Chrysostome Baskiotis

Fascicule 1

LOGIQUE COMPUTATIONNELLE & PROLOG

Calcul propositionnel
Introduction à Prolog

EISTI

Année 2011

1.3LOGIQUE COMPUTATIONNELLE & PROLOG

Site du cours

http://sifoci.eisti.fr

Cours de 1e année, 2e semestre.

30h00 d'enseignement décomposées en 10h00 de cours, 15h00 de TD et 5h00 de TP.

Objectif :

La logique computationnelle malgré son aspect assez théorique constitue un formidable outil pour la pratique de l'informaticien. En effet l'industrie informatique se trouve confrontée à ce qu'on appelle *la crise du logiciel* c'est-à-dire à la difficulté de *vérifier* la conformité d'un système informatique par rapport à ses spécifications formelles et de *valider* que le comportement observé du système n'est pas différent du comportement attendu. Les méthodes de vérification sont fondées sur la logique computationnelle.

Ainsi, après un bref rappel de la logique propositionnelle, on étudiera la logique des prédicats qui est l'outil de base pour la logique computationnelle. On examinera ensuite l'univers et le modèle d'Herbrand qui permet de déterminer la réponse correcte d'un programme. On terminera avec l'étude de la logique floue.

L'aspect théorique de la logique computationnelle sera complété par l'apprentissage du langage de programmation Prolog qui permettra à l'élève de faire de la programmation logique en utilisant ses acquis de la logique.

Prérequis :

Mathématiques pour l'ingénieur. Théorie des graphes. Algèbre de Boole.

Professeurs :

— CERGY : **Chrysostome Baskiotis** (Cours+TD+TP), **Jean-Paul Forest** (TD+TP)

— PAU : **Yannick Le Nir** (Cours+TD+TP)

Supports du cours :

Partie logique

— Polycopié du cours.

— R. CORI & D. LASCAR :*Logique mathématique*, 2 vol. Masson, 1993

— J. W. LLOYD : *Fondements de la programmation logique*, Eyrolles, 1988

— K. R. APT & E.-R. OLDEROG : *Verification of sequential and concurrent programs*, Springer, 1991

Partie Prolog

— W. F. CLOCKSIN & C. S. MELLISH : *Programmer en Prolog*, Éditions Eyrolles, traduction en français du livre *Programming in Prolog* aux éditions Springer-Verlag

— J. ELBAZ : *Programmer en Prolog*, Éditions Ellipses, 1991

— I. BRATKO : *Prolog, Programming for artificial intelligence*, Addison-Wesley, 1986

Supports supplémentaires
Partie Logique

— E. W. BETH : *The foundations of mathematics*, North-Holland, 1965

— G. S. BOOLOS & R. C. JEFFREY : *Computability and logic*, 3e Ed., Cambridge Univ. Press, 1989

— A. CHURCH : *Introduction to mathematical logic*, Princeton Univ. Press, 1956

— R. FRAÏSSÉ : *Cours de logique mathématique*, 2 vol., Gauthiers-Villars, 1972

— M. R. GENESERETH & N. J. NILSSON : *Logical foundations of artificial intelligence*, Morgan Kaufmann, 1987

— G. KREISEL & J. L. KRIVINE : *Éléments de logique mathématique*, Dunod, 1966

— U. NILSSON & J. MAŁUSZYŃSKI : *Logic, Programming and Prolog*, Wiley, 1990

— H. RASIOWA & R. SIKORSKI : *The mathematics of metamathematics*, PWN, 1963

— D. VAN DALEN : *Logic and structure*, Springer, 1997

Partie Prolog

— T. Amble : *Logic programming and knowledge engineering*, Addison-Wesley, 1987

— W. F. CLOCKSIN : *Clause and Effect : Prolog programming for the working programmer*, Springer, 1997 P. DERANSART, A. ED-DBALI, L. CERVONI : *Prolog : The Standard*, Springer-Verlag, 1986 R. O'KEFFE : *The craft of Prolog*, MIT Press, 1990

— J. MALPAS : *Prolog : a relational language and its applications*, Prentice-Hall, 1987

— L. STERLING & E. SHAPIRO : *L'art de Prolog*, Éditions InterÉditions,
et dont tous les exemples de programmes sont sur Internet, en libre accés à l'adresse du MIT : `ftp://mitpress.mit.edu`.

— J. STOBO : *Problem solving with Prolog*, Pitman, 1989

Contrôle des connaissances :

— Examen.

Remarques : Sur le site du cours vous trouverez

— Le poly du cours, réactualisé chapitre par chapitre.

— Des indications pour le cours et le TD ou le TP de chaque semaine.

— Des documents complémentaires.

1.3LOGIQUE COMPUTATIONNELLE & PROLOG

Site du cours

http://sifoci.eisti.fr

Cours de 1e année, 2e semestre.

30h00 d'enseignement décomposées en 10h00 de cours, 15h00 de TD et 5h00 de TP.

Objectif :

La logique computationnelle malgré son aspect assez théorique constitue un formidable outil pour la pratique de l'informaticien. En effet l'industrie informatique se trouve confrontée à ce qu'on appelle *la crise du logiciel* c'est-à-dire à la difficulté de *vérifier* la conformité d'un système informatique par rapport à ses spécifications formelles et de *valider* que le comportement observé du système n'est pas différent du comportement attendu. Les méthodes de vérification sont fondées sur la logique computationnelle.

Ainsi, après un bref rappel de la logique propositionnelle, on étudiera la logique des prédicats qui est l'outil de base pour la logique computationnelle. On examinera ensuite l'univers et le modèle d'Herbrand qui permet de déterminer la réponse correcte d'un programme. On terminera avec l'étude de la logique floue.

L'aspect théorique de la logique computationnelle sera complété par l'apprentissage du langage de programmation Prolog qui permettra à l'élève de faire de la programmation logique en utilisant ses acquis de la logique.

Prérequis :

Mathématiques pour l'ingénieur. Théorie des graphes. Algèbre de Boole.

Professeurs :

— CERGY : **Chrysostome Baskiotis** (Cours+TD+TP), **Jean-Paul Forest** (TD+TP)

— PAU : **Yannick Le Nir** (Cours+TD+TP)

Supports du cours :

Partie logique

— Polycopié du cours.

— R. CORI & D. LASCAR :*Logique mathématique*, 2 vol. Masson, 1993

— J. W. LLOYD : *Fondements de la programmation logique*, Eyrolles, 1988

— K. R. APT & E.-R. OLDEROG : *Verification of sequential and concurrent programs*, Springer, 1991

Partie Prolog

— W. F. CLOCKSIN & C. S. MELLISH : *Programmer en Prolog*, Éditions Eyrolles, traduction en français du livre *Programming in Prolog* aux éditions Springer-Verlag

— J. ELBAZ : *Programmer en Prolog*, Éditions Ellipses, 1991

— I. BRATKO : *Prolog, Programming for artificial intelligence*, Addison-Wesley, 1986

Supports supplémentaires
Partie Logique

— E. W. BETH : *The foundations of mathematics*, North-Holland, 1965

— G. S. BOOLOS & R. C. JEFFREY : *Computability and logic*, 3e Ed., Cambridge Univ. Press, 1989

— A. CHURCH : *Introduction to mathematical logic*, Princeton Univ. Press, 1956

— R. FRAÏSSÉ : *Cours de logique mathématique*, 2 vol., Gauthiers-Villars, 1972

— M. R. GENESERETH & N. J. NILSSON : *Logical foundations of artificial intelligence*, Morgan Kaufmann, 1987

— G. KREISEL & J. L. KRIVINE : *Éléments de logique mathématique*, Dunod, 1966

— U. NILSSON & J. MAŁUSZYŃSKI : *Logic, Programming and Prolog*, Wiley, 1990

— H. RASIOWA & R. SIKORSKI : *The mathematics of metamathematics*, PWN, 1963

— D. VAN DALEN : *Logic and structure*, Springer, 1997

Partie Prolog

— T. Amble : *Logic programming and knowledge engineering*, Addison-Wesley, 1987

— W. F. CLOCKSIN : *Clause and Effect : Prolog programming for the working programmer*, Springer, 1997 P. DERANSART, A. ED-DBALI, L. CERVONI : *Prolog : The Standard*, Springer-Verlag, 1986 R. O'KEFFE : *The craft of Prolog*, MIT Press, 1990

— J. MALPAS : *Prolog : a relational language and its applications*, Prentice-Hall, 1987

— L. STERLING & E. SHAPIRO : *L'art de Prolog*, Éditions InterÉditions,
et dont tous les exemples de programmes sont sur Internet, en libre accés à l'adresse du MIT : `ftp://mitpress.mit.edu`.

— J. STOBO : *Problem solving with Prolog*, Pitman, 1989

Contrôle des connaissances :

— Examen.

Remarques : Sur le site du cours vous trouverez

— Le poly du cours, réactualisé chapitre par chapitre.

— Des indications pour le cours et le TD ou le TP de chaque semaine.

— Des documents complémentaires.

INTRODUCTION

Sous le nom d'Intelligence Artificielle (IA) il y a plusieurs branches de Mathématiques et de l'Informatique qui se regroupent. Le cycle des cours consacré à l'IA est composé de Langages, Logique Computationnelle, Prolog, Décidabilité, Intelligence Artificielle symbolique et computationnelle et Systèmes Experts. Ce cycle est complété par l'option Génie Logiciel - Systèmes Intelligents et Complexes (GL-SICO) qui est, en partie, dédiée à l'IA. Il faut aussi signaler que la plupart des autres options font appel à diverses parties de l'IA.

Il est très difficile, voire impossible, d'avoir une définition de l'IA. La situation la plus acceptable serait d'utiliser une approche circulaire du type « l'intelligence artificielle est la résolution de problèmes par modélisation de connaissances ». En revanche il est très facile d'énumérer les disciplines qui se réclament, par elles-mêmes, comme faisant partie de cette science. Situation normale dans la mesure où l'IA se trouve au croisement des plusieurs disciplines et pour lesquelles constitue un des enjeux principaux. Mais en même temps situation embarrassante pour l'honnête homme de ce début de siècle dans sa tentative d'approcher son contenu.

Il n'est guère possible dans ce cas que de commencer par le début. Nous pouvons dire que l'objet de l'IA est la connaissance de la connaissance. Définition parallèle à celle donnée par Paul Valéry pour l'homme : « une tentative pour créer ce que j'oserai nommer l'esprit de l'esprit »([1]).

[1] P. Valéry Politique de l'esprit, Œuvres, Col. Pléiade.

Bien que l'aventure soit récente, l'histoire est ancienne. Descartes([2]) est certainement le premier qui a posé le problème de la connaissance dans sa généralité et qui a inventé une Méthode pour l'aborder. Et qui, de plus, a décrit son parcours : « Je me trouvais comme contraint d'entreprendre moi-même de me conduire. Comme un homme qui marche seul et dans les ténèbres ». En fait ce qui préoccupe Descartes au beau milieu du deuxième millénaire est l'éternel défi : régler, comme un mécanisme d'horlogerie, la démarche de l'esprit pour conquérir la connaissance, cette chimère sauvageonne et indomptable.

De ces 400 ans qui nous séparent de l'époque de Descartes, les plus importants sont les derniers 60 ans, pendant lesquels une véritable explosion a eu lieu à l'intérieur de l'IA. Création des nouvelles branches : réseaux neuronaux, algorithmes génétiques, machines à support vectoriel, boosting, systèmes experts, agents intelligents, apprentissage par renforcement, traitement du langage naturel, etc. Réorientation d'anciennes branches : classification, classement, reconnaissance de formes, traitement du signal, commande, régulation, etc. Le défi reste le même, peut-être légèrement déplacé : construire de faartificielle de l'intelligence et utiliser l'intelligence ainsi construite.

Pendant ce parcours certains ont craint de perdre Descartes, d'autres l'auraient laissé volontiers au bas-côté de la chaussée. Mais lui, d'après Garcia Lorca, « petit comme une amande verte, lassé des cercles et des droites, s'enfuit par les canaux pour entendre chanter les matelots ivres ». Descartes nous faussant compagnie. Certes non ! Mais simple renouvellement du défi. Attrapons au vol ce chant « Quelqu'un m'a raconté que perdu dans les glaces, / Dans un chaos de monts, loin de tout océan, / Il vit passer sans heurt et sans fumée la masse / Immense et pavoisée d'un

[2]Plus de 400 ans depuis sa naissance le 21 mars 1596.

paquebot géant »([3]) et essayons de doter les paquebots géants mais aussi les minuscules robots-jouets avec des systèmes embarqués. Nous nous trouverons ainsi face aux limites scientifiques et techniques actuelles. Car les systèmes embarqués – et l'informatique de demain sera essentiellement une informatique embarquée – il faut qu'ils puissent planifier des actions, qu'ils puissent surmonter des situations imprévisibles, donc être capables de modifier leur programme, et, encore, qu'ils puissent, en cas de panne, de s'auto-réparer. Tâches éminemment complexes qui requièrent de la connaissance.

Ainsi le traitement de la connaissance, qui est l'objet de l'étude de l'I.A., constitue en même

[3] R. Desnos déjà 60 ans depuis sa mort en 1945 au camp de concentration de Terezine.

temps la composante principale de la science informatique([4]). Comme d'habitude dans les sciences en pareilles circonstances, nous n'avons pas une idée claire de ce que c'est l'objet de la science, à savoir la connaissance. Déjà Gaston Berger écrivait en 1941 « Il faut dire (...), en toute

[4]C'est, d'ailleurs, grâce à cette composante que l'informatique a passé du statut de la technique à celui de la science

rigueur, que la connaissance est indéfinissable »([5]). Il est difficile de savoir si cette pensée est la conséquence de la définition circulaire donnée par Henri Bergson « Si l'instinct et l'intelligence enveloppent, l'un et l'autre, des connaissances, la connaissance est plutôt jouée et inconsciente

[5]G. Berger :Recherches sur les conditions de la connaissance. Essai d'une théorétique pure, Paris, P. U. F., 1941

dans le cas de l'instinct, plutôt pensée et consciente dans le cas de l'intelligence »([6]). Bergson procède à une reduction de la connaissance à l'intelligence. Peut-être en 1907 ceci était-il licite. Mais en 1921 l'intelligence devient un paysage brumeux. Une revue de psychologie demande cette année à quatorze experts de lui fournir une définition de l'intelligence et elle en récolte neuf

[6]H. Bergson : L'Évolution créatrice, Paris, 1907

différentes([7])! Si on posait la même question aujourd'hui, peut-être aurions-nous plus de définitions que d'experts. Une réponse, assez technique dans son essence, est donnée par Gaston Bachelard qui écrivit « Pour un esprit scientifique, toute connaissance est une réponse à une question. S'il n'y a pas de question, il ne peut y avoir de connaissance scientifique. Rien ne va de soi.

[7] cité par R.L.Gregory (dir.) : Le cerveau un inconnu, Robert Laffont, Paris, 1993, pp.667-668

Rien n'est donné. Tout est construit. »([8]). C'est donc grâce au sujet agissant et à son action qu'il y a création de la connaissance. Il est permis toutefois de douter que cette belle et très humaniste définition de Bachelard puisse être appliquée à des machines qui, au plus, peuvent espérer au statut des humanoïdes. Néanmoins nous pouvons, en utilisant cette définition et des éléments de la Théorie de l'Information, évaluer le coût et, donc, la valeur de la connaissance. C'est, comme nous le savons déjà, le nombre de questions élémentaires qu'il faut poser pour obtenir la réponse que l'on cherche. Poursuivant dans cette voie nous pouvons aussi citer une récente – et extrême-

[8]G. Bachelard : La formation de l'esprit scientifique : contribution à une psychanalyse de la connaissance objective,

Vrin, Paris, 1938

ment intéressante – contribution de Hervé Zwirn. Dans un article([9]) il distingue deux types de connaissances. Une qui est une connaissance forte et se produit en ramenant un phénomène inconnu à un phénomène familier qui nous est compréhensible. De cette fanous connaissons quelque chose parce que nous le comprenons et nous pouvons ainsi dire que nous connaissons une parcelle du monde qui nous entoure. Et une autre connaissance, qui est une connaissance faible et qui ramène un phénomène inconnu à une loi générale qui l'explique. Il est évident que la loi générale nous ne la comprenons pas. Par exemple il est impossible de comprendre la loi de la gravitation. Dans ce cas nous disons que nous connaissons le phénomène parce que nous sommes capables, en utilisant la loi générale à laquelle il obéit, de prédire son évolution. En somme nous connaissons le phénomène qui va se passer avant qui se réalise. Nous effectuons ainsi une contraction du temps. Si nous traduisons cette contraction en termes informatiques, elle revient à être une compression des données. Nous approchons ainsi, par une autre voie, l'idée d'Andrei Kolmogorov concernant la relation entre information et complexité. Selon cette approche, qui est aussi celle de Ray Solomonoff, l'information apportée par un phénomène est fonction de la longueur du code du phénomène, si on transcrit ce phénomène en code binaire. La complexité d'un phénomène est considérée comme étant relative à la longueur du code du phénomène. Si donc nous sommes capables de compresser le code binaire pour le faire exécuter et avoir des résultats avant qu'ils se produisent, nous pouvons dire que nous connaissons le phénomène. Si par contre nous ne pouvons pas effectuer une compression du code, alors nous ne sommes pas capables de connaître le phénomène. C'est ce qui se passe avec la génération des nombres aléatoires et l'évolution des processus chaotiques où nous ne connaissons pas les résultats avant qu'ils se réalisent bien que la ou les lois de leur évolution sont connues. Bien sûr il ne faut pas aller vite en besogne et affirmer que plus le phénomène est moins complexe, moins nous le comprenons plus lentement.

Pendant ce cours nous verrons d'abord les raisons pour lesquelles la logique est un excellent langage pour le traitement de la connaissance. Ensuite nous travaillerons avec la logique du premier ordre avant d'examiner des algorithmes qui conduisent à l'automatisation de l'opération d'inférence logique, opération qui permet la production des connaissances nouvelles. Tout au long de ce cours afin de favoriser la compréhension et la consolidation d'un point important soit de la théorie soit de son application, nous le faisons suivre par des ascèses que les élèves doivent résoudre. Par ailleurs chaque chapitre se termine par des exercices qui permettent aux élèves de faire un tour récapitulatif du contenu du chapitre. De plus des cours du langage Prolog s'intercaleront avec ce cours, afin que l'élève puisse avoir des applications informatiques concrètes des concepts et des méthodes de la Logique computationnelle.

[9] H. Zwirn : Compréhension et compression, La Recherche, décembre 2002, p.111

1

INTELLIGENCE, CONNAISSANCES ET LANGAGES

L'*intelligence*, au sens propre du mot, désigne l'ensemble de fonctions mentales qui permettent à un être vivant d'adapter son comportement à son environnement. Elle depend donc de la connaissance que cet être vivant a de son environnement. Elle suppose l'intégration au niveau du système nerveux des fonctions différentes, notamment des fonctions :

— de perception de l'environnement;
— de reconnaissance des situations;
— de décision d'actions;
— de mémorisation.

La tâche fondamentale, du point de vue des applications, pour l'IA est de comprendre l'intelligence afin de la rendre plus productive. Son objectif est donc le suivant :

Étant donné une tâche précise, dont l'exécution par l'homme requiert de l'intelligence, il s'agit de construire un logiciel permettant à un ordinateur d'exécuter la même tâche avec des résultats comparables à ceux obtenus par l'homme.

Deux cas sont à considérer.

Un premier où il existe un algorithme pour la résolution du problème posé. Bien que nous pouvons, sous certains aspects, envisager que cette approche peut faire partie de l'IA, nous la considérons pas ici.

L'autre cas est constitué par des problèmes pour lesquels soit il n'existe pas d'algorithme de résolution, soit il en existe mais il est très difficile à mettre en œvre. On cherche alors, à déterminer une stratégie de résolution, pas forcement optimale. On construit ainsi une méthode qui contient un ensemble de règles qui permettent d'effectuer, à différents stades et en fonction de la situation

du moment, des choix qui favorisent, mais sans pour autant le garantir, l'aboutissement à une solution. On appelle cette démarche une *heuristique*. La raison d'être de l'IA est fondée sur le postulat suivant :

Toute tâche cognitive peut être accomplie par un ordinateur programmé heuristiquement.

Un des axes majeurs de la démarche heuristique est constitué par la *représentation de connaissances*, à savoir la méthodologie d'organisation d'un vaste répertoire de données, de manière à pouvoir en extraire ce qui est nécessaire et pertinent au traitement d'une situation qui émerge d'un ensemble infini des possibilités. Bien sûr les outils à mettre au point pour accomplir cette tâche, n'imiteront pas nécessairement l'intelligence humaine, mais il faut qu'ils aient des performances non inférieures en précision et rapidité à celles de l'intelligence humaine.

Nous avons vu que l'intelligence humaine se fonde sur l'existence de la connaissance de l'environnement et l'exploitation de cette connaissance. Ce processus se fait de deux façons distinctes :

— une première qui consiste en l'assimilation, mémorisation et structuration des connaissances, et
— une seconde qui consiste en la décomposition et en la particularisation de la connaissance.

Par conséquent pour résoudre, de façon artificielle, un problème il faut d'une part stocker la connaissance et, d'autre part, trouver une méthode qui exploite la connaissance. Du fait que la connaissance

— est volumineuse;
— n'est pas facile à caractériser précisément;
— change constamment;

il faut que la méthode qui exploite la connaissance:

(1) Fasse surgir des généralisations. Dans ce cas il n'est pas nécessaire de représenter séparément chaque situation particulière, car les situations qui ont les mêmes propriétés importantes peuvent être groupées ensemble. Si la connaissance n'a pas cette propriété, il faut plus de place mémoire pour la stocker et plus de temps pour la traiter.
(2) Soit facile à modifier afin de corriger les erreurs et/ou de prendre en compte des modifications.
(3) Puisse être utilisée dans la majorité de cas, même si elle n'est pas totalement précise ou complète.

Remarquons que ce que nous avons mis en évidence sur la connaissance humaine nous pouvons le retrouver sur un ordinateur. En effet pour qu'un programme tourne sur un ordinateur, il lui faut des informations. L'exploitation des informations se fait de deux façons :

— par stockage de l'information;
— par recherche de l'information.

Ces deux façons d'exploitation des informations sont déjà utilisées par des logiciels algorithmiques opérant sur des données numériques (c-à-d. les logiciels classiques).

La seule différence, par rapport à ces logiciels, est que l'information relative à la connaissance a, comme nous le verrons au cours de l'Intelligence Artificielle, un caractère symbolique et non pas numérique. Par conséquent la méthode de résolution d'un problème de l'intelligence se traduira par la fabrication d'un logiciel non-algorithmique opérant sur des données symboliques. Le logiciel sera non-algorithmique car la résolution d'un problème d'intelligence ne peut pas être complètement explicitée et donc programmée au sens classique du terme compte tenu du nombre important de choix à effectuer et, de plus, tous les éléments ne sont pas connus a priori. On est donc amené à fournir à de tels logiciels un ensemble d'informations de type symbolique.

Il est bien évident qu'il faut aussi fournir à l'ordinateur les règles de gestion de cet ensemble d'informations symboliques.

La résolution donc, des problèmes dans le cadre de l'IA, s'effectue à l'aide de deux modules :

(1) Un premier qui concerne l'utilisation des informations (méthodes de recherche) : accès aux informations, combinaison des informations entre elles, etc.
(2) Un second module qui est la représentation des connaissances : regroupement des informations au sein d'une base de données, dont la structure tient compte des règles de recherche précédemment établies.

L'IA s'occupe essentiellement du deuxième module. Son objectif est de pouvoir exprimer la connaissance dans une forme qui puisse être utilisée par un ordinateur. Il faut, par conséquent, trouver un langage qui permet d'assimiler, par un ordinateur, la représentation des connaissances. Comme nous le savons, un langage est déterminé en fonction de deux aspects. Un aspect syntaxique qui permet de vérifier que les phrases du langage sont écrites correctement, en respectant les règles établies de syntaxe du langage. Et un aspect sémantique qui permet de vérifier la signification des phrases du langage. En réalité ce qui nous interesse principalement est l'aspect sémantique, c'est-à-dire le sens d'une phrase. Mais pour y acceder au sens d'une phrase, il faut que cette phrase soit correcte, ce qui nous amène à examiner sa forme syntaxique. Mais il est évident qu'une phrase peut être du point de vue syntaxique correcte et, malgré cela, dépourvue de sens.

Les exigences que nous pouvons formuler à ce stade pour le langage de représentation des connaissances concernent sa capacité d'être concis et sans ambiguïtés. Il faut aussi qu'il soit indépendant du contexte, en ce sens qu'il puisse exprimer des connaissances de toute sorte. Il y a plusieurs langages, surtout informatiques, qui peuvent répondre à ses caractéristiques. Mais il y a une autre exigence, très importante, concernant la représentation des connaissances, c'est le formalisme. Nous ne pouvons pas espérer manipuler efficacement les connaissances et obtenir des nouvelles connaissances si nous ne faisons pas une approche extrêmement formelle. Dès lors il devient évident qu'un des langages les plus appropriés pour la représentation et le traitement des connaissances est la logique mathématique selon une triple démarche :

(1) Utilisation de la logique mathématique pour modéliser et stocker la connaissance.
(2) Utilisation de la logique mathématique comme un langage pour communiquer avec l'ordinateur.

(3) Utilisation de la logique mathématique comme une méthode pour la vérification des programmes.

Le cours de la Logique Computationnelle & Prolog a comme objectif de donner la possibilité à l'élève de *modéliser et formaliser, en termes de la logique computationnelle, un problème* afin qu'il puisse d'une part d'écrire des programmes logiques pour la représentation et le traitement des connaissances et, d'autre part, de procéder à la vérification des programmes.

1.1 Références

Nous donnons ci-après la liste des livres qui nous ont servi pour la redaction de ces notes.

J.-M. ALLIOT, T. SCHIEX : *Intelligence Arificielle et Informatique Théorique*, Cépadués, 1993

K. R. APT & E.-R. OLDEROG : *Verification of sequential and concurrent programs*, Springer, 1991

E. W. BETH : *The Foundations of Mathematics*, North-holland, 1965

GEORGE S. BOOLOS, RICHARD J. JEFFREY : *Computability and logic*, Third edition, Cambridge U.P., 1989

STANLEY N. BURRIS : *Logic for mathematics and computer science*, Prentice Hall, 1998

JEAN CAVAILLÉS : *Méthode Axiomatique et Formalisme*, Hermann, 1981

ALONZO CHURCH : *Introduction to mathematical logic*, Princeton Univ. Press, 1956

RENÉ CORI, DANIEL LASCAR : *Logique mathématique*, 2 volumes, Masson, 1993

DIRK VAN DALEN : *Logic and Structure*, Third edition, Springer, 1994

MICHAEL R. GENESERETH, NILS J. NILSSON : *Logical Foundations of Artificial Intelligence*, M.Kaufmann, 1987

ROBERT LKOWALSKI : *Logic for Problem Solving, North-Holland, 1979*,

J. W. LLOYD : *Fondements de la programmation logique*, Eyrolles, 1988

PER MARTIN-LÖF: *Intuitioniostic Type Theory*, Bibliopolis, 1984

ANIL NERODE, RICHARD A. SHORE : *Logic for applications*, Second edition, Springer, 1997

ULF NILSSON, JAN MAŁUSZYŃSKI : *Logic, Programming and Prolog*, Wiley, 1990

JEAN PIAGET : *Essai de logique opératoire*, Dunod, 1972

HELENA RASIOWA, ROMAN SIKORSKI : *The Mathematics of Metamathematics*, Państwowe Wyd. Nauk., 1963

H. ROGERS, JR : *Theory of recursive functions and effective computability*, MIT Press, 1987

STUART J. RUSSEL, PETER NORVIG : *Artificial Intelligence. A Modern Approach*, Prentice-Hall, 1995

MICHAEL SPIVEY : *An introduction to logic programming through Prolog*, PrenticeHall, 1995

ROBERT I. SOARE : *Recursively enumerable sets and degrees*, Springer-Verlag, 1987

ALFRED TARSKI: *Introduction to Logic and to Methodology of Deductive Sciences*, Galaxy, 1965

R. TURNER : *Logics for artificial intelligence*, E.Horwood, 1984

JACQUES ZAHND : *Logique élémentaire,* Presses Polytechniques et Universitaires Romandes, 1998

2

OBJECTIFS ET MÉTHODES DE LA LOGIQUE

Dans ce chapitre nous présentons les objectifs généraux de la logique computationnelle. Comme cette logique est dérivée de la logique mathématique nous présentons d'abord les éléments fondamentaux de celle-ci et en particulier les systèmes de raisonnement qu'elle utilise.

2.1 La logique comme activité humaine

Comme nous venons de voir au chapitre précédent, le comportement intelligent d'un être vivant se resume à sa capacité de s'adapter aux changements de l'environnement. Pour ce faire il y a un prérequis, c'est la connaissance de cet environnement. Sa forme la plus élémentaire est la forme descriptive, c'est-à-dire celle qui présente la connaissance à l'aide des propositions (phrases) déclaratives. À partir de ces connaissances, on doit pouvoir, grâce au raisonnement, élaborer des nouvelles connaissances ou, encore, faire resurgir des connaissances qui étaient déjà présentes sous forme latente. C'est à ce stade qu'intervient la logique.

En effet, logique s'occupe des *raisonnements*.

Un raisonnement est un ensemble d'*hypothèses* qui conduit à une *conclusion*.

Le but de a logique est de vérifier si un raisonnement est correct, c-à-d. si, à partir des hypothèses, nous pouvons déduire la vérité de la conclusion.

Comme un exemple particulier de l'application de la logique, nous pouvons considérer qu'elle constitue un *médiateur* entre l'homme et l'ordinateur. En effet, un programme peut être assimilé à une proposition dont les instructions sont les hypothèses et le résultat attendu la conclusion. Nous pouvons donc utiliser la logique pour savoir si un programme produit le résultat escompté.

L'histoire de la logique est très longue. Aristote déjà a formalisé une logique, qu'on appelle aujourd'hui *logique des propositions* ou *logique d'ordre 0*. Il y aussi la *logique des prédicats* (prédicat = proposition qui contient des variables). Cette logique est aussi appelée *logique d'ordre 1*. Elle sera à la base de notre travail.

Ses domaines d'applications sont très divers. Citons en particulier

— Matériel : Lunité arithmétique et logique (UAL) est construite à partir de « portes logiques »
— Validation et vérification du logiciel : Z, B, model checking, diagnostic, ...
— Intelligence artificielle : aide à la décision, systèmes experts, web sémantique, ...
— Programmation : Prolog, différents prouveurs, ...

2.2 Construction de la langue logique

L'objectif de la logique est l'analyse des propositions et l'évaluation de la valeur de vérité de ces propositions. C'est une tâche très difficile, car les langues naturelles ont évolué d'une façon pas toujours conforme avec la logique, avec comme résultat d'avoir des propositions qui sont ambiguës quant à leur signification. Exemple : « je regarde l'homme avec le télescope ». Il y a aussi des phrases qui du point de vue de la forme sont similaires mais du point de vue de la logique sont différentes. Considérons par exemple les deux raisonnements suivants, empruntés à A. Church:

— J'ai vu un portrait de John Wilkes Booth. John Wilkes Booth a assassiné Abraham Lincoln. J'ai donc vu le portrait de l'assassin d'Abraham Lincoln.
— J'ai vu le portrait de quelqu'un. Quelqu'un a assassiné Abraham Lincoln. J'ai donc vu le portrait d'un assassin d'Abraham Lincoln.

Malgré la similitude de deux propositions il y a une grande différence en ce qui concerne le sens de chacune de ces propositions : la première proposition est vraie tandis que la seconde est en général fausse.

Pour éviter des situations comme les précédentes, il faut réduire de façon drastique la richesse de la langue tant du point de vue nombre de mots utilisés que du point de vue variétés de formes employées pour une phrase. L'objectif est de créer une langue logique aussi réduite que possible, qui permettra d'analyser la validité d'un raisonnement sans faire référence à son domaine d'application et de façon automatique en utilisant des algorithmes appropriés.

Avant d'entreprendre la construction de cette langue logique, il faut résoudre un problème : Comment construire une langue logique sans présupposer la logique ?

Parfois, en mathématiques, quand on est devant un problème difficile, on procède à de tours de passe-passe qui ont le mérite de déplacer le problème. Ainsi, dans ce cas précis nous allons utiliser une astuce : la « langue logique » sera considérée comme un objet, une langue-objet, qui sera élaborée en utilisant une langue naturelle, par exemple le français.

Donc ici le français est, par rapport à la langue-objet de la logique, une langue de niveau supérieur, c-à-d. le français est une *méta-langue* vis-à-vis de la langue logique.

[N.B. Méta-"quelque chose" signifie que "quelque chose" est de niveau supérieur, d'une plus grande généralité.

Donc le français comme métalangue de la logique, signifie que le français a une plus grande généralité que la langue logique. Par exemple, le français n'est pas seulement une langue logique !]

En jargonnant, on appellera dorénavant la langue-objet de la logique un *langage logique*.

On aboutit ainsi à la conclusion selon laquelle le français sera utilisé comme méta-langue pour élaborer le language logique.

Le problème donc est maintenant comment utiliser le français pour construire une langue logique et effectuer, avec son aide, des calculs ?

2.3 Constitution d'un langage logique

Puisque le langage logique doit être applicable à n'importe quelle situation, il faut qu'il soit formel, au contraire de la langue naturelle qui, bien sûr, elle est informelle. Comme à l'aide de ce langage, on doit pouvoir exprimer toute proposition dans n'importe quelle situation d'une quelconque activité humaine, il faut, dans ce langage, remplacer les mots par des symboles qui expriment des objets, des concepts ou, encore, des opérateurs propres à une activité humaine particulière. Pour ce faire, il faut, étant donnée une phrase exprimée dans la langue courante, de séparer, par une démarche d'abstraction, le contenu de la phrase de sa forme. En ne gardant que la forme de la phrase, nous pouvons construire un langage formel qui, en se complétant, sera en mesure d'exprimer la totalité des phrases que nous pouvons formuler en utilisant une langue.

Pour réaliser cette abstraction qui conduit à la formalisation, nous avons besoin de déterminer les ingrédients d'une phrase et aussi leur représentation formelle. Nous distinguons ainsi les éléments suivants pour une phrase :

Les noms Nous avons d'abord les *noms propres* qui peuvent indiquer :

- soit n'importe quelle personne qui porte ce nom, par exemple *Toto*;
- soit une personne particulière, par exemple *Berlioz*.

Remarquons que dans ce dernier cas on peut faire référence à un nom propre par périphrase, par exemple *le compositeur de la symphonie fantastique*. Cette remarque nous aide à comprendre qu'en logique un nom n'est pris en compte que par ce qu'il *dénote*, donc *Berlioz* et *le compositeur de la symphonie fantastique* dénotent la même personne. Malgré tout, la difficulté n'est pas complètement surmontée, car s'il est loisible (en suivant Bertrand Russel) de s'interroger si *Berlioz est le compositeur de la symphonie fantastique*, il est, par contre, sot de se poser la question si *Berlioz est Berlioz*. Il y a donc quelque chose de plus que la dénotation que véhicule un nom; c'est son *sens* d'où découlent d'ailleurs, l'existence et l'unicité de la dénotation.([1])

Nous avons aussi les *noms communs* qui, à leur tour, possèdent un sens et, par conséquent, dénotent ainsi quelque chose, qu'il s'agisse d'un objet, d'un être vivant ou d'un concept. La langue commune fait que beaucoup des noms communs ont des sens additionnels qui

[1] Le premier à faire cette distinction fut G. Frege en 1892. Selon lui un terme dénote (fait référence) à un individu

font que leur signification originelle dévie. Par exemple le nom commun *poirier* qui, dans sa signification originelle, dénote un arbre fruitier, ne retrouve pas ce sens quand on dit qu'un enfant fait *le poirier*.

Pour un nom commun la dénotation (ou référence) est le concept qui désigne, tandis que son

particulier. Le sens du terme est la condition qu'un individu particulier doit remplir pour être la dénotation (le référant).

sens est donné par sa valeur de vérité. Carnap([2]) en suivant Frege, propose de considérer comme dénotation d'un mot commun son *extension*, c-à-d. l'ensemble des objets qui sont désignés par ce mot et comme sens son *intension*, c'est-à-dire la fonction qui permet de savoir si un objet donné peut être référencé par ce mot.

En langage formel nous représenterons les noms propres, qui ont une seule dénotation, par des *constantes* qui seront notées $a, b, c, \ldots$. Nous réserverons les *variables* pour les noms communs ou les noms propres qui ont plusieurs dénotations, par exemple *Toto*. Les variables seront notées $X, Y, Z, \ldots$. Notez l'usage des lettres majuscules. La distinction entre constantes et variables se fait de cette façon : toute constante commence par une lettre minuscule et toute variable par une lettre majuscule. Remarquons que les valeurs numériques sont des constantes.

Les qualités Ce qui est dénoté par un nom, propre ou commun, a des qualités diverses qui s'expriment en utilisant d'autres noms. Ceux-ci sont bien entendu, des valeurs des ces qualités, considérées comme des fonctions sur celui-là. Par exemple *rouge* est la valeur de qualité *couleur* appliquée, comme une fonction, au *petit livre* (de celui que vous savez). Cette fonction a, comme n'importe quelle fonction mathématique, un domaine de définition et un domaine des valeurs qui est son image. Pour des raisons qui s'éclairciront par la suite (cf. paragraphe suivant), nous l'appellerons *foncteur*.

Les propriétés Indépendamment de ses qualités, ce qui est dénoté par un nom, propre ou commun, a aussi des propriétés. Une propriété quelconque appliquée sur un objet soit elle se vérifie, soit non. Elle peut donc être considérée comme une fonction sur l'objet dont l'image se réduit à l'ensemble {vrai, faux}. Une telle fonction s'appelle un *prédicat* ou, encore, *fonction propositionnelle* et qu'il ne faut pas confondre avec le foncteur défini précédemment. .

Les propositions Une proposition dans la langue courante est un assemblage des mots qui ont un sens et qui expriment une pensée. S'il s'agit d'une affirmation, nous avons une proposition sous forme déclarative. En langage formel la proposition joue le même rôle et nous utilisons uniquement des propositions sous forme déclarative. Dans la mesure où une proposition a un sens, elle peut être vraie ou fausse. Nous pouvons donc lui associer une fonction de vérité qui prend deux valeurs – vrai ou 1 et faux ou 0.

Nous pouvons envisager que les propositions peuvent être représentées par des variables qui prennent deux valeurs, vrai ou faux. Ce sont des *variable propositionnelle* et seront notées par $p, q, \ldots$. Remarquons que la formalisation d'une proposition conduit à la définition d'une variable propositionnelle.

La proposition est la forme la plus achevée du langage que nous utiliserons et, comme nous l'avons déjà indiqué, l'objectif de la logique est d'analyser ces propositions et d'évaluer leur valeur de vérité. Depuis Aristote on sait que cette évaluation dépend plutôt de la forme de la

[2] R. Carnap : *Meaning and necessity*, Un. Chicago Press, 1956

proposition que de son contenu([3]). L'évaluation se fait en utilisant différents types de raisonnement que nous présentons brièvement ci-après:

Raisonnement déductif C'est la forme la plus connue et la seule utilisée par la logique classique. Il permet, en partant d'une loi générale, d'obtenir – de déduire – des faits particuliers. Le fameux exemple

Tous les hommes sont mortels
Socrate est un homme
Donc, Socrate est mortel

est l'archetype du raisonnement déductif et qui de façon formelle s'écrit comme suit:

Tous les A sont B.
C est A.
Donc, C est B.

Pour la compréhension de la suite du chapitre nous transformons l'exemple précédent comme suit:

Tous les êtres vivants sur Terre sont mortels.
Socrate est un être vivant sur Terre.
Donc Socrate est un mortel.

Il faut remarquer que le raisonnement déductif ne permet pas d'obtenir des nouvelles connaissances. Il permet seulement de rationaliser et de codifier une démarche qui consiste à prendre un élément d'une famille et à lui attribuer une propriété que tous les éléments de la famille possèdent, en faisant ainsi resurgir une connaissance latente.

Raisonnement inductif Ce raisonnement est attribué à Bacon et il suit la démarche inverse du raisonnement déductif. À partir d'un ensemble des faits particuliers qui ont tous un élément en commun, on érige cet élément commun en loi générale. Par exemple:

Ces êtres vivants (Socrate, Héraclite, Parménide, ...) sont sur la Terre.
Ces êtres vivants (Socrate, Héraclite, Parménide, ...) sont mortels.
Donc, tous les êtres vivants qui sont sur la Terre sont mortels.

Sa traduction formelle est la suivante:

$A_1, A_2, \ldots, A_n$ *sont B.*
$A_1, A_2, \ldots, A_n$ *sont C.*
Donc, tout B est C.

Il est évident que pour appliquer ce raisonnement il faut disposer de plusieurs observations. Sa base scientifique est la loi de grands nombres. Néanmoins il faut se rappeler que l'induction permet d'établir des lois expérimentales mais non pas des lois théoriques. Ce raisonnement est donc à la base de la méthode expérimentale, c'est-à-dire à la base des sciences (physique, chimie, ...) où une loi reste vraie tant qu'il n'y ait pas un contre-exemple

[3] Il s'agit d'une situation qui n'est pas exceptionnelle en mathématiques. Par exemple, en théorie d'information, nous

savons que l'information d'un message ne dépend pas de son contenu mais de la complexité de sa forme. Nous pouvons,

en faisant des observations à d'autres disciplines, de s'apercevoir que souvent, en mathématiques, pour pouvoir faire un

calcul on doit abandonner la matérialité du contenu pour l'idéalité de la forme.

la refutant([4]). En ce sens il permet la production des nouvelles connaissances, qui peuvent éventuellement être potentiellement fausses.

Raisonnement abductif Il relie les effets aux causes et suggère une hypothèse. Par exemple:

Tous les êtres vivants qui sont sur la Terre sont mortels.
Ces êtres vivants (Socrate, Héraclite, Parménide, ...) sont mortels.
Donc, ces êtres vivants (Socrate, Héraclite, Parménide, ...) sont sur la Terrre.

que nous pouvons exprimer de manière formelle:

Tous les A qui sont B sont aussi C.
D est C.
Donc, D est B.

Ce type de raisonnement a été introduit par Peirce qui l'a défini comme étant une adoption probatoire d'une hypothèse. Dans son esprit il s'agissait d'un raisonnement inductif allegé. En effet le raisonnement inductif, en se fondant sur un ensemble d'observations qui ont toutes un élément commun, forge une observation générale relative à cet élément commun et, par conséquent, cette observation générale revêtirait le statut de règle générale permettant ainsi de tirer des conclusions. Par contre la conclusion d'un raisonnement abductif est une hypothèse choisie dans un ensemble d'hypothèses, en raison de sa plausibilité. Ainsi le fait de vivre sur Terre apparaît comme une cause de la mortalité des êtres vivants. Mais elle est une cause parmi d'autres.
Le raisonnement abductif a été appliqué au calcul logique par Kowalski et Kakas. Ce raisonnement construit aussi des nouvelles connaissances qui ont en réalité un statut très précaire.

Raisonnement par analogie Il s'agit du plus faible type de raisonnement. En effet il ne fournit aucune certitude et il est surtout utilisé en justice pour établir des jugements en exploitant la jurisprudence. Un exemple de raisonnement par analogie est le suivant.

Socrate est un être vivant sur Terre et qui est mortel.
Héraclite est un être vivant sur Terre et qui est comme Socrate.
Donc Héraclite est mortel.

L'aspect formel de ce raisonnement est le suivant

A est P.
B est similaire à A.
Donc, B est P.

Une forme encore plus attenuée du raisonnement par analogie est la suivante:

A et B sont P.
A est S.
Donc, B est S.

Ce raisonnement peut conduire à des erreurs et à des fausses croyances solidement ancrées dans la mesure ou elles sont, à l'instar des conclusions du raisonnement lui-même, non démontrables.

De tous les raisonnements que nous venons de voir, seulement le raisonnement déductif fournit la garantie de la validité du résultat.

[4] C'est justement cette remarque qui permet à Karl Popper de distinguer les sciences des croyances, car les premières

2.4 Logiques formelle et computationnelle

La logique est la branche des mathématiques qui étudie les lois de la pensée. Son objectif est de former de règles qui permettent de penser correctement. Transposée à l'informatique, la logique permettrait d'écrire des programmes corrects. Écrire un programme reviendrait ainsi à appliquer un ensemble de règles de la logique.

La *logique formelle* est une version decontextualisée de la logique. En tant que telle elle a :

— un langage formel;
— une syntaxe sans ambiguïtés;
— une sémantique précise, et
— des règles de formation des propositions.

Pour formaliser dans ce langage une phrase comme par exemple *Toto aime la logique* on doit utiliser différents types de symboles. Nous présenterons en detail ces symboles dans les deux chapitres suivants.

La représentation d'une phrase par un langage formel correspond à une représentation des connaissances. Si on veut faire un traitement des connaissances, il faut pouvoir faire un raisonnement en utilisant la représentation formelle de ces connaissances et appliquer les différents types de raisonnement. Pour que le traitement soit efficace il faut envisager son automatisation, c'est-à-dire avoir un procédé mécanique qui applique aux phrases de la logique, des règles de raisonnement d'une façon systématique.

La *logique computationnelle* est une branche qui se trouve au croisement de la logique mathématique et de l'informatique et dont son objectif est d'élaborer les bases théoriques pour la construction des tels procédés mécaniques qui, en réalité, sont ici des programmes informatiques. Elle recupère donc toutes les préoccupations de la logique formelle qui sont relatives aux approches syntaxique et sémantique des raisonnements, auxquelles elle rajoute sa propre préoccupation concernant l'efficacité du raisonnement.

Plus particulièrement la logique computationnelle joue un rôle important au *test des modèles* (model-checking) qui est actuellement une de techniques les plus utilisées pour la vérification et le déverminnage des programmes. Dans ce cadre la logique examine ce qui peut être écrit dans un langage formel - et qui fait partie du domaine de la syntaxe - et la façon dont ce qui est écrit, est interprété par un modèle concret - ce qui constitue le domaine de la sémantique.

L'utilisation de la logique est devenu indispensable dès qu'on a introduit l'ordinateur à l'automatisation des diverses tâches. Prenons l'exemple de la conduite d'une rame de métro. Si le conducteur est un être humain, il dispose d'un certain nombre d'actionneurs - frein, vitesse, ouverture/fermeture des portes, ... - sur lesquels agit directement. Si par exemple il rencontre un feu rouge, il appuiera sur le frein. Cette action est conséquence d'une démarche logique qui est issue d'un modèle interne propre au conducteur et qui représente, sous forme symbolique, le monde réel dans lequel se trouve le conducteur et la rame de métro. Si ce modèle interne est faux, le conducteur fera des bêtises, ce qui peut arriver, par exemple, à quelqu'un qui n'a jamais

sont, selon lui, réfutables au contraire des secondes.

conduit une rame. Et celui qui apprend à conduire, est en train en fait de construire un modèle interne de plus en plus exact. Si on remplace le conducteur par une conduite automatique, on a, en réalité, remplacé le conducteur par un ordinateur qui, muni d'un logiciel, peut, à l'aide d'un certain nombre de dispositifs, agir sur les actionneurs. Le logiciel n'est pas forcement la replique exacte du modèle interne du conducteur. En effet, le logiciel est écrit à l'aide d'un langage formel et il est composé des systèmes mathématiques avec des propriétés bien définies. Nous avons ainsi un *modèle logique* de la réalité qui n'est pas le même avec le modèle interne du conducteur. Mais, malgré cette différence, il faut, devant une situation donnée, que les deux modèles réagissent de la même façon. Il faut donc savoir si le programme informatique se comporte chaque fois comme il faut qu'il se comporte. L'examen de la conformité du comportement des programmes avec les exigences de son utilisation constitue le *test des modèles*.

3

CALCUL PROPOSITIONNEL

Pour décrire un environnement donné, que nous appellerons par la suite *univers du discours* (noté $\mathcal{U}$), la logique utilise des phrases que nous appellerons *propositions logiques* ou encore *formules*. Le but du calcul logique est de donner un fondement à ces propositions c'est-à-dire examiner leur validité. Cet examen de la validité d'une formule se fera indépendamment du contexte dans lequel la formule a été élaborée et le résultat sera en rapport avec sa propriété d'être vraie ou fausse. Nous commençons l'étude de la logique par l'étude des propositions. Nous allons d'abord définir un langage qui permettra la construction des propositions, c-à-d. un alphabet et des mécanismes qui permettent de créer des propositions. Viendra ensuite l'étude sémantique et syntaxique des propositions créées, c'est-à-dire l'étude du sens et de la preuve des propositions.

3.1 Éléments du langage

Les propositions seront représentées par des symboles qui auront une valeur de vérité : *vraie, fausse*. Pour leur étude logique nous allons définir un *langage formel* $\mathcal{L}_0$ à l'aide des éléments suivants :

— Un ensemble V_p, au plus dénombrable, des *propositions* qui seront notées par $p, q, \ldots$. No-

tons que l'« appellation contrôlée » des propositions est *variables propositionnelles* terme que nous n'utiliserons pas, de peur d'introduire une confusion en ce qui concerne le sens de la variable. On pourra par contre utiliser le nom – moins usité – de *propositions atomiques*.

— Un ensemble Ξ, au plus dénombrable, des *constantes*.

— Un ensemble L des *connecteurs* qui sont les suivants :

  – *Connecteur logique* unaire : la négation $\neg$
  – *Connecteurs propositionnels* binaires :
    - Disjonction : $\vee$
    - Conjonction : $\wedge$
    - Implication : $\rightarrow$
    - Équivalence (ou double implication) : $\leftrightarrow$

— Les séparateurs : parenthèses gauche "(" et droite ")", crochets gauche "$[$" et droit "$]$".

Les séparateurs ne font pas partie, à proprement parler, du langage. Leur présence permet de faciliter la lecture des propositions.

Les éléments du langage déterminent un *alphabet*

$$\Sigma_0 = \{V_p, \Xi, L\}$$

La brique élémentaire du calcul propositionnel est l'*atome*. Un proposition atomique est un atome. Une constante aussi. Plus généralement

DÉFINITION 3.1.1 *Un* atome *est une proposition dont la structure interne ne nous préoccupe pas.*

Par abus de notation, les atomes seront notés par la suite avec les ettres $p, q, r, \ldots$, c'est-à-dire de la même manière que les propositions bien qu'ils puissent être aussi des constantes. Implicitement on accepte donc qu'une constate puisse être vue comme une proposition, ce qui est d'ailleurs la réalité. Si nous maintenons la distinction entre constates et propositions, que les puristes pourraient nous reprocher, c'est uniquement pour des raisons de compatibilité avec le contenu du chapitre suivant.

À partir des atomes on peut construire des *formules bien formées* (fbf) dont la définition est la suivante :

DÉFINITION 3.1.2 *Une* formule bien formée *est*

— *soit un atome*

— *soit une proposition obtenue à partir des fbf A et B, selon les constructions suivantes :*

  – $\neg A$
  – $A \vee B$, $A \wedge B$
  – $A \rightarrow B$, $A \leftrightarrow B$

Selon cette définition, les propositions atomiques sont des fbf. Les formules bien formées seront notées par les lettres $A, B, C, \ldots$.

Si on note par F_0 le plus petit ensemble de fbf que nous pouvons construire selon la définition 3.1.2, alors la paire

$$\mathcal{L}_0 = \{\Sigma_0, F_0\}$$

est le *langage d'ordre zéro* ou le *langage du calcul propositionnel*. Par construction F_0 est est un ensemble dénombrable, défini récursivement.

Les principales propriétés des connecteurs sont données ci-après :

(1) Double négation ou involution

$$p \leftrightarrow \neg\neg p$$

(2) Loi de de Morgan

$$\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$$
$$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$$

(3) Définition de l'implication

$$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$$

(4) Introduction de l'implication

$$p \rightarrow (q \rightarrow p)$$

(5) Distributivité de l'implication

$$(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$$

(6) Contradiction

$$(p \rightarrow q) \rightarrow ((p \rightarrow \neg q) \rightarrow \neg p)$$

Étant donnée une proposition, nous pouvons en construire trois autres qui dérivent de celle-là comme suit :

DÉFINITION 3.1.3 *Considérons la proposition $P_1 : p \rightarrow q$.*
La proposition $P_2 : \neg q \rightarrow \neg p$ est la proposition contrapositive *de P_1.*
La proposition $P_3 : q \rightarrow p$ est la proposition inverse *(ou* réciproque*) de P_1.*
La proposition $P_4 : \neg p \rightarrow \neg q$ est la négation *de P_1.*

ASCÈSE 3.1 *Soient les deux propositions :*

p_1 *Aujourd'hui on est le 13 du mois.*

p_2 *Demain on sera le 14 du mois.*

Exprimer en logique des propositions et en français les quatre propositions : directe, contrapositive, inverse et négation.

3.2 Proposition, énoncé et vérité

L'objectif de la logique est de déterminer la valeur de vérité d'une fbf – aussi complexe soit-elle – en se fondant sur la valeur de vérité de ses atomes et sur les tables de vérité des connecteurs logiques.

Le premier problème que l'on rencontre est dû au fait que la langue naturelle (ici le français) peut exprimer à la fois une proposition et sa vérité. Ce fait a des conséquences fâcheuses, en engendrant des paradoxes, comme par exemple le paradoxe du menteur : la proposition "je mens" vaut pour elle-même.

L'archétype des paradoxes est le paradoxe des classes établi par Russell : Soit $\mathcal{G}$ la classe des classes Y qui ne s'appartiennent pas : $\mathcal{G} = \{Y : Y \notin Y\}$ ce qui en langage logique s'écrit $(\forall Y)(Y \in \mathcal{G} \leftrightarrow Y \notin Y)$. Supposons que $\mathcal{G}$ est une classe comme les autres. On peut alors la substituer à la variable (classe) Y et la formule précédente s'écrit : $(\mathcal{G} \in \mathcal{G} \leftrightarrow \mathcal{G} \notin \mathcal{G})$ ce qui est absurde.

On s'en sort (pas très bien) en considérant que les valeurs de vérité Vrai et Faux font partie du méta-langage et non pas du langage logique.

Il faut aussi examiner la nature d'une proposition en logique, qui n'est pas la même chose qu'en langue courante. En effet, en logique une proposition est un énoncé dont on ne tient pas compte ni de la personne qui l'exprime, ni du temps, ni de toute autre chose, à l'exception de l'information sur l'état des choses. Ainsi la proposition "j'affirme que la fenêtre est fermée aujourd'hui on ne retient que "la fenêtre est fermée".

Il faut faire attention à ne pas confondre proposition et énoncé. En effet une même proposition peut être exprimée par plusieurs énoncés différents mais synonymes.

Toute proposition peut être :

— démontrée syntaxiquement en termes de correction grammaticale;

— interprétée sémantiquement en termes des valeurs de vérité.

Donc une proposition est un énoncé déclaratif grammaticalement correct susceptible d'être Vrai ou Faux.

La notion de la vérité n'est pas non plus une notion logique. En effet la vieille idée d'Aristote selon laquelle une proposition est vraie si elle correspond (elle est l'image) d'un fait n'est pas admise de nos jours car (d'après Frege) nous ne pouvons pas mettre en correspondance deux objets de nature différente (proposition – fait). Donc la vérité est logiquement indéfinissable (démarche analogue à la théorie de l'information, où le concept de l'information est mathématiquement indéfinnissable). Selon Russell "les propositions sont vraies ou fausses comme les roses sont roses ou rouges".

3.3 Interprétation sémantique – Modèles

La première tâche du calcul propositionnel est de donner un sens aux fbf, c'est-à-dire d'établir une correspondance entre les fbf et les faits de l'univers du discours. Ces faits sont connus par celui qui fournit la fbf. En effet, il faut admettre qu'en logique une fbf ne signifie, par elle-même, rien. Elle a un sens – si elle en a un – en fonction de l'*interprétation* faite par celui qui émet cette fbf.

Nous devons donc, pour évaluer le sens d'une fbf, établir seulement la valeur de vérité de cette fbf étant donnée l'interprétation adéquate. Pour ce faire, nous adoptons l'hypothèse fondamentale suivante :

Tout atome peut prendre deux valeurs : vrai – 1 et faux – 0 et la valeur de vérité d'une fbf est complètement déterminée par la valeur de chacun de ses atomes.

Concrètement à chaque atome d'une fbf on associe une valeur de vérité, selon l'interprétation que l'on donne à cet atome, et à chaque connecteur on associe une table de vérité fixe qui, en fonction de la valeur de vérité de ses arguments, fournit la valeur de vérité de l'opération effectuée par le connecteur. Dans la table ci-après nous donnons les valeurs de vérité pour tous les connecteurs du langage.

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	1	0	0	0
1	1	0	1	1	1	1

ASCÈSE 3.2 *Traduire les phrase suivantes*

(1) J'aime les pâtes soit à la tomate, soit à l'ail.

(2) Vous n'êtes pas sans savoir.

(3) Vous n'êtes pas sans ignorer.

(4) Si tu es sage, tu auras une glace.

L'utilisation des tables de vérité pour le calcul de la valeur d'une fbf est une méthode sémantique. On verra par la suite des méthodes syntaxiques.

EXEMPLE 3.3.1 *Vérifions que nous avons toujours* $(p \to q) \leftrightarrow (\neg p \vee q)$

p	q	$\neg p$	$p \to q$	$\neg p \vee q$	$(p \to q) \leftrightarrow (\neg p \vee q)$
0	*0*	*1*	*1*	*1*	*1*
0	*1*	*1*	*1*	*1*	*1*
1	*0*	*0*	*0*	*0*	*1*
1	*1*	*0*	*1*	*1*	*1*

La dernière colonne de la table de vérité est toujours égale à 1, ce qui montre que la formule est toujours vérifiée.

ASCÈSE 3.3 *Vérifier si la formule*

$$p \to (q \vee (r \leftrightarrow (r \to \neg p)))$$

est une fbf et établir sa valeur.

ASCÈSE 3.4 *Établir une propsition* $p(a,b,c)$ *qui a la même valeur de vérité que la majorité de ses arguments.*

La valeur de vérité associée à un atome peut changer en fonction de la signification de cet atome. Par exemple si p est la proposition " la terre est ronde", alors la valeur de vérité de p est 1. Par contre si p est la proposition "la terre est un cube" sa valeur de vérité est 0. Associer une proposition abstraite p à une proposition concrète, comme nous venons de faire, c'est donner une *interprétation* à p. Plus généralement, une interprétation d'une fbf est le résultat d'une spécification de la signification, c-à-d. d'une interprétation, de chaque atome de la fbf. Formellement

DÉFINITION 3.3.1 *Considérons un ensemble d'atomes propositionnels noté* Δ. *Ainsi* Δ *forme une* base de données (BdD). *On appelle* valuation *de l'ensemble des Ã©lÃ©ments de la BdD* Δ, *une fonction* $\varphi : \Delta \to \{0,1\}$
Si A est une fbf, on note par Δ_A *l'ensemble d'atomes propositionnels de A et si* $card\,\Delta_A = n$, *alors une valuation de A, qui sera notée* $\varphi(\Delta_A)$ *est un élément de* $\{0,1\}^n$.

La notion de la valuation permet de définir celle de l'interprétation :

DÉFINITION 3.3.2 *Considérons une fbf A et soit* Δ_A *la BdD de ses atomes propositionnels. Si à chaque atome propositionnel on associe une interprétation, alors nous obtenons une valuation de* Δ_A, *c'est-à-dire une application* $\varphi : \Delta_A \to \{0,1\}^n$. *Cette valuation peut être vue comme une interprétation* $\mathcal{I}$ *de la fbf A. La valeur de l'interprétation, qui sera notée* $\varphi_{\mathcal{I}}(A, \Delta_A)$ *ou, encore, plus brièvement,* $\varphi_{\mathcal{I}}(A)$ *est la valeur de la fbf A si on lui applique la valuation* φ *et elle est un élément de* $\{0,1\}$.

EXEMPLE 3.3.2 *Considérons la fbf $A = p \vee q$. Alors $\Delta_A = \{p, q\}$ et soit une valuation (interprétation) φ de Δ_A telle que $\varphi(p) = 1$ et $\varphi(q) = 0$. Alors la valeur de cette interprétation sur A est $\varphi_I(A) = 1$.*

Si Δ contient n atomes propositionnels, alors on a au plus 2^n interprétations possiblesde A. (Attention, elles ne sont pas toutes différents ! Vérifier avec l'exemple précédent). Si la valuation d'une interprétation particulière $\varphi_{\mathcal{I}}$ a la valeur 1, on dit que la valuation *satisfait* A et l'on note $\varphi_{\mathcal{I}}(A) = 1$. Il est évident que toute valuation d'une fbf A n'est pas susceptible de satisfaire à A, c'est-à-dire de conférer à A la valeur de vérité « vrai ». Nous avons ainsi :

DÉFINITION 3.3.3 *Soit A une fbf et $\mathcal{I}$ une interprétation. On dit que A est* satisfiable *par $\mathcal{I}$ ou que A est une* conséquence sémantique *de I, et l'on note $\mathcal{I} \models A$, si l'une de situations suivantes est vérifiée :*

- *si $A \in V_p$, alors $\mathcal{I} \models A$ ssi $\varphi_{\mathcal{I}}(A) = 1$*
- *si A est de la forme $(\neg B)$, alors $\mathcal{I} \models A$ ssi $\mathcal{I} \models / B$*[1]
- *si A est de la forme $(B \wedge C)$, alors $\mathcal{I} \models A$ ssi $\mathcal{I} \models B$ et $\mathcal{I} \models C$*
- *si A est de la forme $(B \vee C)$, alors $\mathcal{I} \models A$ ssi $\mathcal{I} \models B$ ou $\mathcal{I} \models C$*
- *si A est de la forme $(B \rightarrow C)$, alors $\mathcal{I} \models A$ ssi soit $(\mathcal{I} \models / B)$, soit $(\mathcal{I} \models C)$*
- *si A est de la forme $(B \leftrightarrow C)$, alors $\mathcal{I} \models A$ ssi soit $(\mathcal{I} \models / B)$ et $(\mathcal{I} \models / C)$, soit $(\mathcal{I} \models B)$ et $(\mathcal{I} \models C)$*

Il est à noter que dans un table de vérité, les colonnes représentent des fbf et les lignes des interprétations.

ASCÈSE 3.5 *Considérons les fbf suivantes :*

(1) $p \wedge q$

(2) $p \wedge (\neg r \vee q)$

(3) $p \rightarrow q$

(4) $q \rightarrow p$

(5) $(p \vee q) \wedge (q \vee r)$

et l'interprétation I suivante :

$$\phi_I(p) = 1,\ \phi_I(q) = 0,\ \phi_I(r) = 1$$

Donner la valeur de vérité des formules précédentes selon l'interprétation I.
Existe-t-il une interprétation qui rende toutes les formules vraies ?

Nous introduisons maintenant la notion du modèle pour une fbf.

DÉFINITION 3.3.4 *Soit A une fbf et $\mathcal{I}$ une interprétation. Si $\mathcal{I}$ rend A satisfiable, c'est-à-dire si $\mathcal{I} \models A$, alors $\mathcal{I}$ est un* modèle *pour A que l'on note par $M(A)$. On dit aussi que A est vraie dans $\mathcal{I}$.*
On note par $\mathcal{M}$ un ensemble de modèles pour A. Alors $\forall \mathcal{I} \in \mathcal{M}$ on a $\mathcal{I} \models A$ ce qui permet de noter $\mathcal{M} \models A$.

[1] $\models /$ signifie non satisfiable et il est un symbole extra-logique.

De cette définition on peut en conclure que si $\mathcal{I}$ est un modèle pour A, alors dans la table de vérité pour A, la ligne qui correspond à la valuation $\varphi_{\mathcal{I}}$ aura la valeur 1 à la colonne correspondante à A.

La notion du modèle donnée avec la définition précédente pour une fbf, peut être étendue à un ensemble des fbf F. On dit que l'interprétation $\mathcal{I}$ est un modèle pour F si $\mathcal{I}$ est un modèle pour chaque fbf de F.

ASCÈSE 3.6 *Considérons une situation selon laquelle une alarme d'une maison se déclenche, si elle n'est pas en panne, quand il y a un tremblement de terre ou un cambriolage ou les deux à la fois.*

(1) Exprimer le déclenchement de l'alarme à l'aide de la logique propositionnelle.

(2) Trouver, s'il en existe, un modèle pour la formule établie précédemment.

(3) Pour chacune des situations suivantes, trouver, s'il en existe, en modèle :
- *(a) Tremblement de terre.*
- *(b) Cambriolage $\longrightarrow \neg$Cambriolage*
- *(c) (Cambriolage $\rightarrow \neg$Cambriolage) $\wedge$ Cambriolage*

Nous introduisons maintenant quatre notions qui découlent de l'interprétation.

DÉFINITION 3.3.5 *Une fbf A qui est vraie pour toute interprétation est appelée* tautologie *(ou* formule valide*) et sera notée par $\models A$.*
Une fbf A pour laquelle il y a au moins une interprétation I qui la satisfait (c'est-à-dire que I est un modèle pour A) est appelée satisfiable *ou* (sémantiquement) consistante.
Une fbf A pour laquelle il existe une interprétation I telle que $\mathcal{I} \models /A$ est appelée falsifiable. *Une fbf qui est fausse dans toute interprétation est appelée* insatisfiable *ou* sémantiquement inconsistante[2]*.*

Les algorithmes qui à partir d'un ensemble des fbf engendrent des tautologies s'appellent *systèmes de preuve* ou *prouveurs*. Un prouveur est *cohérent* s'il engendre seulement des tautologies. Il est *complet* s'il engendre toutes les tautologies.

Pour un ensemble de fbf, nous avons la définition suivante :

DÉFINITION 3.3.6 *Un ensemble des fbf est* mutuellement exclusif *si et seulement si chaque interprétation satisfait au plus à une fbf.*
Un ensemble des fbf est exhaustif *si et seulement si chaque interprétation satisfait au moins à une fbf.*

ASCÈSE 3.7 *(1) Montrer que l'ensemble de fbf $\mathcal{F} = \{p \wedge q,\ q \vee r\}$ est satisfiable.*

(2) Montrer que l'ensemble de fbf $\mathcal{F} = \{p \wedge q,\ q \wedge r,\ \neg r\}$ est insatisfiable.

Dans le cas où A est une tautologie, dans la table de vérité de A, la colonne qui correspond à A contient seulement de 1.

Remarquons que si A est une tautologie, alors $\neg A$ est sémantiquement inconsistante et vice-versa.

ASCÈSE 3.8 *Pour les fbf suivantes préciser leur nature (tautologie, satisfiable ou insatisfiable).*

[2]Certains auteurs préfèrent le terme *antilogie* ou, encore, *contradiction* pour ce type de fbf.

(1) $p \to (q \to p)$

(2) $q \wedge (q \to p) \to p$

(3) $\neg p \to (p \vee q)$

(4) $p \wedge \neg(p \vee q)$

Deux fbf sont équivalentes si pour toute interprétation elles prennent la même valeur de vérité. La tautologie permet de formaliser la relation d'équivalence entre deux fbf :

DÉFINITION 3.3.7 *Deux fbf A et B sont* équivalentes*, et l'on note par $A \equiv B$, si et seulement si la fbf $A \leftrightarrow B$ est une tautologie.*

La relation $A \equiv B$ entre deux fbf est une relation d'équivalence en ce sens que : $A \equiv B \to B \equiv A$ et $A \equiv B \wedge B \equiv C \to A \equiv C$.

ASCÈSE 3.9 *Soit une fbf P et soit P' la proposition contrapositive de P. Montrer que P' est équivalente à P.*

On trouve dans la littérature une liste impressionnante des tautologies.Voici un extrait des tautologies célèbres :

(1) Loi du syllogisme

$$(p \to q) \to ((q \to r) \to (p \to r))$$

(2)

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$$

(3) Introduction de *ou*

$$p \to (p \vee q) \ , \ q \to (p \vee q)$$

(4)

$$(p \to r) \to ((q \to r) \to ((p \vee q) \to r))$$

(5) Élimination de *et*

$$(p \wedge q) \to p \ , \ (p \wedge q) \to q$$

(6)

$$(r \to p) \to ((r \to q) \to (r \to (p \wedge q)))$$

(7) Loi de l'importation

$$(p \to (q \to r)) \to ((p \wedge q) \to r)$$

(8) Loi de l'exportation

$$((p \wedge q) \to r) \to (p \to (q \to r))$$

(9) Loi de Duns Scotus

$$(p \wedge \neg p) \to q$$

(10)

$$p \to (p \wedge \neg p) \to \neg p$$

(11) Loi du tiers exclu (tertium non datur)

$$p \vee \neg p$$

(12) Introduction de l'absurde $\perp$

$$(p \wedge \neg p) \to \perp$$

(13) Réduction à l'absurde

$$(\neg p \to \perp) \to p$$

(14)

$$(p \to q) \to ((p \to \neg b) \to \neg p)$$

L'exemple classique de fbf insatisfiable est le suivant :

$$p \leftrightarrow \neg p$$

ASCÈSE 3.10 *Vérifier que*

$$\neg (q \to p) \equiv q \wedge \neg p$$

Nous donnons dans la suite une liste numérotée avec des formules équivalentes. Toutes les formules du même numéro sont équivalentes.

(1) $p \to q$, $\neg p \vee q$, $\neg q \to \neg p$, $p \wedge q \leftrightarrow p$, $p \wedge q \leftrightarrow q$

(2) $\neg (p \to q)$, $p \wedge \neg q$

(3) $p \leftrightarrow q$, $(p \wedge q) \vee (\neg p \wedge \neg q)$, $(\neg p \vee q) \wedge (p \vee \neg q)$, $(p \to q) \wedge (q \to p)$, $\neg p \leftrightarrow \neg q$, $(p \vee q) \to (p \vee q)$

(4) $\neg (p \leftrightarrow q)$, $p \leftrightarrow \neg q$, $\neg p \leftrightarrow q$

(5) p , $\neg \neg p$, $p \wedge p$, $p \vee p$, $p \vee (p \wedge q)$, $p \wedge (p \vee q)$, $\neg p \to p$, $(p \to q) \to p$, $(q \to p) \wedge (\neg q \to p)$

(6) $\neg p$, $p \to \neg p$, $(p \to q) \wedge (p \to \neg q)$

(7) $p \wedge q$, $q \wedge p$, $p \wedge (\neg p \vee q)$, $\neg (p \wedge \neg q)$

(8) $p \vee q$, $q \vee p$, $p \vee (\neg p \wedge q)$, $\neg a \to q$, $(p \to q) \to q$

(9) $p \to (q \to r)$, $(p \wedge q) \to r$, $q \to (p \to r)$, $(p \to q) \to (p \to r)$

(10) $p \to (q \wedge r)$, $(p \to q) \wedge (p \to r)$

(11) $p \to (q \vee r)$, $(p \to q) \vee (p \to r)$

(12) $(p \wedge q) \to r$, $(p \to r) \vee (q \to r)$

(13) $(p \vee q) \to r$, $(p \to r) \wedge (q \to r)$

(14) $p \leftrightarrow (q \leftrightarrow r)$, $(p \leftrightarrow q) \leftrightarrow r$

Les lignes numérotées 10 et 11 montrent qu'il y a distributivité à gauche de l'implication par rapport à la conjonction et la disjonction et les deux lignes suivantes montrent que cette distributivité disparaît lorsque l'implication passe à droite.

On termine ce paragraphe par la notion de conséquence valide.

DÉFINITION 3.3.8 *Soit la fbf B dont les atomes font partie des atomes des fbf $A_1, A_2, \ldots, A_n$. B est une* conséquence valide *des $A_1, A_2, \ldots, A_n$, que l'on note par $A_1, A_2, \ldots, A_n \models B$, si B prend la valeur 1 quand tous les A_i, qui contiennent des atomes de B, sont simultanément à 1.*

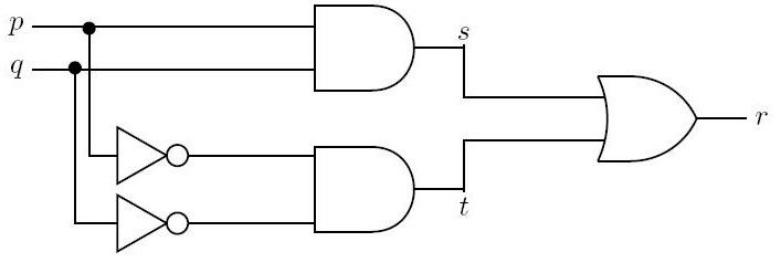
On dit aussi que $A_1, A_2, \ldots, A_n$ impliquent logiquement B. De manière plus simple on peut dire que $A \models B$ ssi tout modèle de A est aussi modèle de B. Si, de plus, on a que tout modèle de A est aussi un modèle de B et tout modèle de B est un modèle de A, alors A et B sont équivalents : $A \equiv B$.

Dans la suite il faut faire attention au fait que le même symbole, à savoir $\models$, est utilisé pour indiquer deux relations différentes :

— la relation de satisfiabilité qui relie une interprétation à une fbf;

— la relation de conséquence valide qui relie deux fbf entre elles.

ASCÈSE 3.11 *Considérons le circuit suivant*



Montrer que ce circuit vérifie la proposition

$$[(p \vee q) \rightarrow \neg(\neg p \vee \neg q)] \rightarrow r$$

Nous avons le

THÉORÈME 3.3.1 *$A \models B$ est équivalent à $\models (A \rightarrow B)$.*

ASCÈSE 3.12 *Traduire et évaluer la phrase suivante :*

Tu ne dis pas où tu te trouves et si Toto ne le dit pas, alors Koko le dira si et seulement si on lui promet de lui acheter une glace.

L'ascèse suivant introduit l'analyse logique d'un programme.

ASCÈSE 3.13 *Soient les deux programmes suivants :*

Programme 1

```
Si (x > 0) et (y > 0) alors
     afficher ('Produit x*y positif')
Finsi
```

Programme 2

```
Si (x > 0) alors
    si (y > 0) alors
        afficher ('Produit x*y positif')
    Finsi
Finsi
```

Montrer que ces deux programmes sont sémantiquement équivalents.

3.4 Modèles et connaissances

Nous pouvons utiliser les modèles $\mathcal{M}(A)$ d'une fbf A pour obtenir des connaissances sur l'univers du discours. Nous avons les résultats suivants :

- $\mathcal{M}(A \wedge B) = \mathcal{M}(A) \cap \mathcal{M}(B)$
- $\mathcal{M}(A \vee B) = \mathcal{M}(A) \cup \mathcal{M}(B)$
- $\mathcal{M}(\neg A) = \mathcal{M}(A)^C$

Nous avons aussi que

- La fbf A est satisfiable ssi $\mathcal{M}(A) \neq \emptyset$.
- Soient deux fbf A et B. Nous avons $A \rightarrow B$ ssi $\mathcal{M}(A) \subset \mathcal{M}(B)$.
- A est équivalente à B ssi $\mathcal{M}(A) = \mathcal{M}(B)$
- Les fbf $A_1, \ldots, A_n$ sont mutuellement exclusives ssi $\mathcal{M}(A_i) \cap \mathcal{M}(A_j) = \emptyset \; ; \; \forall i \neq j$.

De plus nous pouvons utiliser les modèles pour examiner la sémantique d'une fbf. Supposons par exemple que nous avons une fbf A et que nous connaissons les modèles $\mathcal{M}(A)$ pour A. Si notre connaissance est enrichie d'une nouvelle fbf B, alors nous connaissons les modèles $\mathcal{M}(A \wedge B) = \mathcal{M}(A) \cap \mathcal{M}(B)$ qui est une connaissance plus précise que la précédente. Nous arrivons à un état complet de connaissance si toutes les interprétations sont fausses, sauf une qui constitue le seul modèle correspondant aux fbf utilisées.

ASCÈSE 3.14 *Considérons la situation de l'ascèse 3.6. Notre état de connaissances sont les huit interprétations possibles. Si on nous donne la fbf*

A : (*Tremblement de terre* $\wedge$ *Cambriolage*) $\rightarrow$ *Alarme*

que devient notre état de connaissances ?

Supposons qu'une nouvelle fbf

B : *Tremblement de terre* $\rightarrow$ *Cambriolage*

De quelle manière évolue notre état de connaissances ?

Nous avons le

THÉORÈME 3.4.1 *Soient $\mathcal{M}$ et $\mathcal{M}'$ deux ensembles de modèles tels que $\mathcal{M}' \subseteq \mathcal{M}$. Si $\mathcal{M} \models A$, alors $\mathcal{M}' \models A$.*

ASCÈSE 3.15 *Appliquer le théorème précédent à l'ensemble des modèles :*

	p	q
v_1	0	0
v_2	0	1
v_3	1	0
v_4	1	1

et vérifier qu'en restreignant l'ensemble de modèles on devient capable d'inférer des propositions plus fortes (plus restrictives).

3.5 Évaluation syntaxique – Démonstration

L'évaluation syntaxique est un procédé mécanique qui permet de déduire le bien fondé d'une formule indépendamment du sens de ses composantes. Pour ce faire on s'appuie sur un ensemble d'axiomes et des règles d'inférence. Considérée de cette façon, l'évaluation syntaxique est une théorie de la démonstration. La première notion de la théorie de démonstration est le théorème dont voici sa définition.

DÉFINITION 3.5.1 *Une fbf A est un* théorème, *et l'on note $\vdash A$, si A est un axiome ou si A est obtenue par application des règles d'inférence sur d'autres théorèmes.*

Les axiomes de la logique sont les principes fondamentaux de la logique, c'est-à-dire des propositions du langage qui sont vraies en vertu uniquement de leur syntaxe. Le calcul propositionnel possède trois axiomes :

A1.- Introduction de l'implication

$$A \rightarrow (B \rightarrow A)$$

A2.- Distributivité de l'implication

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

A3.- Négation

$$(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$$

où, encore

$$(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$$

Remarquons que ces axiomes sont en fait des schémas d'axiomes, en ce sens qu'à chaque tel schéma correspond une infinité d'axiomes. Par exemple pour le schéma A1, en remplaçant A par $A \rightarrow C$ et B par $B \rightarrow C$, on peut avoir l'axiome : $(A \rightarrow C) \rightarrow ((B \rightarrow C)\,(A \rightarrow C))$.

Les règles d'inférence sont les règles utilisées par le mécanisme de démonstration. Le calcul propositionnel, et en particulier Prolog, utilise la règle d'inférence suivante :

R1.- Modus ponens (règle du détachement)

$$\frac{\vdash A, \vdash A \to B}{\vdash B}_3$$

Pour faciliter les calculs on peut aussi utiliser les règles suivantes :

R2.- Modus tollens (l'inverse de modus ponens) ou, encore, résolutio, unitaire

$$\frac{\vdash A \to B, \vdash \neg B}{\vdash \neg A}$$

R3.- Élimination de « ET »

$$\frac{\vdash A \wedge B}{\vdash A, \vdash B}$$

R4.- Introduction de « ET »

$$\frac{\vdash A, \vdash B}{\vdash A \wedge B}$$

R5.- Introduction de « OU »

$$\frac{\vdash B}{\vdash A \vee B \vee C \cdots}$$

R6.- Loi de l'involution (Double négation)

$$\frac{\vdash \neg\neg A}{\vdash A}$$

R7.- Double résolution

$$\frac{\vdash A \vee \vdash B,\ \vdash \neg B \vee C}{\vdash A \vee C} \text{ ou, de façon équivalente } \frac{\vdash \neg A \to B,\ \vdash B \to C}{\vdash \neg A \to C}$$

R8.- Lois de de Morgan

$$\frac{\vdash \neg(A \vee B)}{\vdash (\neg A \wedge \neg B)} \qquad \frac{\vdash \neg A \wedge B)}{\vdash (\neg A \vee \neg B)}$$

R9.- Lois de simplification

$$\frac{\vdash A \vee (A \wedge B)}{\vdash A} \qquad \frac{\vdash A \wedge (A \vee B)}{\vdash A} \qquad \frac{\vdash A \vee (\neg A \wedge B)}{\vdash A \vee B}$$

Nous pouvons déterminer un *système d'inférence* ou *système de déduction* comme étant constitué d'un ensemble d'axiomes et d'un ensemble de règles d'inférence, c'est-à-dire par le couple $\mathcal{S} = (\mathcal{A}, \mathcal{R})$.

Si on se limite aux trois axiomes (A1)-(A3) et la règle (R1) du modus ponens, nous avons le système d'inférence de Hilbert, noté $\mathcal{H}$. Dans ce système, on peut établir le théorème :

---

[3] La notation $\frac{\vdash A}{\vdash B}$ est utilisée à la place de $(\vdash A) \vdash (\vdash B)$ pour indiquer qu'il s'agit d'une règle d'inférence et non pas

THÉORÈME 3.5.1 *On a*

$$\vdash A \rightarrow A$$

Dans ce système nous avons aussi la règle (controversée par les intuitionnistes) de *reductio ad absurdum* qui s'enonce comme suit :

$$Si\ A \vdash \neg B \rightarrow \bot,\ alors\ A \vdash B$$

Cette règle est à la base du théorème de réfutation (cf. infra) qui est la pierre angulaire du mécanisme de démonstration utilisé par Prolog.

Une *théorie* est constituée par la donnée d'un système d'inférence et de tous les théorèmes qui peuvent être obtenus par le système d'inférence.

Nous sommes maintenant en mesure de définir la démonstration et la déduction.

DÉFINITION 3.5.2 *Soit un théorème A. Une* démonstration *de A est une suite finie* $(A_1, A_2, \ldots, A_n, A)$ *où chaque* A_i *est soit un axiome, soit le résultat d'une règle d'inférence appliquée sur des éléments* A_j *précédemment obtenus (c'est-à-dire* $j < i$*).*

ASCÈSE 3.16 *Démontrer la fbf*

$$\vdash (A \rightarrow A)$$

DÉFINITION 3.5.3 *Une fbf A est une* déduction *de l'ensemble de fbf* $B_1, B_2, \ldots, B_n$*, que l'on note* $B_1, B_2, \ldots, B_n \vdash A$*, s'il existe une suite finie* $(A_1, A_2, \ldots, A_n, A)$ *où chaque* A_i *est soit un axiome, soit un des* B_i *soit il est obtenu par application d'une règle d'inférence sur des éléments* A_j *précédemment obtenus.*
Les fbf B_i *sont appelées des* hypothèses.

3.6 Équivalence entre modèles et théorie de démonstration

Nous allons examiner la relation qu'il existe entre l'interprétation sémantique d'une fbf et sa démonstration syntaxique. Nous avons vu que pour établir la validité d'une formule l'interprétation sémantique utilise des éléments qui ne font pas nécessairement partie du langage de la logique. Par contre la démonstration syntaxique utilise de méthodes de la logique pour élaborer la démonstration d'une formule.

Le problème que nous avons ici est que les résultats de l'interprétation sémantique ne doivent pas être en contradiction avec les résultats de la démonstration syntaxique. Il faut donc que, pour une formule donnée, $\vdash f$ entraîne $\models f$ (adéquation de la logique) et aussi que $\models f$ entraîne $\vdash f$ (complétude de la logique). Plus formellement, nous avons :

DÉFINITION 3.6.1 *Une logique est* adéquate *si tout théorème* $\vdash A$ *est une formule valide* $\models A$*.*

Une logique est syntaxiquement consistante *s'il n'existe aucune formule du langage telle que* $\vdash A$ *et* $\vdash \neg A$*.*

Une logique est (faiblement) complète *si toute formule valide est un théorème, c'est-à-dire* $(\models A) \rightarrow (\vdash A)$*.*

---

d'une proposition.

L'équivalence cherchée est obtenue à l'aide des trois théorèmes suivants :

THÉORÈME 3.6.1 *Le calcul propositionnel est adéquat :* $(\vdash A) \rightarrow (\models A)$.

THÉORÈME 3.6.2 *Le calcul propositionnel est syntaxiquement consistant.*

THÉORÈME 3.6.3 *Le calcul propositionnel est (faiblement) complet :* $(\models A) \rightarrow (\vdash A)$.

Les théorèmes 3.6.1 et 3.6.3 illustrent le fait qu'en calcul propositionnel, toute tautologie est un théorème et vice-versa, ce qui exprime le théorème suivant :

THÉORÈME 3.6.4 *Si* $\vdash p$, *alors* p *est une tautologie et vice-versa.*

3.7 Quelques méta-théorèmes

Dans les paragraphes précédents nous avons introduit un certain nombre de concepts concernant les fbf considérées individuellement. Nous allons étendre ces notions à un ensemble E des fbf. Nous avons ainsi :

DÉFINITION 3.7.1 *Un ensemble* E *de fbf est* insatisfiable *ou* sémantiquement inconsistant *si et seulement s'il n'existe aucune interprétation* $\mathcal{I}$ *telle que chaque fbf* A *de* E *soit satisfiable par* $\mathcal{I}$.

Cette définition permet d'établir le théorème de la réfutation :

THÉORÈME 3.7.1 (Th. de la réfutation) *Une fbf* A *est* conséquence valide *d'un ensemble* E *de fbf, c'est-à-dire* $E \models A$, *si et seulement si* $E \cup (\neg A)$ *est insatisfiable.*[4]

Si on note par $\perp$ la fbf qui est toujours fausse, on déduit qu'un ensemble de fbf est insatisfiable si et seulement s'il a comme conséquence la formule $\perp$. Il s'ensuit donc, que toute vérification de la validité d'un ensemble de fbf peut se réduire à la preuve de son inconsistance sémantique.

Cette remarque permet d'établir une autre méthode pour la vérification d'une fbf. Jusqu'ici la seule méthode que nous avons examiné était fondée sur les tables de vérité. La preuve par réfutation est une autre méthode qui a permis d'obtenir des algorithmes très éfficaces concernant la preuve des fbf.

[4]Une autre forme équivalente (et utile) de ce théoreme est la suivante : Si $F \vee A$ est inconsistante, alors $F \vdash \neg A$

Nous donnons ci-après quelques métathéorèmes[5] supplémentaires du calcul propositionnel en commençant par la partie sémantique. Nous avons :

THÉORÈME 3.7.2 (Th. de la déduction) $A_1, A_2, \ldots, A_n \models B$ *ssi* $A_1, A_2, \ldots, A_{n-1} \models (A_n \to B)$.

THÉORÈME 3.7.3 (Th. de finitude) *Soit F un ensemble fini ou dénombrable de fbf et A une fbf. Si $F \models A$, alors il existe un ensemble fini $F' \subset F$ tel que $F' \models A$.*

THÉORÈME 3.7.4 (Th. de monotonie) *Soient F_1, F_2 des ensembles finis de fbf. Si $F \models A$, alors $\{F_1, F_2\} \models A$.*

Du point de vue syntaxique nous avons :

THÉORÈME 3.7.5 (Th. de la déduction) $A_1, A_2, \ldots, A_n \vdash B$ *ssi* $A_1, A_2, \ldots, A_{n-1} \vdash (A_n \to B)$.

THÉORÈME 3.7.6 (Th. de transitivité) *Si $C \vdash A_1, \ldots, C \vdash A_n$ et si $(A_1, \ldots, A_n) \vdash B$, alors $C \vdash B$.*

THÉORÈME 3.7.7 (Th. d'échange) *Soit A une sous-formule de la fbf F. Si F est un théorème et si $A \leftrightarrow B$ est un théorème, alors la formule obtenue en remplaçant dans F une occurrence de A par B est un théorème.*

THÉORÈME 3.7.8 (Th. de substitution) *Soit S un théorème, x une proposition ayant au moins une occurrence dans S et A une fbf. Alors $S[x/A]$ est un théorème.*

THÉORÈME 3.7.9 (Th. de monotonie) *Soient F_1, F_2 deux fbf. Si $F \vdash A$,alors $F_1, F_2 \vdash A$.*

THÉORÈME 3.7.10 (Contraposition) $F \wedge A \vdash \neg B$ *ssi* $F \wedge B \vdash \neg A$

Nous donnons maintenant un méta-théorème qui part de la sémantique pour aboutir à la syntaxe.

THÉORÈME 3.7.11 (Règle T) *Si $F \models A_1, \cdots, F \models A_n$ et $A_1, \ldots, A_n \models A$, alors $F \vdash A$.*

ASCÈSE 3.17 *Inférer la proposition*

$$(p \to q, q \to r) \vdash p \to r$$

— *sans utiliser le th. de déduction;*
— *en utilisant le th. de déduction.*

Remplacer les hypothèses d'une conséquence sémantique par d'autres plus simples peut parfois être utile. Le théorème d'interpolation de Craig fournit une réponse à cette préoccupation.

THÉORÈME 3.7.12 (Th. d'interpolation) *Soient A et B deux fbf telles que $\models A \to B$. Alors il existe une fbf C composée uniquement par des propositions qui sont communes à A et à B et telle que $\models A \to C$ et $\models C \to B$.*

[5] Les métathéorèmes sont des théorèmes sur les théorèmes

Le théorème suivant dû à E.Beth, affirme qu'en logique propositionnelle une définition implicite peut toujours avoir une forme explicite. C'est une technique applicable à l'intelligence artificielle et qui permet d'obtenir des informations explicites à partir des informations exprimées implicitement.

THÉORÈME 3.7.13 (Th. de définissabilité) *Soit A une fbf ne contenant pas q et r. Si la fbf $(A(p,q) \wedge A(p,r)) \rightarrow (q \leftrightarrow r)$ est une tautologie, alors il existe une fbf B ne contenant pas p, q et r et telle que $A \rightarrow (p \leftrightarrow B)$ soit une tautologie.*

Nous allons finir cete section avec le théorème de compacité qui assure la possibilité, étant donné un ensemble infini d'hypothèses (i.e. des fbf) qui induit une fbf, d'en extraire un sous-ensemble fini qui induit la même fbf.

DÉFINITION 3.7.2 *Un ensemble infini de fbf est* finiment consistant *si tous ses sous-ensembles finis sont consistants.*

Un ensemble finiment consistant est maximal *s'il n'existe pas un sur-ensemble qui est finiment consistant.*

Concrètement si F est un ensemble infini de fbf et si pout tout $A \subset F$, avec A fini, il existe un modèle $M(A)$ de A, alors F est *finiment consistant*.

De plus étant donnée une proposition p, si on a $p \notin F$, alors $\neg p \in F$.

Nous avons les deux théorèmes suivants :

THÉORÈME 3.7.14 *Tout ensemble finiment consistant maximal est consistant et admet un modèle unique.*

THÉORÈME 3.7.15 (Th. de la compacité) *Tout ensemble finiment consistant est consistant.*

3.8 Arborescences sémantiques

Afin de représenter une fbf nous pouvons utiliser une arborescence([6]) qui est une structure particulière de la théorie des graphes.

Les définitions de la théorie des graphes qui seront utilisées ici sont les suivantes (cf. *C. Berge : Théorie des graphes et ses applications, Dunod, 1959*) :

Un *graphe* est un couple $G = (X, \Gamma)$, où X est un ensemble d'éléments, appelés *sommets* du graphe, et Γ est une application de X dans X. Une paire (x, y) d'éléments de X avec $y \in \Gamma(x)$ est appelée *arc* du graphe et sera noté par u et l'ensemble des arcs d'un graphe sera noté U. D'où une autre définition d'un graphe comme étant le couple $G = (X, U)$.

Une suite $c = [u_1, u_2, \ldots, u_n]$ d'arcs tels que l'extrémité terminale de chaque arc coïncide avec l'extrémité initiale de l'arc suivant, est appelé *chemin*. Si dans un chemin le sommet initial x_1 coïncide ave le sommet terminal x_n, alors nous avons un *circuit*.

Un graphe fini $G = (X, U)$ est une *arborescence de racine* $x_0 \in X$ si :

1^0 $\forall x \in X$, $x \neq x_0$ est l'extrémité terminale d'un seul arc;

[6] Notons que l'habitude en Intelligence Artificielle est d'appeler arbre l'arborescence, à cause d'une fâcheuse simplification de la terminologie de la théorie des graphes.

2^0 x_0 n'est l'extrémité terminale d'aucun arc;

3^0 G ne contient pas de circuits.

Les sommets qui ne sont pas l'extrémité initiale d'un arc quelconque s'appellent *sommets terminaux* ou *feuilles*.

On introduit d'abord la définition suivante :

DÉFINITION 3.8.1 *Étant donné un atome p, un* littéral *relatif à cet atome est soit p, soit $\neg p$.*

L'arborescence sémantique d'une fbf F composée des atomes $p_1, p_2, \ldots, p_n$ est construite de la façon suivante :

— Pour chaque atome $p \in F$, les deux éléments de $[p]$ sont deux sommets distincts de l'arborescence.

— Deux arcs dont l'extrémité terminale du premier est le sommet p et et du second le sommet $\neg p$, ont leur extrémité initiale commune.

— Aucun chemin ne comporte plus d'une occurrence de chaque atome.

L'arborescence sémantique est complète si chaque chemin contient une et une seule fois chaque atome. Elle est partielle si chaque chemin contient au plus une fois chaque atome.

Il est clair que l'arborescence sémantique complète de n atomes contient 2^n sommets terminaux. Les tables de vérité conduisent à l'examen de ces 2^n sommets terminaux et par conséquent leur utilisation est complètement inefficace dès que n dépasse la valeur de 5.

3.9 Formes clausales

Afin de construire des méthodes plus efficaces pour la preuve de la validité d'une fbf, nous allons utiliser la notion de la clause.

DÉFINITION 3.9.1 *Une* clause *C est une disjonction de littéraux $p_1 \vee p_2 \vee \cdots \vee p_n$.*
La clause vide sera notée par $\perp$. Il s'agit d'une fbf toujours fausse.

DÉFINITION 3.9.2 *Une conjonction de clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ est une* forme conjonctive normale (fcn).
Une fcn est parfois notée sous forme ensembliste $\{C_1, C_2, \ldots, C_n\}$.

On peut penser à transformer l'examen de la satisfiabilité d'une fbf en un examen de la validité d'une fcn, en espérant que ce dernier soit plus facile à réaliser. Néanmoins il faudrait qu'on puisse passer de la fbf à la fcn. Le théorème suivant introduit cette démarche.

THÉORÈME 3.9.1 (Théorème de normalisation) *Toute fbf peut se transformer à une fcn qui est logiquement équivalente.*

L'algorithme de la transformation d'une fbf en une fcn est le suivant :

(1) Élimination du connecteur $\leftrightarrow$: $(A \leftrightarrow B) \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

(2) Élimination du connecteur $\rightarrow$: $(A \rightarrow B) \equiv (\neg A \vee B)$

(3) Transfert de la négation au niveau le plus intérieur (i.e. devant les atomes) par utilisation des formules :
   — des lois de de Morgan $\neg(A \wedge B) \equiv \neg A \vee \neg B$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$ en tant que règles de réécriture;
   — de la loi de l'involution : $\neg\neg A \equiv A$ qui permet la suppression des doubles négations.

(4) Application de la distributivité
   — de $\vee$ par rapport à $\wedge$: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (B \vee C)$
   — de $\wedge$ par rapport à $\vee$: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (B \wedge C)$

   en tant que règles de réécriture.

La formule obtenue au terme de cet algorithme est une fcn, équivalente à la formule initiale. L'examen de la validité d'une fcn est fondé sur les trois remarques suivantes :

— Une fcn vide est consistante.
— Une fcn contenant la clause absurde $\bot$ est inconsistante.
— On réduit une fcn contenant deux clauses opposées en $\bot$, i.e. $A \wedge \neg A = \bot$.

ASCÈSE 3.18 *Appliquer l'algorithme ci-dessus à la fbf*

$$p \longleftrightarrow (q \rightarrow r)$$

Un type particulier de clause et – opératoirement – très utilisé est la clause de Horn.

DÉFINITION 3.9.3 *Une clause de Horn est une clause qui a une des trois formes suivantes :*

Clauses de Horn strictes (règles)	$p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$
Clauses de Horn négatives (questions)	$p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow$
Clauses de Horn positives (faits)	$\rightarrow q$

Donc une clause de Horn est une clause qui a au plus un atome positif. En effet, la clause de Horn ci-dessus peut aussi s'écrire $\neg(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \vee q$.

Les clauses de Horn strictes sont équivalentes à $\{p_1, p_2, \ldots, p_n\} \models q$. Les clauses de Horn négatives sont équivalentes à $\{p_1, p_2, \ldots, p_n\} \models \bot$. Les clauses de Horn positives sont équivalentes à $\models q$.

Notons que la clause vide $\rightarrow$ sans atome ni à gauche, ni à droite, est la seule clause inconsistante.

ASCÈSE 3.19 *Vérifier si les formules suivantes sont des clauses de Horn :*

(1) $(p \wedge q \wedge r \rightarrow p) \wedge (q \wedge s \rightarrow p) \wedge (p \wedge r \rightarrow r)$

(2) $(p \wedge q \wedge r \rightarrow \bot) \wedge (q \wedge s \rightarrow p) \wedge (\top \rightarrow r)$

(3) $(p \wedge q \wedge r \rightarrow \neg p) \wedge (q \wedge s \rightarrow p) \wedge (p \wedge r \rightarrow r)$

(4) $(p \wedge q \wedge r \rightarrow \bot) \wedge (\neg q \wedge s \rightarrow p) \wedge (\top \rightarrow r)$

Actuellement, un de problèmes majeurs de l'industrie du logiciel est la vérification, du point de vue des spécifications, du code écrit. Une voie de résolution est de transformer ce problème à un problème de satisfiabilité propositionnelle (PSAT) qui consiste à chercher un modèle pour une fbf donnée. Une procédure exhaustive pour résoudre le PSAT en utilisant des fcn est de tester systématiquement tous les éléments de la fcn. S'il y n atomes différents dans la formule, alors le problème s'appelle n-SAT et il faut faire 2^n tests. Il y a de cas spéciaux : 2-SAT avec complexité polynômial et 3-SAT avec complexité NP-complète.

L'importance des fcn vient aussi du fait que elle peut être utilisée pour démontrer des théorèmes en logique propositionnelle. En effet pour prouver A, on peut la réduire en sa fcn. Si cette forme prend la valeur vraie, alors A est aussi vraie car la transformation en fcn préserve l'équivalence logique. En effet, supposons que la fcn de A est $A_1 \wedge A_1 \wedge \cdots \wedge A_n$. Si A est valide, alors chaque A_i l'est aussi. Supposons que A_i est une disjonction des littéraux $L_1 \vee \cdots \vee L_{m_i}$. On cherche à savoir s'il y a une interprétation qui rend faux tous les littéraux L_j. Une telle interprétation existe sauf s'il y a deux littéraux L_j et L_k différents et tels que $L_j = \neg L_k$. Dans ce cas $L_1 \vee \cdots \vee L_{m_i} = \top$.

ASCÈSE 3.20 *Vérifier, en utilisant la reduction à la fcn si les fbf suivantes sont des théorèmes.*

(1) $(p \to q) \to ((q \to r) \to (p \to r))$

(2) $(p \vee q) \to (q \vee r)$

Cette méthode d'évaluation de la valeur de vérité d'une fbf est une méthode syntaxique. Elle est plus facilement mécanisable que les tables de vérité, mais elle souffre du même inconvénient que ces dernières : temps de calcul exponentiel.

On termine cette section avec la notion du résolvant d'un ensemble de clauses. Soit donc un ensemble C des clauses mis, éventuellement, sous forme conjonctive normale. Considérons un littéral p (c'est-à-dire un atome ou sa négation) et notons par C_p l'ensemble de clauses de C qui contiennent p et par $C_{\neg p}$ les clauses qui contiennent sa négation. Nous avons la

DÉFINITION 3.9.4 *Le résolvant de C par rapport à p est la clause $C_R(p) = C_p \cup C_{\neg p}$.*

Le résolvant permet de savoir si une clause sous forme disjonctive A est conséquence d'un ensemble de clauses C. En effet soit A est une clause de C, soit elle pourra être obtenue comme une clause d'un résolvant de C ou d'une application itérative du résolvant, en remplçant à chaque itération C par $C - \cup C_R(p) \cup \left(C_p^- \cup C_{\neg p}^-\right)$, où on a noté C_p^- l'ensemble de clauses de C_p privées du littéral p et $C_{\neg p}^-$ l'ensemble de clauses de $C_{\neg p}$ privées du littéral $\neg p$.

EXEMPLE 3.9.1 *Soit l'ensemble de clauses*

$$C = (p \vee q) \wedge (p \vee \neg q \vee r) \wedge \neg r$$

Nous allons montrer que $C \vdash p$.
On a $C_R(q) = p \vee (p \vee r)$ et donc $C = C_R(q) \wedge \neg r$. Ensuite on a $C_R(r) = p$ et donc $C = p \wedge p = p$.

Cette démarche s'appuie sur le théorème suivant :

THÉORÈME 3.9.2 (COHÉRENCE DU RÉSOLVANT) *Si C un ensemble de clauses et $C_R(p)$ son résolvant par rapport à p, on a*

$$C \models C_R(p)$$

.

On a aussi le théorème suivant qui fait la liaison entre syntaxe et sémantique pour le résolvant.

THÉORÈME 3.9.3 *Soit C un ensemble de clauses. Si $C \vdash C_R$, alors C_R est un résolvant de C, c-à-d; $C \models C_R$.*

ASCÈSE 3.21 *Prouver que $(a \wedge (a \rightarrow b) \wedge \neg b) \models \bot$.*

3.10 Algorithmes pour le calcul propositionnel

Le problème de la satisfiabilité d'une fbf A à partir des hypothèses $C_1, C_2, \ldots, C_n$ avec n fini, se décline de trois façons différentes :

— La fbf A est-elle une conséquence valide des fbf $(C_1, C_2, \ldots, C_n)$? Cette proposition s'écrit $C_1 \wedge C_2 \wedge \ldots \wedge C_n \models A$.

— La fbf $C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow A$ est-elle une formule valide ? Cette proposition s'écrit $\models (C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow F)$.

— $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg A$ est-elle une fbf insatisfiable ?

L'équivalence entre les trois propositions ci-dessus découle du théorème 3.7.1 de réfutation.

Cette pluralité d'approches du problème a donné naissance à une multitude d'algorithmes concernant la preuve de la validité d'une fbf.

Dans la suite nous présenterons les algorithmes suivants :

— Algorithme de Quine (arborescences sémantiques).

— Algorithme de réduction (tableaux sémantiques).

— Algorithme de Davis et Putnam (arborescences sémantiques).

— Algorithme de résolution (règle de la réfutation).

Tous les algorithmes qui suivent utilisent la fbf :

$$(\omega) \qquad C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow A$$

dont on cherche à prouver la validité.

3.10.1 Algorithme de Quine

Il est fondé sur les arborescences sémantiques mais il évite leur développement complet.

L'algorithme de Quine consiste en l'élaboration de l'arborescence sémantique étape par étape.

— À chaque sommet p ou $\neg p$ d'une branche de l'arborescence :

  - On procède à la substitution $\sigma = (p/1, \neg p/0)$.
  - On évalue la formule réduite $\omega' = \omega\sigma$.
  - Si ω' se réduit à une valeur de vérité (0 ou 1), arrêt de l'exploration de la branche. Sinon on poursuit l'exploration de la branche.

— Le processus s'arrête :

  - avec succés, lorsque on a obtenu un résultat sur une branche avec valeur de vérité à 1 et sur une autre branche avec valeur de vérité à 0
  - avec échec, lorsque on a parcouru toutes les branches.

ASCÈSE 3.22 *Appliquer l'algorithme de Quine à la fbf*

$$((p \to q) \wedge p) \to q$$

3.10.2 Algorithme de réduction

Cet algorithme utilise la technique de la preuve par l'absurde. Il consiste à supposer que la fbf à démontrer soit fausse, c'est-à-dire que la fbf $(C_1, C_2, \ldots, C_n, \neg A)$ est insatisfiable, et à arriver, en appliquant l'algorithme, à une contradiction.

— On construit une table avec deux colonnes : 1 et 0.

— On met la fbf (ω) dans la colonne 0 (i.e. on suppose que (ω) est fausse).

— On décompose (ω) en sous-formules qu'on place soit à la colonne 1, soit à la colonne 0, selon les règles suivantes :

  - Si on a $p \to q$ dans la colonne 0, alors on place p dans la colonne 1 et q dans la colonne 0.
  - Si on a $p \wedge q$ dans la colonne 1, alors on place p et q dans la colonne 1.
  - Si on a $p \vee q$ dans la colonne 0, alors on place p et q dans la colonne 0.

— Si on arrive à avoir dans les deux colonnes, 0 et 1, la même formule, alors on est en présence d'une contradiction (la même formule est à la fois vraie et fausse !) et on peut s'arrêter car la fbf (ω) est vraie.

ASCÈSE 3.23 *Appliquer l'algorithme de la réduction à la fbf*

$$((p \wedge q) \to r) \to (p \to (q \to r))$$

3.10.3 Algorithme de Davis - Putnam

Soit C une fcn et p un atome. Alors C en tant qu'ensemble est partionné en trois sous-ensembles :

— C_p ensemble des clauses contenant p ;

— $C_{\neg p}$ ensemble des clauses contenant $\neg p$;

— $C_R = C - (C_p \cup C_{\neg p})$

D'autre part on note par

— C_p^- l'ensemble des clauses de C_p privées de p.

— $C^-_{\neg p}$ l'ensemble des clauses de $C_{\neg p}$ privées de $\neg p$.

L'algorithme est fondé sur les deux propriétés suivantes :

PROPRIÉTÉ 3.10.1 *Si p est vraie, alors l'ensemble C équivaut à $C^-_{\neg p} \cup C_R$.*

PROPRIÉTÉ 3.10.2 *Si p est fausse, alors l'ensemble C équivaut à $C^-_p \cup C_R$.*

L'algorithme est le suivant :

(1) Si $C = \perp$, alors C est satisfiable.
(2) Si $C = \{\perp\}$, alors C est insatisfiable.
(3) Sinon
   (a) Choisir une proposition $p \in C$
   (b) Calculer $C_p, C_{\neg p}$ et C_R.
   (c) Calculer C^-_p et $C^-_{\neg p}$.
   (d) Si les deux ensembles $C^-_{\neg p} \cup C_R$ et $C^-_p \cup C_R$ sont insatisfiables, alors C est insatisfiable.

Nous pouvons nous apercevoir que cet algorithme à chaque étape :

— supprime de la fcn sous examen un atome;
— décompose la fcn sous examen en deux fcn plus simples dans la mesure où il n'y figure plus l'atome supprimé.

ASCÈSE 3.24 *Examiner si l'implication suivante*

$$\{p \to q,\ q \to r\} \models (p \vee q) \to r$$

est valide.

3.10.4 Algorithme de résolution

Il s'agit d'un algorithme proposé par A. Robinson et fondé sur le théorème 3.7.1 de réfutation et sur le résolvant.

Considérons une fnc $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, où chaque C_i clause. Dans la suite on notera une fnc sous forme ensembliste $C = \{C_1, C_2, \ldots, C_n\}$, où la virgule ici est mise à la place du connecteur $\wedge$. Si cette fcn est insatisfiable, alors il est toujours possible d'obtenir, à partir de C, une contradiction qui, sous forme clausale, équivaut à la clause vide. Ainsi si on veut automatiser la procédure de la détermination de l'insatisfiabilité, on peut envisager une méthode qui produirait des conséquences à partir de la fcn sous test et qui s'arrêterait dès que la clause vide serait engendrée.

Il est évident que cette technique peut aussi s'appliquer quand on veut établir qu'une formule est une conséquence logique d'un ensemble de formules, i.e. si on veut établir que $C \models A$. En utilisant le théorème de réfutation on sait que $C \models A$ ssi $C \cup \{\neg A\}$ est insatisfiable.

L'algorithme de résolution par réfutation est le suivant :

(1) Considérons une fnc $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ où les C_i sont des clauses disjonctives et soit la fbf $C \rightarrow A$ où A est une clause disjonctive.
Exemple : Soit la fbf $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow r$ avec $C = p \wedge (p \rightarrow q) \wedge (q \rightarrow r)$ et $A = r$. On transforme C en fnc et on obtient $C = p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$.

(2) On écrit C sous forme ensembliste $C = \{C_1, \ldots C_n\}$.
Exemple : $C = \{p, \neg p \vee q, \neg q \vee r\}$

(3) On rajoute à cet ensemble la négation de la conclusion A et on obtient l'ensemble $S = \{C_1, \ldots C_n, \neg A\}$
Exemple : $S = \{p, \neg p \vee q, \neg q \vee r, \neg r\}$

(4) On cherche dans S un littéral p tel que S_P (sous-ensemble d'éléments de S contenant p) et $S_{\neg p}$ (sous-ensemble d'éléments de S contenant $\neg p$) soient non vides, c-à-d. $S_P \neq \emptyset, S_{\neg p} \neq \emptyset$. S'il n'y en a pas, A n'est pas une déduction logique de C et l'algorithme s'arrête.
Sinon on forme le résolvant de S par rapport à p $R_S(p) = S_p \cup S_{\neg p}$ et on remplace S par $S = S - \cup C_R(p) \cup \left(C_p^- \cup C_{\neg p}^-\right)$
Exemple : $S_p = p$ et $S_{\neg p} = \neg p \vee q$. Le résolvant est $R_S(p) = \{q\}$.

(5) Si S se réduit à la clause vide, alors la déduction logique est valide.
Sinon, on retourne à (4).
Exemple : $S = \{q, \neg q \vee r, \neg r\}$. On continue $S = \{q, \neg q \vee r, \neg r\}, S_q = q, R_{\neg a} = \neg q$. Donc $R_S(q) = \{r\}$ et $S = \{r, \neg r\}$ d'où $C = \{\perp\}$ où $\perp$ représente la clause vide dont la valeur de vérité est Faux([7]). Par conséquent la fbf $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow r$ est valide.

ASCÈSE 3.25 *Examiner si l'implication suivante*

$$\{p \rightarrow q, q \rightarrow r\} \models (p \vee q) \rightarrow r$$

est valide.

Nous pouvons aussi utiliser la règle de résolution pour examiner si une fbf A est valide. Pour ce faire on nie la fbf A et on cherche à montrer que cette négation conduit à une absurdité.

EXEMPLE 3.10.1 *Soit la fbf $A = (p \rightarrow \neg p) \rightarrow \neg p$. On nie la proposition et on a $\neg A = \neg [(p \rightarrow \neg p) \rightarrow \neg p]$. On traduit cette formule en fnc $\neg [(p \rightarrow \neg p) \rightarrow \neg p] \leftrightarrow \neg [\neg (\neg p \vee \neg p) \vee \neg p]$. Donc $C = \{\neg p \vee \neg p, p\} \leftrightarrow C \ \{\neg p, p\} \leftrightarrow C = \{\square\}$ et donc la formule est valide dans la mesure où sa négation est absurde.*

Nous pouvons encore utiliser la règle de résolution pour calculer la conséquence d'une fbf. Dans ce cas il faut appliquer l'algorithme sans nier la fbf et récupérer la conclusion.

EXEMPLE 3.10.2 *Reprenons l'exemple précédent en considérant uniquement les prémisses : $A = (p \rightarrow \neg p)$. On applique l'algorithme à A, ce qui donne $C = \{\neg p \vee \neg p\} \Rightarrow C = \{\neg p\}$ c-à-d. que cette formule est équivalente à $\neg p$.*

[7] N.B. La valeur de vérité d'une clause vide $\perp$ est Faux. Par contre la valeur de vérité d'un ensemble de clauses vide

3.11 Démonstration automatique

On termine ce chapitre en présentant un algorithme qui permet de prouver une fbf du calcul propositionnel. Il s'agit de la méthode de balayage qui accepte en entrée une fbf qui n'a pas des connecteurs de l'implication ou de la double implication et retourne soit $\top$ si la formule est prouvée, soit $\bot$ sinon.

L'algorithme est le suivant :

Debut

(1) Soit C une fbf sans les connecteurs de l'implication ou de la double implication Si ces connecteurs sont présents, on les élimine en appliquant les règles :
   — $p \rightarrow q \leftrightarrow \neg p \vee q$
   — $p \leftrightarrow q \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p$

(2) Tant que $C \neq \top$ ou $C \neq \bot$ faire

(3) Début
   (a) Choisir un littéral p apparaîssant dans C.
   (b) Remplacer C par la fbf $C(p/\top) \wedge C(p/\bot)$, où $C(p/a)$ est la fbf C dans laquelle on a remplacé chaque occurrence de p par a.
   (c) On simplifie la formule, en appliquant autant que faire se peut les équivalences suivantes dans n'importe quel ordre :
      — $\neg \bot \leftrightarrow \top$
      — $\neg \top \leftrightarrow \bot$
      — $p \vee \top \leftrightarrow \top$
      — $p \vee \top \leftrightarrow p$
      — $p \wedge \top \leftrightarrow p$
      — $p \wedge \bot \leftrightarrow \bot$

(4) Fin Tant que

Fin

ASCÈSE 3.26 *Démontrer la fbf*

$$(p \rightarrow q) \rightarrow ((p \rightarrow q) \rightarrow \neg p))$$

en faisant les deux choix différents pour l'atome p, à savoir p et q.

Il faut remarquer que le choix du littéral à substituer influe sur la longueur de la solution. En règle générale, on préférera, pour diminuer la longueur de la solution, de commencer par le littéral qui est le plus fréquent dans la fbf.

3.12 Exercices

EXERCICE 3.1 *On se place dans l'univers des tribunaux de justice. Considérons la phrase*

Si Toto est innocent d'un crime, alors il n'est pas suspect.

---

est Vrai. Car dans ce dernier cas, l'absence de clause signifie que cet ensemble est toujours vérifié.

p	q	p ↓ q
0	0	1
0	1	0
1	0	0
1	1	0

(1) Écrire la proposition contrapositive et calculer sa valeur de vérité.

(2) Calculer la valeur de vérité de la phrase donnée.

(3) Écrire la proposition inverse et calculer sa valeur de vérité.

(4) Écrire la négation de cette phrase et calculer sa valeur de vérité.

EXERCICE 3.2 *Considérons l'opérateur logique* `nand` *dont la table de vérité est la suivante :*

p	q	p `nand` q
1	1	0
1	0	1
0	1	1
0	0	1

(1) Trouver une proposition équivalente à $\neg p$ en utilisant uniquement le connecteur `nand`*.*

(2) Trouver une proposition équivalente à $p \vee q$ en utilisant uniquement le connecteur `nand`*.*

(3) Trouver une proposition équivalente à $p \wedge q$ en utilisant uniquement le connecteur `nand`*.*

EXERCICE 3.3 *Considérons le connecteur $\downarrow$ dont la table de vérité est la suivante :*
Refaire le travail de Wittgenstein, qui définissait tous les autres connecteurs à partir de ce connecteur.

EXERCICE 3.4 *Soit la fbf*

$$A = (Q \wedge R) \rightarrow (P \leftrightarrow (\neg Q \vee R))$$

dans laquelle P, Q, R sont des variables propositionnelles.

(1) Déterminer une formule équivalente à A, écrite en utilisant uniquement les connecteurs $\rightarrow$ et $\leftrightarrow$.

(2) Donner pour A la forme normale disjonctive la plus réduite.

(3) Montrer que les formules

$$B = R \rightarrow (Q \rightarrow (P \leftrightarrow (Q \rightarrow R))) \text{ et } C = R \rightarrow (Q \rightarrow P)$$

sont équivalentes.

EXERCICE 3.5 *On additionne deux nombres binaires à deux digits ab et cd. Le résultat est un nombre à trois digits au plus : pqr. Utiliser le calcul propositionnel pour obtenir une expression de p, q, r en fonction de a, b, c, d.*

EXERCICE 3.6 *Donner la fcn de deux fbf ci-après :*

(1) $(\neg p \wedge q) \rightarrow (p \wedge (r \rightarrow q))$

(2) $(p \wedge \neg q) \vee (p \wedge q) \vee (\neg p \wedge \neg q)$

EXERCICE 3.7 *Protagoras était convenu avec son élève Euathlos qu'il paierait ses leçons que lorqu'il gagnerait son premier procès. L'élève refusant de plaider, Protagoras le poursuivit en justice en faisant le raisonnement suivant :*

Il gagnera ou perdra ce procès.
S'il gagne, il me paiera afin de respecter notre accord.
S'il perd, il me paiera afin de respecter le verdict des juges.
Donc, il me paiera.
Formaliser le raisonnement de Protagoras et donnez-en une démonstration.

EXERCICE 3.8 *Pour effectuer le test de bon fonctionnement d'un procédé de fabrication on vérifie la présence des quatre symptômes : S_1, S_2, S_3 et S_4. Lors des différents tests, nous avons observé que :*

- *Si S_1 et S_2 sont présents, alors un de S_3, S_4 est aussi présent.*
- *Si S_2 et S_3 sont présents, alors soit aucun de S_1, S_4 n'est présent, soit tous les deux sont présents.*
- *Si aucun de S_1 et S_2 n'est présent, alors aucun de S_3, S_4 n'est non plus présent.*
- *Si aucun de S_3 et S_4 n'est présent, alors aucun de S_1, S_2 n'est non plus présent.*

Faire le travail suivant :

(1) Traduire les observations en langage des propositions.

(2) Montrer que les deux propositions

(a) Les trois symptômes S_1, S_2 et S_3 ne peuvent pas être présents en même temps.
(b) Si les symptômes S_1, S_2 ne sont pas présents, alors S_3 n'est pas non plus présent.

sont valides, en utilisant

- *les tables de vérité;*
- *l'algorithme de Davis-Putnam;*
- *l'algorithme de résolution.*

EXERCICE 3.9 *Examiner si la proposition*

$$[(p \vee q) \wedge (\neg r \rightarrow \neg q)] \rightarrow (p \vee r \vee \neg q)$$

est valide.

EXERCICE 3.10 *Soit le raisonnement d'Aristote*

- *S'il faut philosopher, alors il faut philosopher.*
- *S'il ne faut pas philosopher, alors il faut encore philosopher.*
- *Conclusion : il faut philosopher.*

Traduisez ce raisonnement et prouvez-le.

EXERCICE 3.11 *Bias, un de sages grecs de l'antiquité, tenait le raisonnement suivant contre le mariage :*

(1) Si vous vous marier, votre femme sera belle ou laide.

(2) Si elle belle, vous serez en proie à la jalousie.

(3) Si elle est laide, vous ne la supporterez pas.

Donc, il ne faut pas vous marier.

Montrer que le raisonnement n'est pas valide et trouvez les propositions à rajouter aux prémisses pour qu'il devienne valide.

EXERCICE 3.12 *Démontrer, en utilisant l'algorithme de balayage, la formule suivante :*

$$(p \to (q \vee r)) \to ((p \to q) \vee (p \to r))$$

4

PROLOGUE AU PROLOG

`Prolog` est un langage de programmation issu de la logique computationnelle. Son nom vient de « PROgrammation LOgique », son fondateur est Alain Colmerauer de l'université de Marseille et son année de naissance 1973. Comme langage de programmation, `Prolog` utilise un formalisme différent des autres langages pour l'écriture des programmes et la définition des spécifications, car il est de nature différente.

En effet on considère, en général, que le travail de l'informaticien est de construire un programme qui représente l'organisation et le fonctionnement d'un système complexe d'informations. Pour ce faire il dispose de plusieurs langages de programmation qu'on peut classer en trois catégories :

(i) LANGAGES PROCÉDURAUX OU IMPÉRATIFS : Le programme écrit dans un tel langage doit indiquer à l'ordinateur les actions à faire pour obtenir le résultat voulu. Il s'agit donc des langages de bas niveau, la programmation se limitant à passer des ordres à l'ordinateur afin que ce dernier puisse effectuer la suite des actions désirées. Ainsi il y a, de la part de l'informaticien, un travail considérable à fournir pour traduire les spécifications d'un système complexe d'informations en un programme qui marche. Des langages de telle nature sont tous les langages de 3e génération (`Fortran`, `Pascal`, `C`, `Ada`, etc).

(ii) LANGAGES FONCTIONNELS : Le programme écrit dans un tel langage comporte une suite d'appels à des fonctions. Comme on le sait, le retour d'une fonction appelée fournit au programme qui fait cet appel, une valeur. La suite de ces valeurs doit nous permettre d'obtenir le réultat voulu. Ces langages sont des langages de niveau intermédiaire car la programmation est pensée en termes des valeurs calculées et non pas, comme précédement, en termes de comportement. Il y a donc pour l'informaticien moins de travail à effectuer,

car les valeurs font partie des spécifications d'un système complexe d'informations. `Lisp`, `Simula`, `ML` et `Caml` sont des langages de cette nature.

(iii) LANGAGES RELATIONNELS OU DÉCLARATIFS : Le programme écrit dans un tel langage comporte des définitions des relations entre différentes entités du système d'information. Nous pouvons donc le considérer comme une déclaration de l'existence de ces relations. Le résultat voulu est obtenu par la mise en fonctionnement de ces relations. Il s'agit d'un langage de haut niveau, dans la mesure où les spécifications d'un système complexe d'informations sont faites selon une approche relationnelle. Le travail donc, que doit fournir l'informaticien est minime. Par conséquent ces programmes sont faciles à construire, faciles à comprendre, faciles à tester, faciles à maintenir et faciles à adapter pour satisfaire à d'autres buts. `Prolog` est un langage de cette catégorie.

Que l'industrie du logiciel soit quasi monopolisée par l'utilisation des langages procéduraux est la conséquence du fait que les ordinateurs ont une architecture machine de type von Neumann.(Les ordinateurs à architecture von Neumann ont un procédé de traitement de l'information fondé sur le cycle « fetch-execute », à savoir une boucle qui passe successivement par les trois états suivants :

— activer l'instruction prochaine;
— décoder l'instruction;
— exécuter l'instruction.

En effet un langage de programmation est un intermédiaire – on aurait pu dire une interface – entre l'homme et l'ordinateur. Il permet à l'homme d'expliquer à la machine ce qu'il souhaiterait qu'elle fasse. Donc avec un langage de programmation on doit pouvoir faire l'une de deux choses :

— soit établir la structure de l'ordinateur. Ce que nous faisons avec tous les langages procéduraux. Dans ce cas le programmeur doit obtenir une correspondance entre le modèle de l'ordinateur et le modèle du problème. Ce travail n'est pas intrinsèque au langage de programmation. Il s'agit d'une tâche qui, pour le même problème, doit se faire quasiment de la même façon pour tout langage procédural, à l'extérieur du travail de la programmation – et même avant celui-ci. Ainsi avec un langage procédural il est difficile d'écrire un programme et encore plus difficile de le maintenir, ce qui explique assez bien la raison pour laquelle on a créé ce qu'on appelle *industrie du logiciel*.
— soit établir la structure du problème. Pour ce faire nous devons avoir un modèle du monde ou, de façon plus modeste, de la partie de l'univers (du micromonde, comme on disait jadis) dans lequel ce situe notre problème. C'est l'approche utilisée les langages de deux autres catégories. Pour le `Lisp`, par exemple, toute action peut s'exprimer à l'aide d'une fonction, toute donnée peut être codée à l'aide d'une liste. Donc pour le `Lisp` tous les problèmes sont réductibles à des fonctions et des listes. La manière dont les fonctions et les listes sont prises en compte par l'ordinateur n'est pas une préoccupation pour le programmeur. De même pour `Prolog` tout problème peut se ramener à un calcul de la valeur de vérité d'une suite des prédicats.

De ce qui précède découle qu'un avantage des langages déclaratifs est le fait qu'ils obligent le programmeur d'établir une démarche algorithmique indépendante des caractéristiques technologiques et de l'architecture matérielle de l'ordinateur sur lequel le programme s'exécutera. `Prolog` en particulier utilise comme élément de base de la programmation la notion du prédicat. Sa définition inclut les arguments d'entrée et de sortie et laisse donc au programmeur seulement le soin de déterminer le résultat souhaité – la sortie – et les paramètres – les entrées – qui permettent d'obtenir ce résultat, sans qu'il soit préoccupé d'écrire des instructions détaillées concernant la démarche pour arriver au résultat.

Les langages de programmation fondés sur la logique computationnelle ont en commun :

— l'utilisation des faits du domaine de connaissance comme des donné
— l'utilisation des règles pour exprimer les relations entre les faits;
— l'utilisation de la déductions pour répondre à des questions concernant le domaine de connaissance particulier.

Leurs différences par rapport aux langages procéduraux ce concentrent essentiellement à l'absence de contrôle du flux des instructions à exécuter. Ainsi il n'y a pas dans la programmation logique les différents blocs conditionnels qu'on trouve dans les autres langages. Par exemple il n'existe pas le bloc `if - then - else`, ni des boucles `while` ou `repeat`. De plus il n'y a pas des variables globales ni des états qui se modifient globalement.

`Prolog` permet d'utiliser la logique pour traiter des informations avec un ordinateur, c'est-à-dire d'utiliser la logique comme un langage de programmation. L'idée qui était dominante à l'époque de la première apparition de `Prolog` peut se résumer à la fameuse formule de N. Wirth (« inventeur » des langages de programmation à forte dominante algorithmique, `Algol` et `Pascal`.)

Algorithmes + Structure de données = Programmes

qui fournissent aux programmes une comportement dynamique en ce sens que la conséquence d'un programme est le résultat de.l'intervention d'un algorithme dans un ensemble de données doté d'une structure. En même temps cette formule établit la séparation entre algorithmes et structure de données.

P. Hayes, à la même époque, pensait que la computation était de la déduction contrôlée, tandis que E. F. Codd envisageait les systèmes de bases de données comme étants composés de deux éléments : un premier élément relationnel qui déterminait la structure logique des données et un second élément de contrôle qui permettait le stockage et le traitement des données. R. Kowalski en partant de ces idées, a établi (in Communications ACM, vol.22, no 7, july 1979, pp.424-436) une formule analogue à celle de Wirth et qui constitue aujourd'hui la thèse principale de la programmation logique, à savoir

Algorithme = Logique + Contrôle

Cette formule établit la séparation entre la logique, qui contient les spécifications des données, et le contrôle, qui établit l'ordre dans lequel les opérations logiques doivent s'effectuer. Si donc on enlève des programmeurs la tâche de spécifier la composante « contrôle » de la formule, les programmes logiques acquièrent une structure statique car il contiennent la connaissance la plus

appropriée à utiliser mais ils n'explicitent pas les méthodes qui permettraient de l'explorer. Par exemple `Prolog` exécute la partie « Contrôle » de l'algorithme automatiquement. Il ne laisse donc à la charge du programmeur que la partie « Logique ».

L'utilisation industrielle de `Prolog` est en dents de scie. Il a été le langage choisi par les japonais dans leur tentative de construire l'ordinateur de la 5e génération. Le projet a échoué – pour des raisons financières et politiques – mais `Prolog` n'est pour rien. Récemment il a été utilisé pour la programmation de l'énorme base de données du projet du génome humain avec beaucoup de succès. `Prolog` souffre essentiellement du fait qu'il n'est pas connu. Il y avait un espoir de rendre `Prolog` moins « ovni-esque » avec l'introduction dans le primaire à la fin des années 80 du langage `Logo`, qui à bien des égards ressemble beaucoup au `Prolog`, mais quelques années après, des esprits supérieurs du ministère de l'Éducation Nationale ont troqué le `Logo` contre le `Basic` ! Ainsi l'abrutissement des élèves du primaire était garanti et la méconnaissance de Prolog assurée.

Le cours de `Prolog` se place en même temps que le cours de la Logique Computationnelle qui permet aux élèves d'étudier les fondements théoriques de la programmation logique. L'objectif du cours est de permettre aux élèves d'apprendre à écrire des programmes en `Prolog`, ce qui, d'une part, leur permettra de maîtriser un langage très puissant de mise en œuvre des maquettes des programmes et, d'autre part, leur sera utile et nécessaire pour aborder les deux cours de 2e année et qui font partie du cycle des cours d'IA, à savoir Intelligence Artificielle et Systèmes Experts.

RÉFÉRENCES

Pour compléter l'enseignement, les élèves pourraient utiliser soit le livre de

W. F. CLOCKSIN - C. S. MELLISH : *Programmer en Prolog*, Éditions Eyrolles, traduction en français du livre *Programming in Prolog* aux éditions Springer-Verlag,
soit le livre de
J. ELBAZ : *Programmer en Prolog*, Éditions Ellipses, 1991
soit encore le livre de
I. BRATKO : *Prolog, Programming for artificial intelligence*, Addison-Wesley, 1986
dont il existe aussi une traduction française.

Les élèves qui souhaiteraient, en plus, avoir un livre qui regroupe les fondements de la programmation logique et les techniques de base du langage de programmation Prolog, peuvent utiliser le livre de

U. NILSSON - J. MAŁUSZYNSKI : *Logic, Programming and Prolog*, Wiley, 1990.

Dans le même esprit mais nettement plus orienté Prolog, il y a le livre

M. SPIVEY : An introduction to logic programming through Prolog, Prentice-Hall, 1995.

Les élèves qui voudraient, après la fin de ce cours, approfondir leurs connaissances en Prolog, peuvent consulter le livre de

L. STERLING - E. SHAPIRO : *L'art de Prolog*, Éditions InterÉditions,

et dont tous les exemples de programmes sont sur Internet, en libre accés à l'adresse du MIT : `ftp://mitpress.mit.edu`.

On peut aussi citer trois autres livres introductifs qui contiennent quelques applications interéssantes :
J. MALPAS : *Prolog : a relational language and its applications*, Prentice-Hall, 1987
C. MARCUS : *Prolog programming*, Addison-Wesley, 1986
J. STOBO : *Problem solving with Prolog*, Pitman, 1989

ainsi qu'un livre plus orienté ingénierie logicielle
T. AMBLE : *Logic programming and knowledge engineering*, Addison-Wesley, 1987.
Signalons encore un livre qui pourrait être utile au ptogrammeur :
W. F. CLOCKSIN : *Clause and Effect : Prolog programming for the working programmer*, Springer, 199 7.
Enfin le manuel de référence du langage peut se trouver dans les livres :
P. DERANSART, A. ED-DBALI, L. CERVONI : *Prolog : The Standard*, Springer-Verlag, 1986.
R. O'KEEFE : *The craft of Prolog*, MIT Press, 1990.

5

LOGIQUE DES PROPOSITIONS ET PROGRAMMATION

Comme nous avons vu au chapitre précédent, le langage de programmation Prolog est fondé sur la logique mathématique afin de stocker et traiter des informations sous forme symbolique. Ces informations symboliques sont issues de la modélisation des connaissances que possède un être intelligent. Afin d'effectuer cette modélisation, la connaissance est décomposée en différentes parties qui sont considérées comme étant autonomes et indépendantes, bien que c'est rarement le cas. Chacune de ces parties qui, en règle générale, est relative à un environnement donné, constitue ce qu'on appelle un *univers du discours* et chaque fois, pour résoudre un problème, on travaille à l'intérieur d'un univers du discours spécifique que l'on notera $\mathcal{U}$.

Une manière répandue et commode pour modéliser les connaissances d'un univers du discours est d'utiliser le triplet *objet – attribut – valeur*. Par *objet* on entend les objets ou entités physiques ou conceptuelles de l'univers du discours. On utilise un *attribut* pour décrire les caractéristiques et/ou les propriétés des objets et aussi leurs relations. La *valeur*, enfin, est obtenue par la spécification d'un attribut pour un objet particulier.

La programmation en Prolog se fait à l'intérieur d'un univers du discours. Comme Prolog est un langage relationnel, ses objets sont essentiellement des relations entre les objets de l'univers du discours. Ainsi la programmation en Prolog consiste

— en la spécification des *faits* concernant les objets,

— en la construction des *règles* concernant les relations entre objets,

— en la réponse à des *questions* posées concernant l'existence des objets et/ou les relations entre objets.

Les faits, règles et questions sont des *expressions logiques*. D'autre part une *constante* – qui est le nom d'un objet ou d'une relation entre objets – est un *terme logique*. Le nom d'une constante en `Prolog` doit commencer toujours par une lettre minuscule. Un terme logique est aussi une *variable* qui peut représenter n'importe quel objet. Le nom d'une variable en `Prolog` doit toujours commencer par une lettre majuscule.

Nous donnons ci-après un exemple de fichier source de `Prolog` :

```
1  ilPleut.                    // Un fait
2  porteParapluie :- ilPleut. // Une r\'egle
3
4  humain(socrate).            // Un fait
5  humain(aristote).           // Un autre fait
6  mortel(X) :- humain(X).     // Une r\'egle
7  existeHumain :- humain(X). // Une autre r\'egle
8
9  animal(X) :- lion(X).       // Un fait
10
11 ?- mortel(socrate).         // Une question
12 ?- humain(X).               // Une autre question
```

Exemple des questions avec les réponses de `Prolog` :

```
1  ?- ilPleut.
2  yes
3
4  ?- porteParapluie.
5  yes
6
7  ?- animal(socrate).
8  no
9
10 ?- mortel(X).
11 X = socrate;
12 X = aristote;
13 no
14
15 ?- existeHumain.
16 yes
```

5.1 Syntaxe de `Prolog`

Dans cette section nous passons en revue les éléments de base du langage, ainsi que leur syntaxe.

5.1.1 Les faits

Les faits en `Prolog`

— soit affirment l'existence d'un objet particulier avec des caractéristiques particulières. Exemple : toto est un cube de couleur bleue, ce qui en Prolog donne : `cube(toto, bleue).`, ce qui a comme conséquence que le fait en question a la valeur de vérité 1 - vrai,

— soit fournissent la valeur – numérique ou symbolique – de la caractéristique d'un fait. Exemple : le volume du cube toto est 100 ce qui en Prolog donne : `volumeCube(toto, 100).` .

La forme générale d'un fait est la suivante:

$$\texttt{nomFait(objet}_1\texttt{, objet}_2\texttt{, ... , objet}_n\texttt{).}$$

Le point « . » avec lequel termine le fait, indique la fin du fait.

Les différents objets qui interviennent à la déclaration d'un fait ce sont les arguments du fait. Remarquons que deux faits de même nom peuvent ne pas faire référence à la même relation. Cela dépend du nombre d'arguments. Si le nombre d'arguments est le même, les deux faits font référence à la même relation avec, éventuellement, des arguments différents. Si par contre le nombre d'arguments n'est pas le même, alors les deux faits se rapportent à deux relations différentes.

Nous pouvons composer les faits entre eux en utilisant les connecteurs logiques « $\wedge$ - *et* » et « $\vee$ - *ou* ». La conjonction « *et* » est notée en Prolog par la virgule « *,* » et la disjonction « *ou* » par le point-virgule « *;* ».

Avec beaucoup de précautions, nous pouvons considérer que les faits en Prolog jouent le même rôle que les données pour un langage de 3e génération. Cette analogie permet aussi de comprendre qu'un fait, dans la mesure ou il est une donnée, il ne peut pas être inconnu et, donc, il ne peut pas être représenté par une variable, c'est-à-dire avoir un nom qui commence avec une lettre majuscule. Ainsi `cube(toto, bleue, 10).` est un fait legal en Prolog, tandis que `X(toto, bleue, 10).` qui exprime une relation inconnue pour le cube `toto` est incomprehensible pour Prolog.

De ce qui vient d'être dit, on conçoit aisément qu'en Logique Computationnelle `nomFait` était appelé soit *prédicat*, soit *foncteur*. En Prolog, `nomFait` est toujours un prédicat qui est considéré comme le nom d'une base de données et si les valeurs des arguments se trouvent dans cette base de données, alors la valeur du prédicat `nomFait` est égale à vraie.

5.1.2 Les règles

Une règle en Prolog exprime la manière dont un fait est relié ou découle d'autres faits, c'est-à-dire une règle est ce que, en Logique Computationnelle, on a appelé axiome ou théorème logique.

La forme générale d'une règle est la suivante :

$$\texttt{A:- B}_1\texttt{, B}_2\texttt{, ... , B}_n$$

où `A` et $\texttt{B}_k$ sont des faits. `A` est *l'entête* de la règle et il doit être un atome positif (c'est-à-dire n'ayant aps de négations) et $\texttt{B}_1\texttt{, B}_2\texttt{, ... , B}_n$ constituent *le corps* de la règle et ce sont des littéraux.

Cette forme de la règle est la forme qu'en logique computationnelle nous l'avons vu sous le nom de la *clause de Horn*. Nous remarquons qu'un fait est aussi une clause de Horn dont le corps est vide.

Les faits et les règles d'un univers du discours, constituent *la base de données* de cet univers du discours ou, encore, la base de connaissances de l'environnement.

Remarquons pour finir que, avec beaucoup plus de précautions que ci-dessus, nous pouvons considérer que les règles en `Prolog` jouent le même rôle que les sous-programmes ou les fonctions pour un langage de 3e génération. Si, donc, on tient compte des équivalences établies – avec des precautions – entre `Prolog` et les langages de 3e génération, on peut dire qu'en `Prolog`, programmes et données sont melangées et ne font qu'une entité – la base de données. Cet aspect des choses, c'est-à-dire le melange programmes et données, constitue une des grandes particularités des langages de 5e génération. Comme on verra plus tard nous pouvons modifier par programme la base de données. Donc un programme qui est en train de se dérouler, pourrait s'auto-modifier, c'est-à-dire nous pouvons avoir des modifications dynamiques des parties d'un programme qui ne sont pas prévues par son code mais sont dues à la nature des données.

5.1.3 Les questions

La dernière forme de l'expression logique en `Prolog` ce sont les questions. Une question est en réalité posée par l'utilisateur. `Prolog` parcourt sa base de données – à savoir les faits et les règles – afin de vérifier si la question est filtrée soit par les faits qu'il possède (c'est-à-dire le fait de la question est filtré par les faits de la base de données), soit par un fait issu par application des règles qu'il possède sur les faits de sa base de données. Ainsi la réponse à une question est conditionnée par la possibilité que la question est ou n'est pas une conséquence logique de la base de données.

Une question a la forme suivante :

$$\texttt{nomRelation(objet}_1\texttt{, objet}_2\texttt{, ... , objet}_n\texttt{)?}$$

On dit aussi qu'une question est un *but*. Remarquons qu'une question est une clause avec une entête vide.

5.1.4 Présentation en `Prolog`

Prolog fonctionne avec des clauses de Horn. Leur syntaxe est la suivante :

— Faits
`humain(socrate).`
On affirme le fait que Socrate est un être humain.

— Règles
`porteParapluie :- ilPleut.`
On lira ce morceau de programme "je porte un parapluie SI il pleut". Ici la notation "`:-`" se lit "SI".

— Variables
Le nom d'une variable commence toujours par une lettre majuscule. On distingue des variables

- universelles
  `mortel(X) :- humain(X).`
  `X` ici est une variable universelle : tout être humain est mortel.

– existentielles
`existeHumain :- humain(X).`
`X` est ici une variable existentielle. On cherche à savoir s'il existe un être humain.

MISE EN PRATIQUE

La plus simple manière pour exprimer une relation est de disposer d'une liste de faits, c'est-à-dire établir une base de données avec des faits. Par exemple construisons une base de données qui contient les noms de différentes personnes :

```
    homme(socrate).
homme(aristote).
```

Remarquons que même les noms propres doivent commencer par une lettre minuscule, car dans le cas contraire Prolog considérera qu'il s'agit d'une variable.

Nous pouvons maintenant poser des questions. Par exemple

```
    ?- homme(socrate).
==> true
```

mais

```
    ?- homme(toto).
==> false
```

Cette réponse peut paraître surprenante car Toto est bien un homme, peut-être même un de nos copains. On doit comprendre que si quelque chose n'est pas déclaré dans la base de données, Prolog, comme d'ailleurs tous les langages de programmation, considère qu'il n'existe pas. Pour s'en convaincre, voici la réponse de Prolog à la négation de la dernière question :

```
    ?- not(homme(toto)).
==> true
```

Nous pouvons aussi voir défiler toute la base de données en utilisant les variables comme suit :

```
    ?- homme(X).
==> X = socrate ;
X = aristote.
```

Cette réponse est obtenue parce que Prolog a unifié la variable `X` avec chaque élément de la base de données `homme`.

Nous pouvons ajouter une nouvelle base de données concernant les hommes qui sont mortels([1]) :

```
    mortel(socrate).
mortel(aristote).
```

Nous pouvons maintenant poser des questions composées, comme par exemple :

```
?- mortel(socrate), mortel(aristote).
==> true
```

[1] Bien sûr nous savons que tout homme est mortel. Mais Prolog ne le sait pas et nous ferons en sorte qu'il puisse l'envisager et, donc, de nous le dire.

Mais, comme nous avons dit, notre objectif est d'arriver à stocker une connaissance non pas factuelle, véhiculée par les bases de données, mais une connaissance générale, une loi ou, comme on dit en Prolog et en IA, une *règle* applicable sur des faits particuliers.

Il faut pour cela réfléchir. Nous avons deux bases de données `homme` et `mortel` que nous voulons mettre en association. Ceci est a priori envisageable parce que ces deux bases ont quelques (ici tous) éléments en commun. Une relation peut être, entre autre, une relation de causalité (s'il pleut, je prends mon parapluie) ou d'antériorité (j'initialise d'abord `somme` à 0 et ensuite je rajoute des éléments dans `somme`). Dans notre cas nous avons une relation de causalité. Il faut maintenant trouver l'élément qui est la cause et l'élément qui est l'effet. Si on considère que mortel est la cause de l'homme, on peut se rendre compte que ce n'est pas vrai en général car même le soleil est mortel et le soleil n'est pas un homme ! Par contre un homme est toujours mortel. Par conséquent nous avons la formule bien formée $\neg homme \vee mortel$ ce qui en Prolog s'écrit

```
mortel(X) :- homme(X).
```

qui est une règle. En utilisant cette règle, on a

```
mortel(socrate).
==> true
```

ce que l'on savait déjà. Ce qui est intéressant ici ce que nous pouvons des nouvelles connaissances sous forme explicite. Par exemple si on ajoute le fait `homme(toto).`, alors on a

```
?- mortel(toto).
==> true
```

c'est-à-dire nous obtenons une nouvelle connaissance qui, certes, était latente dans la base de données mais l'ordinateur n'y avait pas accès et par conséquent il ne pouvait pas l'utiliser.

5.2 Les données

Si on se réfère à la logique computationnelle, toutes les données en `Prolog` sont des termes. Les différents types de termes que nous avons déjà vus en logique computationnelle on les retrouve tout naturellement en `Prolog`, saupoudrés en plus avec des épices propres à ce langage. Ainsi une distinction fondamentale pour les données en `Prolog` s'opère entre les données simples et les données structurées.

5.2.1 Les données simples

`Prolog` est un langage absolument, ou mieux, viscéralement non typé. L'avantage de ce fait est que nul part on ne déclare qu'un élément est un tableau ou un réel ou une chaîne de caractères. On écrit un programme comme on construit un discours sans avoir à établir d'avance quels seront les adjectifs, les verbes ou les noms communs que nous utiliserons. Mais malgré tout, il faut au moins pouvoir faire la distinction entre constantes et variables. En `Prolog` cette distinction se fait de manière implicite. Le nom d'une variable doit toujours commencer par une lettre majuscule, par exemple `Variable`, tandis que celui d'une constante par une lettre minuscule, par exemple `cONSTANTE`.

On distingue les constantes en nombres et atomes. Les nombres peuvent être des entiers ou des réels. Tout ce qui n'est pas nombre et qui commence par une lettre miniscule, peut être considéré comme un atome. Par exemple `toto` ou `av_du_Parc_95000_Cergy` sont des atomes.

De même une chaîne de caractères alphanumériques entourée de simple quote « ' » est aussi un atome, e.g. `'Toto', '1,av. du Parc, 95000 Cergy' ou '3a'` sont des atomes.

Les variables commencent toujours par une lettre majuscule ou par le caractère « _ ». Il y a aussi la variable anonyme, représentée par « _ » et qui peut être unifiée à n'importe quelle variable ou constante pour laquelle il n'y aura pas dans le programme un usage explicite. Par exemple considérons l'ensemble de règles :

$$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$$
$$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$$

où la première ne fait pas appel à $\texttt{objet}_3$ et la seconde n'utilise pas $\texttt{objet}_2$. On peut donc les écrire sous la forme:

$$\texttt{nomRelation}(\texttt{objet}_1, \texttt{objet}_2, \_) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$$

$$\texttt{nomFait}(\texttt{objet}_1, \_, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$$

5.2.2 Les données structurées

Les constantes et les variables sont des données brutes, sans aucune structuration. On peut aussi envisager des données structurées, par exemple des données contenues dans des bases de données. Ainsi on peut envisager une base de données contenant des adresses des personnes selon l'exemple suivant : `adresse('Toto', 1, av, 'du Parc', 95000, 'Cergy')` et dont l'explication est évidente. En se référant au cours de la logique computationnelle, on constate que `adresse` représente un foncteur. Bien évidemment, dans la mesure où `Prolog` est un langage de logique d'ordre 1, nous pouvons envisager d'avoir comme arguments d'un foncteur d'autres foncteurs. Ainsi on peut aussi écrire `adresse(nom('Toto'), numero(1), rue(av, 'du Parc'), ville(95000, 'Cergy'))` à la place du foncteur précédent et où `nom, numero, rue, ville` sont aussi des foncteurs.

Un autre type des données structurées, sont les prédicats qui, comme nous le savons, expriment des valeurs de vérité concernant des relations. Par exemple le prédicat `couleurRouge (titreLivre, rouge)` est un prédicat qui a la valeur 1 si la couleur du livre `titreLivre` est rouge. Il est possible aussi, selon la remarque précédente, que les arguments d'un prédicat soient des foncteurs. Ainsi la valeur du prédicat `couleurRouge(titre(TitreLivre), couleur(Couleur))` est égale à 1 si la variable `Couleur` est unifiée à la couleur `rouge` et à 0 sinon. Ici `titre` et `couleur` sont des foncteurs.

La distinction, en programmation `Prolog`, entre foncteurs et prédicats est parfois assez subtile et, de toute fa, elle est toujours laissée à la charge du programmeur, c'est-à-dire, en clair, `Prolog` n'est pas capable de distinguer entre prédicats et foncteurs et, pour s'en sortir, il applique à la lettre les règles établies par la logique des prédicats. Ainsi, si on écrit en `Prolog`, la règle

```
couleurRouge(X) :- livre(titreLivre(X),couleur(rouge)).
```

accompagnée des faits :

```
livre(titreLivre(petitLivre), couleur(rouge)).
livre(titreLivre(vert), couleur(verte)).
```

on obtient pour la question `couleurRouge(petitLivre).` la réponse `oui` et pour la question `couleurRouge(vert).` la réponse `non`. Mais, grâce à la particularité de `Prolog` de pouvoir répon-

dre à des questions symétriques, on peut aussi poser la question `couleurRouge(X).` et avoir comme réponse non pas une réponse affirmative, à laquelle il fallait s'attendre du fait que dans la base de données il y a un livre de couleur rouge, mais carrément son titre, à savoir `X=petitLivre`, comme si le prédicat `couleurRouge` était un foncteur. Ainsi si on écrit dans le programme, la ligne supplémentaire :

```
bibliothequeRouge(couleurRouge(X)).
```

le compilateur (ou l'interpréteur) de `Prolog` ne s'aperît pas que `couleurRouge(X)` soit un prédicat et en tant que tel ne doit pas apparaître comme un argument. La surprise viendra si on veut connaître tous les livres rouges d'une bibliothèque et, tout naturellement, on pose la question : `bibliothequeRouge(couleurRouge(X)).` Dans ce cas la seule réponse qu'on reest `X` égale au nom de la variable interne avec laquelle `Prolog` a unifié `X`.

5.2.3 Types des données

Les principaux types des données en Prolog, sont les suivants :

— Des entiers. Ex. `factoriel(6).`

— Des réels. Ex. `pi(3.1415).`

— Des constantes. Ex. `socrate, toto, ...` . Notons que les nombres entiers ou flottants sont considérés comme des constantes.

— Prédicats avec termes : Exemple. Description des branches gauche et droite d'un arbre binaire.

```
brancheGauche(arbre(L,R), L).
brancheDroite(arbre(L,R),R).
```

Ici `brancheGauche` et `brancheDroite` sont des prédicats, `arbre` est un foncteur.

5.3 Fonctionnement (simplifié) de `Prolog`

`Prolog` est principalement un langage interprété, ce qui signifie que l'ordinateur exécute immédiatement les commandes saisies par l'utilisateur : le code source est immédiatement traduit en code machine. Concrètement, `Prolog` consiste en un interpréteur où on peut saisir des expressions après le prompt (?-), et un moteur d'inférence qui teste si ces expressions sont vraies ou fausses (par unification) en parcourant la base de faits (par backtracking). Le moteur d'inférence fonctionne selon une approche *top-down*, c'est-à-dire il prend le premier élément de la base, il essaie de le prouver et il continue avec le suivant.

Pour exécuter un programme `Prolog` il faut d'abord le charger dans le fenêtre de travail à l'aide de la commande `?-consult ('nomProgramme.pl').`. Ensuite il faut poser la question.

Afin de répondre à une question posée, `Prolog` est toujours capable de construire, à partir de sa base de données, une arborescence dont la racine est la question posée. Ensuite `Prolog` parcourt cette arborescence en appliquant l'algorithme de résolution SLD, que nous avons vu en logique computationnelle, sans toutefois faire la vérification des occurrences. Si, en appliquant cet algorithme, il arrive à « filtrer » la question, alors il affiche le message `yes` et, en fonction de la question posée, le contenu des variables de la question exemplifiées (instanciées) à des constantes. `Prolog` est en mesure de fournir toutes les réponses contenues dans la base de données.

Pour les avoir, il suffit, après chaque réponse affichée, de taper la touche «; » qui, rappelons-le, en `Prolog` signifie « `ou` ». Si, par contre on veut s'arrêter, alors il faut taper la touche « Retour ».

Pour illustrer le fonctionnement de `Prolog` considérons le programme E suivant :

```
1 p.
2 q.
```

Le programme s'écrit sous forme ensembliste `E = {p, q}`. Soit maintenant la question.

`p ∧ q`

Pour répondre à la question,Prolog applique la résolution par réfutation. Donc dans E est placée aussi la négation de la question

`E = {p, q, (¬p ∧ q)} = {p, q, ¬p ∨ ¬ q}`

qui nous donne la clause absurde $\perp$. En conséquence le programme induit la fbf `p ∧ q`, c'est-à-dire `E` $\models$ `p ∧ q`.

5.4 Repésentation des nombres naturels

Considérons l'univers de discours $\mathcal{U}$ composé de l'ensemble de nombres naturels et de l'opération de l'addition, munie de son élément neutre 0. Si on veut construire un langage $\mathcal{L}_0$ dont on chercherait une interprétation dans $\mathcal{U}$, on doit avoir un foncteur équivalent à l'opération de l'addition, ainsi que l'élément neutre. Plaçons-nous dans le cadre d'un Prolog pur, c'est-à-dire d'un Prolog qui ne contient pas des formes arithmétiques. Convenons d'appeler `add/3` le prédicat qui représente l'opération d'addition dans $\mathcal{U}$ [2]. Notons aussi par `zero` l'élément neutre de `add` qui est, par ailleurs, une des constantes du langage $\mathcal{L}_0$. Il nous faut aussi un prédicat, que nous appellerons `nat/1` pour caractériser les nombres naturels. On pourra ainsi écrire :

```
nat(zero).
```

pour indiquer que zero est un nombre naturel. La question qui se pose maintenant concerne les autres nombres naturels, à savoir de quelle manière nous allons représenter ces nombres. Depuis Peano, au moins, on sait que nous pouvons construire l'ensemble des nombres naturels à partir de 0 et de l'opérateur de succession $s(X) = X + 1$, où X est un nombre naturel. On peut utiliser cette même technique pour le langage $\mathcal{L}_0$. On se dote donc d'un foncteur `s/1` qui représente l'opérateur de succession dans $\mathcal{L}_0$ et, par conséquent, le programme complet de caractérisation des nombres naturels s'écrit :

```
nat(zero).
nat(s(X)) :- nat(X).
```

On peut, par exemple, poser la question

```
?- nat(s(s(s(s(zero))))).
```

et on aura comme réponse `yes`. Mais le plus rigolo c'est quand on pose la question

```
?- nat(X).
```

Dans ce cas on recupère comme réponse

[2] `add/3` signifie que le foncteur `add` est d'arité 3, c'est-à-dire que le nombre de ses arguments est 3.

```
X = zero ;
X = s(zero) ;
X = s(s(zero)) ;
X = s(s(s(zero))) ;
X = s(s(s(s(zero)))) ;
X = s(s(s(s(s(zero)))));
. . . . . .
```

c'est-à-dire l'ensemble des nombres naturels.

Avant d'avancer en programmation, essayons de voir, sous l'aspect logique des prédicats, ce que nous venons de construire. Nous avons une logique du premier ordre $\mathcal{L}_0$ dont les termes sont $\mathbb{T} = \{\texttt{zero}, \texttt{s}/1, \texttt{X}, \texttt{Y}\}$ et nous avons aussi les prédicats $\{\texttt{nat}/1, \texttt{add}/3\}$. L'interprétation I que nous avons adoptée conduit à la signification ϕ_I suivante des termes : (cf. Définition 4.3.3, p.52 du poly de logique) :

- $\texttt{zero} \rightarrow \phi_I(\texttt{zero}) = \texttt{zero}_I = 0$
- $\texttt{s(X)} \rightarrow \phi_I\left(\texttt{s}(\overline{\phi}_I(\texttt{X}))\right) = \texttt{s}_I(\overline{\phi}_I(\texttt{X})) = \overline{\phi}_I(\texttt{X}) + 1$
- $\texttt{add}(\texttt{X},\texttt{Y},\texttt{Z}) \rightarrow \phi_I\left(\texttt{add}(\overline{\phi}_I(\texttt{X}), \overline{\phi}_I(\texttt{Y}), \overline{\phi}_I(\texttt{Z}))\right) = \texttt{add}_I\left(\texttt{add}(\overline{\phi}_I(\texttt{X}), \overline{\phi}_I(\texttt{Y}), \overline{\phi}_I(\texttt{Z})\right)$ d'où $\overline{\phi}_I(\texttt{X}) + \overline{\phi}_I(\texttt{Y}) = \overline{\phi}_I(\texttt{Z})$.

Pour écrire le programme de l'addition de deux naturels en Prolog nous allons utiliser les deux remarques suivantes :

(1) Si on ajoute 0 à une valeur X, le résultat ets X. En Prolog on a

```
add(zero,X,X).
```

(2) Si on a X+Y=Z, alors (X+1) + Y = (Z+1). En Prolog on a

```
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

On peut, par exemple, faire l'addition

```
?- add(s(s(s(s(s(zero))))), s(s(zero)), Z).
```

et avoir comme réponse

```
Z = s(s(s(s(s(s(s(zero)))))))
```

Mais on peut aussi faire la soustraction entre le troisième terme et le premier

```
?- add(s(s(s(s(s(zero))))), Y, s(s(s(s(s(s(s(zero))))))).
```

avec réponse

```
Y = s(s(zero))
```

et aussi entre le troisième terme et le deuxième

```
?- add(X, s(s(zero)), s(s(s(s(s(s(s(zero))))))).
```

avec réponse

```
X = s(s(s(s(zero))))
```

5.5 Les opérateurs de base de Prolog

Toutes les clauses se terminent par un point ".". Les atomes et les symboles de fonctions commencent par une lettre minuscule et les variables par une lettre majuscule. Le tiré bas (ou souligné) représente une variable anonyme : on l'utilise pour indiquer que la variable existe mais on n'a pas besoin de conna
ccitre sa valeur. Le ET est représenté par une virgule, le OU par un point-virgule.

Les opérateurs d'égalité et de comparaison sont les suivnats :

— **=** Unifie deux termes : `X = Y` (`X` et `Y` sont des termes quelconques). Si `X` et `Y` sont non instanciés, `X = Y = _`(variable anonyme). Si `X` instanciée à un atome et `Y` est non instancié, `Y` s'instancie à l'instance de `X`. Les variables instanciées sont toujours égales à elle-mêmes, par exemple `2=2` est vrai.

— **==** Vérifie si 2 termes sont identiques

— **<** Vérifie si un terme est plus petit qu'un autre

— **=<** Vérifie si un terme est plus petit ou égal à un autre

— **>** Vérifie si un terme est plus grand qu'un autre

— **>=** Vérifie si un terme est plus grand ou égal à un autre

— `\=` Teste la non-unification de 2 termes

— `\==` Teste si 2 termes ne sont pas identiques

Les opérateurs arithmétiques :

— **<** Plus petit

— **=<** Plus petit ou égal

— **>** Plus grand

— **>=** Plus grand ou égal

— **=:=** Égal

— `=\=` Différent

— **is** Evaluation d'une expression et assignation à une variable

5.6 Un exemple

Considérons le graphe de la figure 5.1 :

Si nous utilisons la logique des propositions nous pouvons représenter, en Prolog ce graphe par la base de données suivante :

PROGRAMME 5.6.1

```
1  gr(a,b,2).
2  gr(a,g,6).
3  gr(b,e,2).
4  gr(b,c,7).
5  gr(g,e,1).
6  gr(g,h,4).
```

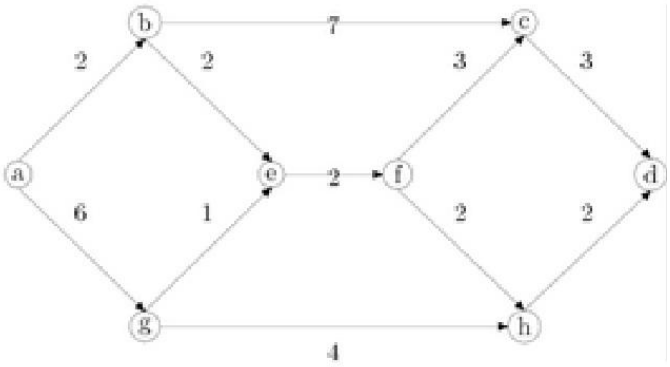


Figure 5.1: Graphe pour un réseau

```
7     gr(e,f,2).
8     gr(f,c,3).
9     gr(f,h,2).
10    gr(c,d,3).
11    gr(h,d,2).
```

Chacune de ces clauses représente une proposition qui a une valeur de vérité – vrai ou faux. Si on pose la question

`?-gr(b,c,7).`

alors la réponse est `oui`, c'est-à-dire vrai, tandis que la réponse à la question

`?-gr(b,f,10)`

est bien évidemment `non`, c'est-à-dire faux. Si donc on représente par E l'ensemble de clauses : $E = \{gr(a,b,2), gr(a,g,6), \ \ldots, gr(h,d,2)\}$, alors on obtient une réponse positive à une question composée de la clause q si et seulement si $E \models q$. Il est évident que cette condition est remplie si $q \in \widetilde{\mathcal{I}}_E$ qui est le modèle minimal d'Herbrand du programme E. Dans la mesure où le programme E est, dans notre cas, composé des clauses filtrées, nous avons comme modèle minimal d'Herbrand, les prédicats qui font partie des clauses du programme, ce qui en absence des règles donne: $\widetilde{\mathcal{I}}_E = E$. Par conséquent les seules fbf qui peuvent être satisfiables par E sont les clauses du programme. On voit ainsi que la logique des propositions est une logique « pauvre » qui ne permet pas d'exprimer toute la richesse du raisonnement humain.

5.7 Exercices

EXERCICE 5.1 *Écrire le programme Prolog* `mult(X,Y,Z)` *qui calcule le produit de deux naturels* `X,Y` *et stocke le résultat dans* `Z`.

Exemple : `mult(s(s(zero)), s(s(s(zero))), Z)` *donnera comme résultat* `Z = s(s(s(s(s(zero)))))`.

EXERCICE 5.2 *En utilisant le programme précédent, calculer le résultat de la division de deux naturels* `X,Y`. *On fait l'hypothèse que* `X` *divise* `Y`, *c'est-à-dire que Y/X est un entier.*

EXERCICE 5.3 *Écrire le programme Prolog* `egal(X,Y)` *qui teste si les variables* `X,Y` *ont la même*

valeur.

EXERCICE 5.4 *Écrire le programme Prolog* `pair(X)` *qui teste si la valeur de* `X` *est paire. On fera l'hypothèse que* `zero` *est pair.*

EXERCICE 5.5 *Même chose pour le prédicat* `impair(X)`.

EXERCICE 5.6 *Examiner la base de données de l'exemple 5.6. Établir la notion d'un sommet successeur d'un autre. Par exemple le sommet* d *est successeur du sommet* a.

EXERCICE 5.7 *Considérons le texte suivant :*

Si Cyclope est un personnage mythologique, alors il est immortel mais s'il n'est pas un personnage mythologique, alors il est mortel. Si Cyclope est soit immortel, soit mortel, alors il a un seul œil. Cyclope est magique s'il a un seul œil.

(1) Représenter ce texte à l'aide d'une base de connaissances avec des propositions de la logique propositionnelle.
(2) Donner les modèles pour que Cyclope soit magique si nous faisons l'hypothèse qu'il soit un personnage mythologique.
(3) Écrire en `Prolog` *un programme qui permet de répondre aux questions suivantes :*
  (a) Cyclope est-il mythique?
  (b) Cyclope est-il magique?
  (c) Cyclope a-t-il un œil?

EXERCICE 5.8 « A moins que nous continuons la politique de soutien des prix, nous perdrons les voix des agriculteurs. Si nous continuons cette politique, la surproduction continuera, sauf si nous contingentons la production. Sans les voix des agriculteurs, nous ne serons pas réélus. Donc si nous sommes réélus et si nous ne contingentons la production, il continuera d'y avoir surproduction. »

(1) Représenter ce texte à l'aide d'une base de connaissances avec des propositions de la logique propositionnelle.
(2) Écrire un programme en `Prolog` *qui teste la validité logique de la conclusion.*

EXERCICE 5.9 *La stratégie de recherche en profondeur d'abord de* `Prolog` *est incomplète, c'est-à-dire elle ne garantit pas de trouver une solution si elle existe car ce type de recherche peut conduire à une branche infinie de l'arbre de résolution.*

Pour vérifier cette incomplétude on utilise la base de données suivante Pour examiner d'abord la branche infinie, il faut permuter les clause (2) et (3).

(1) a :- b , c.
(2) b :- c , d.
(3) b :- a , d.
(4) c :- e.

(5) d.

(6) e.

Supposons que le but à démontrer est b.

Établir l'arbre de résolution pour ce programme et aussi quand on échange la place des clauses 2 et 3.

EXERCICE 5.10 *Soit le problème suivant :*

Trois coureurs Toto, Koko et Lolo portent des maillots de couleur différente et courent ensemble lors d'une course. Nous allons essayer d'établir l'ordre de l'arrivée de ces coureurs. Pour cela nous disposons des informations suivantes :

Toto dit qu'il est arrivé avant le coureur qui porte le maillot rouge. Koko, qui porte le maillot jaune, dit qu'il est arrivé avant le coureur au maillot vert.

Utiliser `Prolog` *pour trouver l'ordre d'arrivée des coureurs.*

TABLE DES MATIÈRES