists in translating
% the two components and in associating these translations
% by the Syntax predicate And.

Translate (program,module,i,For(x,y),z,programp,j,v) <-
          Translate (program,module,i,x,x2,programx,ix,vx) &
          Translate (programx,module,ix,y,y2,programp,j,vy) &
          Merge (vx,vy,v) &
          Or (x2,y2,z).

% Same system as the previous definition
% but for the disjunction of two expressions.
% The Syntax predicate is Or.


Translate (program,module,i,Fimplic(x,y),z,programp,j,v) <-
          Translate (program,module,i,x,x2,programx,ix,vx) &
          Translate (programx,module,ix,y,y2,programp,j,vy) &
          Merge (vx,vy,v) &
          Implies (x2,y2,z).

% Same system as the precedent definition
% but for the implication of two expressions.
% The Syntax predicate is Implies.


Translate (program,module,j,Fall(V(i),x),zh,programf,k+1,sv2) <-
           Inttostring (i,si) &
           VariableName (v,"v"++si,0) &
           Translate(program,module,j,x,x2,programp,k,sv) &
           Clean(sv,svp) &
           Inttostring (k+1,kc) &
           DeleteFirst (i,svp,sv2) &
           Length (sv2,l) &
           All ([v],x2,z) &
           (IF l=0
```

```
            THEN
                ProgramPropositionName
                   (programp,module,"C"++kc,name) &
                InsDelProgramProposition
                    (programp,module,Module,name,programg) &
                StringToProgramFormula
                    (programg,module,"C"++kc,[zh])
            ELSE
                ProgramPredicateName
                    (programp,module,"C" ++ kc,1,name) &
                Typo (sv2,ss) &
                StringToProgramType(programp,module,ss,tv) &
                InsDelProgramPredicate
                 (programp,module,Module,name,NoPredInd,tv,programg) &
                TradTerm(_,lp,sv2) &
                StringToProgramFormula
                (programg,module,"C"++kc++ " (" ++ lp ++ ")",[zh])) &
        IsImpliedBy(zh,z,zs) &
        InsDelStatement(programg,module,zs,programf).

% The translation of a universally quantified expression by a variable.
% The predicate translates the expression.
% - It deletes the quantified variable from the list
%   of the free variables.
% - It builds a quantified expression with the variable and the result
%   of the translation of the expression
%   by using the Syntax predicate All.
% - If the remainig list is empty
%   the predicate has to construct an object proposition
% - Otherwise,  an object predicate with the free variables as arguments
%   is constructed.
% The technical aspect of the proposition or  predicate is C++ a counter.
% First the predicate constructs the definition of the new predicate,
% then it introduces this counter into the program
% and it obtains a new program.
% It builds the definition of the statement for this predicate
% and it introduces it into the lastly obtained program
% to get the resulting program.
% To check universal quantification it just needs
% to call this new predicate.

Translate (program,module,j,Fexist(V(i),x),zh,programf,k+1,sv2) <-
          Inttostring (i,si) &
          VariableName (v,"v"++si,0) &
          Translate(program,module,j,x,x2,programp,k,sv) &
```

```
            Clean(sv,svp) &
            Inttostring (k+1,kc) &
            DeleteFirst (i,svp,sv2) &
            Length (sv2,l) &
            Some ([v],x2,z) &
            (IF l=0
                 THEN
                     ProgramPropositionName
                         (programp,module,"C"++kc,name) &
                     InsDelProgramProposition
                          (programp,module,Module,name,programg) &
                     StringToProgramFormula
                           (programg,module,"C"++kc,[zh])
                 ELSE
                      ProgramPredicateName
                          (programp,module,"C" ++ kc,l,name) &
                      Typo (sv2,ss) &
                      StringToProgramType(programp,module,ss,tv) &
                      InsDelProgramPredicate
                    (programp,module,Module,name,NoPredInd,tv,programg) &
                      TradTerm(_,lp,sv2) &
                      StringToProgramFormula
                      (programg,module,"C"++kc++ " (" ++ lp ++ ")",[zh])) &
            IsImpliedBy(zh,z,zs) &
            InsDelStatement(programg,module,zs,programf).

% Same system as the precedent defintion
% but for the existential quantification.

Translate (program,module,j,Fonlyone(V(i),x),zh,programf,k+1,sv2) <-
            Inttostring (i,si) &
            VariableName (v,"v"++si,0) &
            Translate(program,module,j,x,x2,programp,k,sv) &
            Clean(sv,svp) &
            Inttostring (k+1,kc) &
            DeleteFirst (i,svp,sv2) &
            Length (sv2,l) &
            IntensionalSet (v,x2,sx2)&
            ProgramPredicateName
              (programp,"Sets","Size",2,nameS) &
            VariableName (vl,"z",0) &
            PredicateAtom (for,nameS, [sx2,vl]) &
            StringToProgramFormula
               (programp,module,"z=1",[z1]) &
            And (for,z1,form) &
```

```
        (IF l=0
             THEN
                 ProgramPropositionName
                     (programp,module,"C"++kc,name) &
                 InsDelProgramProposition
                      (programp,module,Module,name,programg) &
                 StringToProgramFormula
                       (programg,module,"C"++kc,[zh])
             ELSE
                  ProgramPredicateName
                      (programp,module,"C" ++ kc,l,name) &
                  Typo (sv2,ss) &
                  StringToProgramType(programp,module,ss,tv) &
                  InsDelProgramPredicate
                 (programp,module,Module,name,NoPredInd,tv,programg) &
                  TradTerm(_,lp,sv2) &
                  StringToProgramFormula
                  (programg,module,"C"++kc++ " (" ++ lp ++ ")",[zh])) &
        IsImpliedBy(zh,form,zs) &
        InsDelStatement(programg,module,zs,programf).

% For the unique existential quantification the system is a little bit
% more complicated because there is no syntax builder for this type
% of quantification.  The result predicate is an intentional set
% built with the variable and with the translation of the expression
% and a test on the size of this set wich must be equal to 1.

Translate (program,module,i,Fnot(x),z,programp,j,v) <-
                     Translate(program,module,i,x,x2,programp,j,v) &
                      Not (x2,z).

% The translation of a negation of an expression consists
% in translating the negated expression and in using
% the Syntax predicate Not to the result.

%-----------------------------------------------------------------

TradTerm
 ([Var(V(i))|t],"v" ++ n ++ "," ++ tx,[i|ti]) <-
                                        TradTerm (t,tx,ti) &
                                        Inttostring (i,n).

TradTerm ([Var(V(i))],"v" ++ n,[i]) <- Inttostring (i,n).

%-----------------------------------------------------------------
```

```
Clean ([a|c],e) <-    Member (a,c) &
                      DeleteFirst(a,c,d) &
                      Clean(Cons(a,d),e).
Clean ([a|c],[a|d]) <- ~Member(a,c) &
                       Clean (c,d).
Clean ([],[]).

%-----------------------------------------------------------------

Typo ([],"").
Typo ([_],"Symbol").
Typo ([_|t],"Symbol *" ++ tt) <- t ~=[] &
                                 Typo (t,tt).

%-----------------------------------------------------------------



Test (prog_string,x,resultat) <-
              FindInput(prog_string ++ ".prm", In(stream)) &
              GetProgram(stream, program) &
              EndInput(stream) &
              MainModuleInProgram(program, module) &
              Translate (program,module,0,x,y,program2,_,_) &
              (IF SOME [z] (Succeed (program2,y,z)) THEN
                (resultat = "YES")
               ELSE
                 (resultat = "NO")).

%-----------------------------------------------------------------

Inttostring (0,"0").
Inttostring (1,"1").
Inttostring (2,"2").
Inttostring (3,"3").
Inttostring (4,"4").
Inttostring (5,"5").
Inttostring (6,"6").
Inttostring (7,"7").
Inttostring (8,"8").
Inttostring (9,"9").
Inttostring (-1,"-1").
Inttostring (-2,"-2").
Inttostring (-3,"-3").
```

```
Inttostring (-4,"-4").
Inttostring (-5,"-5").
Inttostring (-6,"-6").
Inttostring (-7,"-7").
Inttostring (-8,"-8").
Inttostring (-9,"-9").
Inttostring (x,z1 ++ z2) <- x >= 10 & Inttostring (Abs(x Mod 10),z2 ) &
                    Inttostring (x Div 10 ,z1) .

%-----------------------------------------------------------------------
```

A.10 IOPrgs

Export

```
EXPORT IOPrgs.

IMPORT IO,Dico,Strings,Lists,Axioms.

PREDICATE

          WriteStringslash: OutputStream * String;
          OutputListWord  : OutputStream * List(Word);
          OutputFormula   : OutputStream * Formula;
          OutputProgram   : OutputStream * Program;
          Depth           : Tree * Integer ;
          Breadth         : Tree * Integer ;
          Writenumbers    : OutputStream * Integer;
          OutputTree      : OutputStream * Tree
                            * Integer * Integer          ;
          OutputTrees     : OutputStream * List(Tree).
```

Local

```
LOCAL IOPrgs.

IMPORT Axioms,Categor.

PREDICATE
          OutputVariable  : OutputStream * Variable;
          OutputListTerm  : OutputStream * List(Term);
          OutputTerm      : OutputStream * Term;
          OutputRelation  : OutputStream * Relation;
          OutputWord      : OutputStream * Word;
          OutputNumber    : OutputStream * Number;
          OutputPerson    : OutputStream * Person;
          OutputCase      : OutputStream * Case;
          OutputIndex     : OutputStream * Index;
          Inttostring     : Integer      * String;
          SaveProgram     : String       * Program;
```

```
        SaveFormula     : String       * Formula;
        OutputTree2     : OutputStream * Tree * Integer;
        OutputSommet2   : OutputStream * Info * Integer.


PREDICATE
         OutputSommet   : OutputStream * Info    * Integer *
                          Integer         ;
         OutputVector   : OutputStream *Tree * Integer * Integer *
                          Integer * Integer;
         OutputCategory : OutputStream * Cat  ;
         DrawVector     : OutputStream * Integer * Integer
                          * Integer * Integer.

Writenumbers (stream,i) <- Inttostring (i,s) & WriteString (stream,s).

WriteStringslash (stream,s) <- Put (stream,92) & WriteString (stream,s).

SaveFormula (n,f) <- FindOutput (n,Out(stream)) &
                     OutputFormula (stream,f) &
                     EndOutput (stream).

SaveProgram (n,p) <- FindOutput (n,Out(stream)) &
                     OutputProgram (stream,p) &
                     EndOutput (stream).


Inttostring (0,"0").
Inttostring (1,"1").
Inttostring (2,"2").
Inttostring (3,"3").
Inttostring (4,"4").
Inttostring (5,"5").
Inttostring (6,"6").
Inttostring (7,"7").
Inttostring (8,"8").
Inttostring (9,"9").
Inttostring (-1,"-1").
Inttostring (-2,"-2").
Inttostring (-3,"-3").
Inttostring (-4,"-4").
Inttostring (-5,"-5").
Inttostring (-6,"-6").
Inttostring (-7,"-7").
Inttostring (-8,"-8").
```

```
Inttostring (-9,"-9").
Inttostring (x,y) <- x >= 10 &
                    Inttostring (Abs(x Mod 10),z2 ) &
                    Inttostring (x Div 10 ,z1) &
                    y = z1 ++ z2.

OutputFormula(stream,FBottom) <-  WriteString (stream,"$") &
                                  WriteStringslash (stream,"bot ") &
                                  WriteString (stream,"$").

OutputFormula(stream,FTop) <-  WriteString (stream,"$") &
                               WriteStringslash (stream,"top ") &
                               WriteString (stream,"$").
OutputFormula(stream,FE) <-   WriteString (stream,"$") &
                              WriteStringslash (stream,"epsilon ") &
                              WriteString (stream,"$") .


OutputFormula(stream,FR(r)) <- OutputRelation(stream,r).
OutputFormula(stream,Fequal(a,b)) <- OutputTerm(stream,a) &
                                     WriteString(stream,"=") &
                                     OutputTerm(stream,b).
OutputFormula(stream,Fand(a,b)) <- OutputFormula(stream,a) &
                                   WriteString (stream,"$") &
                                   WriteStringslash(stream,"wedge ") &
                                   WriteString (stream,"$") &
                                   OutputFormula(stream,b).

OutputFormula(stream,Fnot(a)) <-
                                 WriteString (stream,"$") &
                                 WriteStringslash(stream,"neg ") &
                                 WriteString (stream,"$") &
                                 WriteString(stream,"(") &
                                 OutputFormula(stream,a) &
                                 WriteString(stream,")") .

OutputFormula(stream,Fexist(v,a)) <- WriteString (stream,"$") &
                                     WriteStringslash(stream,"exists ") &
                                     WriteString (stream,"$") &
                                     OutputVariable(stream,v) &
                                     WriteString(stream,"(") &
                                     OutputFormula(stream,a) &
                                     WriteString(stream,")") .

OutputFormula(stream,Fprecexist(p,a)) <- WriteString(stream,"<") &
```

```
                                        OutputProgram(stream,p) &
                                        WriteString(stream,">") &
                                        WriteString(stream," (") &
                                        OutputFormula(stream,a) &
                                        WriteString(stream,")").

OutputFormula(stream,Fimplic(a,b)) <- WriteString(stream,"(") &
                                      OutputFormula(stream,a) &
                                      WriteString(stream,")")  &
                                      WriteString (stream,"$") &
                                      WriteStringslash(stream,
                                                  "rightarrow ") &
                                      WriteString (stream,"$") &
                                      WriteString(stream,"(") &
                                      OutputFormula(stream,b) &
                                      WriteString(stream,")") .

OutputFormula(stream,Fequiv(a,b)) <- WriteString(stream,"(") &
                                     OutputFormula(stream,a) &
                                     WriteString(stream,")") &
                                     WriteString (stream,"$") &
                                     WriteStringslash(stream,
                                               "leftrightarrow ") &
                                     WriteString (stream,"$") &
                                     WriteString(stream,"(") &
                                     OutputFormula(stream,b) &
                                     WriteString(stream,"(").

OutputFormula(stream,For(a,b)) <- OutputFormula(stream,a) &
                                  WriteString (stream,"$") &
                                  WriteStringslash(stream,"vee ") &
                                  WriteString (stream,"$") &
                                  OutputFormula(stream,b).

OutputFormula(stream,Fprecuniv(p,a)) <- WriteString(stream,"[") &
                                        OutputProgram(stream,p) &
                                        WriteString(stream,"]") &
                                        OutputFormula(stream,a).
OutputFormula(stream,Fpar(p,a)) <- WriteString(stream,"(") &
                                        OutputProgram(stream,p) &
                                        WriteString(stream,")") &
                                        OutputFormula(stream,a).

OutputFormula(stream,Fall(v,a)) <-   WriteString (stream,"$") &
                                     WriteStringslash(stream,"forall ") &
```

```
                                    WriteString (stream,"$") &
                                    OutputVariable(stream,v) &
                                    WriteString(stream," (") &
                                    OutputFormula(stream,a) &
                                    WriteString(stream,")").

OutputFormula(stream,Fonlyone(v,a)) <- WriteString (stream,"$") &
                                       WriteStringslash(stream,
                                                        "exists! ") &
                                       WriteString (stream,"$") &
                                       OutputVariable(stream,v) &
                                       WriteString(stream," (") &
                                       OutputFormula(stream,a) &
                                       WriteString(stream,")").

OutputProgram(stream,PBottom) <- WriteString (stream,"$") &
                                 WriteStringslash (stream,"bot ") &
                                 WriteString (stream,"$") .

OutputProgram(stream,PTop) <- WriteString (stream,"$") &
                              WriteStringslash (stream,"top ") &
                              WriteString (stream,"$") .

OutputProgram(stream,PR(r)) <- OutputRelation(stream,r).

OutputProgram(stream,Pequal(a,b)) <- OutputTerm(stream,a) &
                                     WriteString(stream," = ") &
                                     OutputTerm(stream,b).

OutputProgram(stream,Pnot(a)) <- WriteString (stream,"$") &
                                 WriteStringslash(stream,"neg (") &
                                 WriteString (stream,"$") &
                                 OutputProgram(stream,a) &
                                 WriteString (stream,")") .

OutputProgram(stream,Papplic(a,b)) <-
                                      OutputProgram(stream,a) &
                                      WriteString(stream,"; ") &
                                      OutputProgram(stream,b) .

OutputProgram(stream,Pimplic(a,b)) <-WriteString(stream,"(") &
                                      OutputProgram(stream,a) &
                                      WriteString (stream,"$") &
                                      WriteString(stream,")") &
                                      WriteStringslash(stream,
```

```
                                                    "Rightarrow ") &
                                        WriteString(stream,"(") &
                                        WriteString (stream,"$") &
                                        OutputProgram(stream,b) &
                                        WriteString(stream,")").

OutputProgram(stream,Piota(v,p)) <- WriteString (stream,"$") &
                                    WriteStringslash(stream,"iota ") &
                                    WriteString (stream,"$") &
                                    OutputVariable(stream,v) &
                                    WriteString(stream,": ") &
                                    WriteString(stream,"(") &
                                    OutputProgram(stream,p) &
                                    WriteString(stream,")") .

OutputProgram(stream,Peta(v,p)) <-  WriteString (stream,"$") &
                                    WriteStringslash(stream,"eta ") &
                                    WriteString (stream,"$") &
                                    OutputVariable(stream,v) &
                                    WriteString(stream,": ") &
                                    OutputProgram(stream,p).

OutputVariable(stream,V(i)) <-
                                WriteString(stream," $v_{") &
                                Writenumbers(stream,i) &
                                WriteString(stream,"}$ ").

OutputTerm(stream,Var(v)) <- OutputVariable(stream,v).

OutputTerm(stream,Cst(c)) <- WriteString(stream,c).

OutputListTerm(_,[]).

OutputListTerm(stream,[a|b]) <- OutputTerm(stream,a) &
                                OutputListTerm(stream,b).

OutputRelation(stream,Rel(n,lt)) <- WriteString(stream,n) &
                                    WriteString(stream,"(") &
                                    OutputListTerm(stream,lt) &
                                    WriteString(stream,")") .


OutputWord(stream,C(n,i)) <- WriteString(stream,n) &
                             OutputIndex(stream,i).
```

```
OutputWord(stream,T(n,i,j)) <- WriteString(stream,n) &
                               OutputIndex(stream,i) &
                               OutputIndex(stream,j).

OutputWord(stream,S(n)) <- WriteString(stream,n).

OutputIndex(stream,D(i)) <-  WriteString(stream,"$_") &
                             Writenumbers(stream,i) &
                             WriteString(stream,"$").

OutputIndex(stream,U(i)) <-  WriteString(stream,"$^") &
                             Writenumbers(stream,i) &
                             WriteString(stream,"$").


OutputListWord(_,[]).
OutputListWord(stream,[a|b]) <- OutputWord(stream,a) &
                                WriteString(stream," ") &
                                OutputListWord(stream,b).

OutputCategory (stream,E(n,p,c)) <- WriteString (stream,"E (")
                             & OutputNumber (stream,n)
                             & WriteString (stream, ",")
                             & OutputPerson (stream,p)
                             & WriteString (stream,",")
                             & OutputCase (stream,c)
                             & WriteString (stream,")").

OutputCategory (stream,S) <- WriteString (stream,"S").
OutputCategory (stream,DET(n)) <- WriteString (stream,"DET (")
                             & OutputNumber (stream,n)
                             & WriteString (stream,")").

OutputCategory (stream,NP(n,p,c)) <- WriteString (stream,"NP (")
                             & OutputNumber (stream,n)
                             & WriteString (stream, ",")
                             & OutputPerson (stream,p)
                             & WriteString (stream,",")
                             & OutputCase (stream,c)
                             & WriteString (stream,")").


OutputCategory (stream,TV(n,p)) <- WriteString (stream,"TV (")
                             & OutputNumber (stream,n)
                             & WriteString (stream, ",")
```

```
                                & OutputPerson (stream,p)
                                & WriteString (stream,")").

OutputCategory (stream,N(n)) <- WriteString (stream,"N (")
                                & OutputNumber (stream,n)
                                & WriteString (stream,")").

OutputCategory (stream,VP(n,p)) <- WriteString (stream,"VP (")
                                & OutputNumber (stream,n)
                                & WriteString (stream, ",")
                                & OutputPerson (stream,p)
                                & WriteString (stream,")").

OutputCategory (stream,AUX(n,p)) <- WriteString (stream,"AUX (")
                                & OutputNumber (stream,n)
                                & WriteString (stream, ",")
                                & OutputPerson (stream,p)
                                & WriteString (stream,")").


OutputCategory (stream,NEG) <- WriteString (stream,"NEG").

OutputCategory (stream,a/b) <- OutputCategory (stream,a) &
                                WriteString (stream,"/") &
                                OutputCategory (stream,b).

OutputCategory (stream,a\b) <- OutputCategory (stream,a) &
                                WriteString (stream,"$") &
                                WriteStringslash (stream,"backslash$") &
                                OutputCategory (stream,b).


OutputNumber (stream,Sg) <- WriteString (stream,"Sg").
OutputNumber (stream,Pl) <- WriteString (stream,"Pl").
OutputPerson (stream,Third) <- WriteString (stream,"Third").
OutputPerson (stream,Other) <- WriteString (stream,"Other").
OutputCase (stream,Nom) <- WriteString (stream,"Nom").
OutputCase (stream,Acc) <- WriteString (stream,"Acc").
Breadth(Empty,0).
Breadth(A(_,g,d),b) <- Breadth(g,b1) &
                       Breadth(d,b2) &
                       b = b1 + b2 + 1.

Depth(Empty,1).
Depth(A(_,g,d),p) <- Depth(g,p1) &
```

```
                    Depth(d,p2) &
                    p = Max(p1,p2)+1.
OutputTree (_,Empty,_,_).

OutputTree (stream,A(i,g,d),m,dl) <- Breadth(A(i,g,d),l) &
                                     x = (l+1) Div 2 + m &
                                     OutputSommet (stream,i,x,dl) &
                                     y = l Div 2   &
                                     d2 = dl-1 &
                                     y2 = (y+1) Div 2 + m &
                                     OutputVector
(stream,g,x,dl,y2,d2) &
                                     OutputTree (stream,g,m,d2) &
 %                                    m2 = m+y &
                                     y3 = x-m+y2 &
                                     OutputVector
(stream,d,x,dl,y3,d2) &
                                     OutputTree(stream,d,x,d2).


OutputSommet (stream,I2(n,m),x,d) <-
                        WriteStringslash(stream,"put(") &
                        x20 = x*20 &
                        d30 = d*30 &
                        Inttostring (x20,x20p) &
                        WriteString (stream,x20p) &
                        WriteString(stream,",") &
                        Inttostring (d30,d30p) &
                        WriteString (stream,d30p) &
                        WriteString(stream,"){") &
                        WriteStringslash (stream,"makebox(0,0){") &
                        OutputCategory(stream,n) &
                        WriteString(stream,"}}") &
                        NewLine(stream)  &
                        y1 = d * 30 - 7 &
                        l = y1 - 7  &
                        WriteStringslash(stream,"put(") &
                        Writenumbers (stream,x20) &
                        WriteString(stream,",") &
                        Writenumbers (stream,y1) &
                        WriteString(stream,"){") &
                        WriteStringslash(stream,"line (0,-1){") &
                        Writenumbers (stream,l) &
                        WriteString(stream,"}}") &
                        NewLine(stream) &
```

```
                        WriteStringslash(stream,"put(") &
                        Inttostring (x20,x20p) &
                        WriteString (stream,x20p) &
                        WriteString (stream,",0){") &
                        WriteStringslash (stream,"makebox(0,0){") &
                        OutputListWord (stream,m) &
                        WriteString(stream,"}}") &
                        NewLine(stream) .

OutputSommet (stream,I1(n),x,d) <-
                        WriteStringslash(stream,"put(") &
                        x20 = x*20 &
                        Inttostring (x20,x20p) &
                        WriteString (stream,x20p) &
                        WriteString(stream,",") &
                        d30 = d*30 &
                        Inttostring (d30,d30p) &
                        WriteString (stream,d30p) &
                        WriteString (stream,"){") &
                        WriteStringslash(stream,"makebox(0,0){") &
                        OutputCategory(stream,n) &
                        WriteString(stream,"}}") &
                        NewLine(stream) .

OutputVector (_,Empty,_,_,_,_).
OutputVector (stream,_,x1,y1,x2,y2) <- yr1 = y1 * 30 - 10 &
                                       yr2 = y2 * 30 + 10 &
                                       xr1 = x1*20 &
                                       xr2 = x2*20 &

DrawVector(stream,xr1,yr1,xr2,yr2).

% Tracer un vecteur de coordonnees (xr1,yr1,xr2,yr2).

DrawVector (stream,x1,y1,x2,y2) <-
             lx = x2-x1 &
             ly = y2-y1 &
             l = Abs(lx) &
             n = Min (l,Abs(ly)) &
             ex = Min(((Abs(lx) * 1000) Div n + 500 ) Div 1000,4)
             * Abs(lx) Div lx &
             ey = Min(((Abs(ly) * 1000) Div n + 500 ) Div 1000,4)
             * Abs(ly) Div ly &
             WriteStringslash(stream,"put(") &
             Writenumbers (stream,x1) &
```

```
            WriteString(stream,",") &
            Writenumbers (stream,y1) &
            WriteString(stream,"){") &
            WriteStringslash(stream,"vector(") &
            Writenumbers (stream,ex) &
            WriteString(stream,",") &
            Writenumbers (stream,ey) &
            WriteString(stream,"){") &
            Writenumbers (stream,l) &
            WriteString(stream,"}}") &
            NewLine(stream)  .

OutputTree2 (_,Empty,_).

OutputTree2 (s,A(i,l,r),m) <- OutputSommet2 (s,i,m)
                              & OutputTree2 (s,l,m+1)
                              & OutputTree2 (s,r,m+1).

OutputSommet2 (s,I1(c),m) <- WriteStringslash (s,"parindent ")
                          & Writenumbers (s,m)
                          & WriteString (s," cm")
                          & NewLine (s)
                          & OutputCategory (s,c)
                          & NewLine (s)
                          & NewLine (s).



OutputSommet2 (s,I2(c,w),m) <- WriteStringslash (s,"parindent ")
                          & Writenumbers (s,m)
                          & WriteString (s," cm")
                          & NewLine (s)
                          & OutputCategory (s,c)
                          & WriteString (s,": ")
                          & OutputListWord (s,w)
                          & NewLine (s)
                          & NewLine (s).

OutputTrees (_,[]).
OutputTrees (s,[h|t]) <- OutputTree2 (s,h,0)
                       & OutputTrees (s,t)
                       & WriteStringslash (s,"parindent 0 cm").
```

A.11 Dico

Export

```
EXPORT Dico.

IMPORT Lists,Strings.

BASE Cat,Expr,Variable,Program,Word,Term, Relation,
     Const, Number, Person, Case, Index.

% The constants are the basic categories and some synonymes

CONSTANT      S, NEG         : Cat;
              PBottom, PTop  : Program;
              Sg, Pl         : Number;
              Third, Other   : Person;
              Acc, Nom       : Case.


% U = Upper Index
% D = Down  Index

FUNCTION
              U : Integer  -> Index;
              D : Integer  -> Index.

% Function to represent sentences
% S = singleton
% C = couple
% T = triplet


FUNCTION      S : String                          -> Word;
              C : String   * Index                -> Word;
              T : String   * Index   * Index      -> Word.


% Function to build categories
% If A and B are categories then A/B and A\B are categories


FUNCTION      E   : Number * Person * Case        -> Cat;
              N   : Number                        -> Cat;
```

```
            NP  : Number * Person * Case        -> Cat;
            DET : Number                        -> Cat;
            TV  : Number * Person               -> Cat;
            VP  : Number * Person               -> Cat;
            AUX : Number * Person               -> Cat;

            / : yFx (400) : Cat * Cat -> Cat;
            \ : yFx (400) : Cat * Cat -> Cat.

% Function to represent lambda expression
% A lambda expression is
%   V     - A single variable
%   A     - An application of two lambda expressions
%   L     - An abstraction of a variable into a expression

%   Vdal  - A variable representing a DAL variable
%   T     - A variable representing the text of DAL program
%   Iota  - A variable representing a iota definition of a Vdal
%   Eta   - A variable representing a eta definition of a Vdal


FUNCTION      Vl   : Integer        -> Variable;
              VE   : Variable       -> Expr;
              A    : Expr    * Expr -> Expr;
              L    : Integer * Expr -> Expr;
              P    : Program        -> Expr;
              LE   : Expr           -> Program.

FUNCTION
      V           : Integer              -> Variable;
      Var         : Variable             -> Term;
      Cst         : String               -> Term;
      Rel         : String   * List(Term) -> Relation;
      PR          : Relation             -> Program;
      Pequal      : Term     * Term      -> Program;
      Pnot        : Program              -> Program;
      Papplic     : Program  * Program   -> Program;
      Pimplic     : Program  * Program   -> Program;
      Peta        : Variable * Program   -> Program;
      Piota       : Variable * Program   -> Program.

% The predicate Dicl memorises a Word, its category and its
% associated lambda expression.

PREDICATE
```

```
Dicl      : List (Word) * Cat * Expr;
Lp        : Program * Program;
Syncat    : Cat       * Cat;
Text      : List(Cat).
```

Local

```
LOCAL Dico.

PREDICATE

              Lname : String      * Expr;
              Ltv   : String      * Expr;
              Lvp   : String      * Expr.


% --------------------------D I C L----------------------------------

% ------------- D E T E R M I N E R --------------------------------

Dicl([C("a",U(i))],DET(Sg),
L(1,L(2,P(Papplic(Peta(V(i),LE(A(VE(Vl(1)),VE(V(i)))),LE(A(VE(Vl(2)),
VE(V(i))))))))).

Dicl([C("the",U(i))],DET(_),
L(1,L(2,P(Papplic(Piota(V(i),LE(A(VE(Vl(1)),VE(V(i)))),LE(A(VE(Vl(2)),
VE(V(i))))))))).

Dicl([T("the",U(i),D(j))],DET(_),
L(1,L(2,P(Papplic(Piota(V(i),Papplic(Pequal(Var(V(i)),Var(V(j))) ),
LE(A(VE(Vl(1)),VE(V(i))))),LE(A(VE(Vl(2)),VE(V(i))))))))).

% ------------- N O U N ----------------------------------------------

Dicl([S("man")],N(Sg),1) <- Lname ("man",1).
Dicl([S("manager")],N(Sg),1) <- Lname ("manager",1).
Dicl([S("pc")],N(Sg),1) <- Lname ("pc",1).
Dicl([S("character")],N(Sg),1) <- Lname ("character",1).
Dicl([S("hat")],N(Sg),1) <- Lname ("hat",1).
Dicl([S("capital")],N(Sg),1) <- Lname ("capital",1).
Dicl([S("bachelor")],N(Sg),1) <- Lname ("bachelor",1).
Dicl ([S("wife")],N(Sg),1) <- Lname ("wife",1).
Dicl([S("king")],N(Sg),1) <- Lname ("king",1).
```

```
% -------------- T V ------------------------------------------------
Dicl([S("sees")],TV(Sg,Third), 1) <- Ltv ("see",1).
Dicl([S("be"),S("cross"),S("with")],TV(Sg,_),1) <-
                Ltv ("be-cross-with",1).
Dicl([S("admires")],TV(Sg,Third),1) <- Ltv ("admire",1).
Dicl([S("cheers")],TV(Sg,Third),1) <- Ltv ("cheer",1).
Dicl([S("use")],TV(Sg,_),1) <- Ltv ("use",1).

Dicl([S("is")],TV(Sg,Third),
L(1,L(2,A(VE(V1(1)),L(3,P(Pequal(Var(V1(2)),Var(V1(3))))))))
).

% -------------- N P ------------------------------------------------
Dicl([C("John",U(i))],NP(Sg,Third,_),
L(1,P(Peta(V(i),Papplic(Pequal(Var(V(i)),Cst("John")),LE(A(VE(V1(1)),
VE(V(i))))))))).

Dicl([C("Mary",U(i))],NP(Sg,Third,_),
L(1,P(Peta(V(i),Papplic(Pequal(Var(V(i)),Cst("Mary")),LE(A(VE(V1(1)),
VE(V(i))))))))).

Dicl ([C("he",D(i))],NP(Sg,Third,Nom),L(1,A(VE(V1(1)),VE(V(i))))).
Dicl([C("him",D(i))],NP(Sg,Third,Acc),L(1,A(VE(V1(1)),VE(V(i))))).
% --------------V P ------------------------------------------------
Dicl([S("sits"),S("down")],VP(Sg,Third),1) <- Lvp("sit-down",1).

% --------------A U X ----------------------------------------------
Dicl([S("does")],AUX(Sg,Third),L(1,VE(V1(1)))).

% -------------- M I S C ------------------------------------------

Dicl([S("not")],NEG,
L(1, L(2, L(3, A(VE(V1(1)), P(Pnot (LE(A ( VE(V1(2)),
VE(V1(3)) )) )) ) ))) ).

Ltv (s,L(1,L(2,A(VE(V1(1)),L(3,P(PR(Rel(s,[Var(V1(2)),Var(V1(3))]
)))))))).

Dicl([S("then")],S/S,L(1,VE(V1(1)))).
Dicl ([S("if")],(S/S)/S,L(1,L(2,P(Pimplic(LE(VE(V1(1))),
LE(VE(V1(2))))))).

Dicl([S("with")],(N(_)\N(_))/NP(_,_,_),
L(1,L(2,L(3,P(Papplic(LE(A( VE(V1(2)),VE(V1(3)) ) ),
```

```
                    LE(A( VE(Vl(1)),L(4,P(PR(Rel("is-of",
[Var(Vl(4)), Var(Vl(3))] ))))))
)))))
).

% ------------- L A M B D A   M A C R O -----------------------------

Lname(s,L(1,P(PR(Rel(s,[Var(Vl(1))])))))).

Ltv (s,L(1,L(2,A(VE(Vl(1)),L(3,P(PR(Rel(s,[Var(Vl(2)),Var(Vl(3))])))))))).

Lvp (s,L(1,P(PR(Rel(s,[Var(Vl(1))])))))).

Lp
(PR(Rel("bachelor",i)),Papplic(PR(Rel("man",i)),
PR(Rel("adult",i)))).
Lp (_,PTop).


% ------------ S Y N C A T -----------------------------------------

Syncat (N(n), S/E(n,_,_)).
Syncat (DET(n), (S/(E(n,a,_)\S))/(S/E(n,a,_))).
Syncat (NP(n,p,c), S/(E(n,p,c)\S)).
Syncat (TV(n,p), (E(n,p,Nom)\S)/(S/(E(_,_,Acc)\S))).
Syncat (VP(n,p), (E(n,p,Nom)\S)).
Syncat (AUX(n,p) , (E(n,p,Nom)\S)/(E(_,_,_)\S)).
Syncat (NEG, ((E(n,p,Nom)\S)/(E(_,_,_)\S))\((E(n,p,Nom)\S)/(E(_,_,_)\S))).
Syncat (S,S).
Syncat (E(n,p,c),E(n,p,c)).
Syncat (a/b,(c/d)) <- Syncat(a,c) &
                      Syncat(b,d).
Syncat (a\b,(c\d)) <- Syncat(a,c) &
                      Syncat(b,d).


% ------------ T E X T ---------------------------------------------

DELAY Text (x) UNTIL NONVAR (x).
Text([]).
```