Chrysostome Baskiotis

LOGIQUE COMPUTATIONNELLE & PROLOG

Fascicule 2

# Calcul des prédicats
# Programmation logique



EISTI

Année 2011

# 6

# CALCUL DES PRÉDICATS



Pour pouvoir utiliser la logique en tant que mode de calcul il faut d'abord procéder à une conceptualisation de la partie du monde qui nous intéresse, c'est-à-dire établir les objets du monde et leurs inter-relations. Il s'agit, en réalité, de procéder à une *représentation de la connaissance*. Les objets sur lesquels nous allons exprimer notre connaissance constituent l'*univers du discours*. La représentation de la connaissance est la mise en relation des plusieurs objets de l'univers du discours par l'intermédiaire de la formulation des fbf.

Avec le calcul propositionnel on ne peut pas formaliser toutes les connaissances relatives à un univers du discours. Par exemple la phrase « tous les élèves aiment la logique » sera vue en calcul propositionnele comme une proposition $p$. On ne pourra donc pas la distinguer d'une autre du type « il existe des élèves qui aiment la logique ». qui sera, elle aussi, notée $p$. Il faut donc introduire dans le langage des symboles qui désignent soit l'existence, soit l'universalité et, aussi, des variables.

Nous avons ainsi un cas particulier de ce qu'on pourrait considérer comme étant des *fonctions propositionnelles logiques* et qui sont des fonctions au sens classique du terme mais dont le résultat est une valeur de vérité - 0 (faux) ou 1 (vrai). Ces fonctions propositionnelles logiques on les appellera, par la suite, *prédicats*.

Il existe bien sûr toujours des fonctions au sens classique du terme, c-à-d. des fonctions dont le résultat est un objet de l'univers du discours et non pas une valeur de vérité. Pour les distinguer des autres fonctions, qui sont les prédicats, nous les appellerons *foncteurs*.

Nous allons maintenant, comme nous l'avons fait au chapitre précédent, développer un langage qui permettra la création et l'étude des prédicats. C'est l'objet du présent chapitre. On débute avec la définition du langage. On continue avec les évaluations sémantique et syntaxique d'une fbf avant d'aborder leur équivalence. On termine avec la définition des formes clausales.

## 6.1 Les éléments du langage

Pour l'étude des prédicats nous allons définir un langage formel $\mathcal{L}_1$ dont l'alphabet $A_1$ est composé des éléments suivants :

- Un ensemble **V**, au plus dénombrable, des variables qui seront notées $x, y, \ldots$ ou $X, Y, \ldots$.
- Un ensemble $\Xi$, au plus dénombrable, des constantes qui seront notées $a, b, \ldots$.
- Un ensemble **F** de fonctions $f : \mathbf{V} \times \Xi \rightarrow \mathbf{V} \cup \Xi$ d'arité quelconque. Notons que l'arité d'une fonction est le nombre d'arguments de la fonction. Remarquons qu'une constate est une fonction d'arité zéro. On appelle ces fonctions des *foncteurs* que l'on notera $f, g, \ldots$ ou $F, G, \ldots$.
- Un ensemble **P** de fonctions d'arité quelconque $f : \mathbf{V} \times \Xi \rightarrow \{Vrai, Faux\}$ qui seront appelées *prédicats* et qui seront notés par $p, q, \ldots$ ou $P, Q, \ldots$. Un prédicat particulier est la relation d'égalité qui sera notée soit de façon fonctionnelle $eg(x, y)$, soit, lorsqu'il n'y a pas risque de confusion, $x = y$. À ce propos, notons qu'il faut clairement distinguer entre *égalité* "$=$" et *équivalence* "$\equiv$". La égalité entre deux termes est un prédicat qui indique si oui ou non les deux termes ont la même valeur. L'équivalence entre deux fbf indique que les deux fbf ont la même table de vérité. La question peut se poser si deux termes égaux sont équivalents. La réponse est non. Par exemple nous avons "10= 3+7" mais par rapport au prédicat *nombre pair* il n'y a pas d'équivalence. De même deux fbf équivalentes ne sont pas égales. Par exemple la fbf $a \vee \neg a$ est équivalente à la fbf $a \vee (a \rightarrow b)$ car toutes les deux sont des tautologies, mais elles ne sont pas égales.
- Un ensemble **L** des connecteurs et quantificateurs :
  - Connecteur logique unaire : la négation $\neg$
  - Connecteurs propositionnels binaires :
    - Conjonction : $\wedge$
    - Disjonction : $\vee$
    - Implication : $\rightarrow$
    - Équivalence (ou double implication) : $\leftrightarrow$
  - Quantificateurs :
    - Quantificateur existentiel : $\exists$
    - Quantificateur universel : $\forall$
- Les séparateurs (symboles auxiliaires) : $($, $)$, $[$, $]$ .

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

Les éléments du langage déterminent un alphabet $A_1 = \{\mathbf{V}, \Xi, \mathbf{F}, \mathbf{P}, \mathbf{L}\}$.

### 6.1.1 Les termes

La brique élémentaire du calcul des prédicats est le *terme*. Une variable est un terme. Une constante aussi. Plus généralement

DÉFINITION 6.1.1 *Un* terme *est défini de façon itérative comme suit :*

- *une constante est un terme ;*
- *une variable est un terme ;*
- *si $f$ est un foncteur d'arité $n$ et $t_1, t_2, \ldots, t_n$ sont des termes, alors $f(t_1, t_2, \ldots, t_n)$ est un terme.*

*Tout terme est obtenu par application des règles précédentes un nombre fini de fois.*
*L'ensemble de termes sera noté par $\mathbb{T}$ .*

On peut construire, à partir des termes, des formules bien formées (fbf).

DÉFINITION 6.1.2 *Soit $\mathbb{T}$ l'ensemble des termes sur un alphabet $A_1$. L'ensemble $\mathbb{F}$ des* formules bien formées *(par rapport à $A_1$) est le plus petit ensemble tel que :*

- *Si $p$ est un prédicat d'arité $n$ et $t_1, t_2, \ldots, t_n$ sont des termes, alors $p(t_1, t_2, \ldots, t_n) \in \mathbb{F}$ .*
- *Si $F$ , $G \in \mathbb{F}$ alors les constructions suivantes :*
  - $\neg F$
  - $F \vee G$ , $F \wedge G$
  - $F \rightarrow G$ , $F \leftrightarrow G$

  *sont aussi des fbf.*
- *Si $F \in \mathbb{F}$ et $X$ est une variable, alors $(\forall X) F \in \mathbb{F}$ et $(\exists X) F \in \mathbb{F}$ .*

ASCÈSE 6.1 *Donner l'équivalent en français de deux fbf suivantes :*

*(1)* $\exists X \, (p(X) \wedge (\forall Y \, p(Y) \rightarrow X = Y))$

*(2)* $(\exists X \, p(X)) \wedge (\forall X \forall Y \, p(X) \wedge p(Y) \rightarrow X = Y)$

L'équivalent de l'atome de la logique des propositions est donné par la définition suivante :

DÉFINITION 6.1.3 *Si $p$ est un prédicat d'arité $n$ et $t_1, t_2, \ldots, t_n$ sont des termes, alors $p(t_1, t_2, \ldots, t_n)$ est une* formule atomique.

DÉFINITION 6.1.4 *Le triplet*

$$\mathcal{L}_1 = \{A_1, \mathbb{T}, \mathbb{F}\}$$

*est le* langage d'ordre un *ou le* langage du calcul des prédicats

Par construction $\mathbb{F}$ est un ensemble dénombrable, défini itérativement.

ASCÈSE 6.2 *On se place dans l'ensemble des nombres réels. Écrire, en utilisant le langage des prédicats, les propositions suivantes :*

*(1) Pour tout réel il existe un autre réel qui est plus grand.*

*(2) Il existe un réel qui est plus grand que tout autre réel.*

(3) *Chaque réel positif est un carré.*

(4) *Si un réel est plus petit qu'un autre réel, alors il existe un troisième réel, différent de deux précédents, et qui est entre les deux.*

ASCÈSE 6.3 *Pour les propositions de l'ascèse précédent, déterminer les prédicats et les foncteurs utilisés.*

ASCÈSE 6.4 *Soit les textes suivants :*

(1) *Tous les hommes sont mortels. Socrate est un homme. Donc, Socrate est mortel.*

(2) *Toute personne saine d'esprit peut comprendre la logique. Aucun des fils de Socrate ne peut comprendre la logique. Aucune personne non saine d'esprit a le droit de vote. Donc, aucun des fils de Socrate n'a le droit de vote.*

*Pour chaque texte*

(1) *Écrire la fbf correspondante.*

(2) *Montrer que chaque fbf est valide.*

### 6.1.2 Les quantificateurs et leur portée

Les quantificateurs ont une *portée*.

DÉFINITION 6.1.5 *La* portée *d'un quantificateur dans une formule est la partie de la formule qui se trouve sous l'influence du quantificateur.*
*Une variable d'une formule qui est sous la portée d'un quantificateur de la même variable est appelée* variable liée *(ou* bornée*). Sinon elle est une* variable libre.
*Une formule qui n'a pas des variables libres s'appelle* formule close. *Une formule qui n'a pas des variables s'appelle* formule filtrée.
*Une formule sans quantificateurs s'appelle* formule ouverte.

ASCÈSE 6.5 *Pour les fbf ci-après indiquer les variables libres et les variables liées.*

(1) $\exists y\ (p(r) \wedge f(x,y) \vee q(y) \wedge f(x,y)) \leftrightarrow p(r) \vee q(x) \vee p(y)$

(2) $\forall x \exists y\ (p(w) \vee q(z)) \vee \exists z\ (q(w) \vee p(r))$

(3) $\forall x \exists y\ (p(w) \vee q(z)) \vee \exists z\ (q(w) \vee p(y))$

(4) $\forall x\,(\forall z p(y) \wedge \neg\,(q(z) \wedge s(y)) \rightarrow \exists y\,(q(y) \vee p(x) \vee s(z))) \wedge (p(z) \wedge s(y))$

Pour résoudre les problèmes de l'ambiguïté entre démonstration sous forme conditionnelle et démonstration sous forme générale, on introduit la définition ci-après :

DÉFINITION 6.1.6 *Soit $F$ une fbf avec variables libres $x_1, x_2, \ldots, x_n$.*
*La* clôture universelle *de $F$ , notée $\forall F$, est la formule close $\forall x_1, \forall x_2, \ldots, \forall x_n\ F$.*
*La* clôture existentielle *de $F$, notée $\exists F$, est la formule close $\exists x_1, \exists x_2, \ldots, \exists x_n\ F$.*

Notons qu'il est obligatoire quand on passe d'une fbf $F$ à une clôture (soit universelle, soit existentielle) de renommer, par substitution, les variables qui ont le même nom mais qui ne dépendent pas du même quantificateur afin d'éliminer les ambiguïtés. Par exemple pour la fbf $\forall x \exists y\,(p(x) \wedge q(y)) \vee (\exists y \neg p(x) \wedge q(y))$ on doit renommer la deuxième occurrence de $x$ et de $y$ et obtenir ainsi $\forall x \exists y\,(p(x) \wedge q(y)) \vee (\exists z \neg p(x) \wedge q(z))$.

ASCÈSE 6.6 *On se place dans le cadre des nombres entiers et on suppose que nous disposons des opérations (foncteurs) d'addition ($+$) et de multiplication ($\times$). Donner, en langage des prédicats, la définition des prédicats suivants :*

*(1)* $pair(x)$ = *x est un nombre pair.*

*(2)* $div(x,y)$ = *x divise y, c'est-à-dire que* $y/x$ *est un nombre entier.*

*(3)* $premier(x)$ = *x est un nombre premier.*

ASCÈSE 6.7 *En appliquant la définition des prédicats pair et div, écrire le prédicat* $div2(x)$ = 2 *divise le nombre pair x.*

### 6.1.3 Propriétés des connecteurs

Les principales propriétés des connecteurs sont données ci-après :

(1) Double négation

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

(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

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

(7) Négation des quantificateurs universel et existentiel :

$$\neg(\forall X) \leftrightarrow (\exists X)\ ,\ \neg(\exists X) \leftrightarrow (\forall X)$$

Il est important de se rappeler que nous avons pour les négations les relations suivantes :

$$(\neg\forall X\ p(X)) \leftrightarrow (\exists X\ \neg p(X))$$

et

$$(\neg\exists X\ p(X)) \leftrightarrow (\forall X\ \neg p(X))$$

ASCÈSE 6.8 *Traduire en langage des prédicats dans* $\mathbb{N}$ *la phrase suivante :*

*(1)* $x \equiv y$ *modulo n.*

## 6.2 Substitution

Pendant le traitement d'une fbf, les variables peuvent être remplacées (substituées) par des termes ou, encore, par des fbf.

DÉFINITION 6.2.1 *Une* substitution *est un ensemble fini de couples de la forme* $(x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ *où chaque* $A_i$ *est soit un terme, soit une fbf et chaque* $x_i$ *une variable telle que* $x_i \neq A_i$ *et* $A_i \neq A_j$ *si* $x_i \neq x_j$ .

*La substitution vide sera notée par* $\varepsilon$.

La substitution peut être considérée comme une application de l'ensemble des variables $V$ dans celui des termes et des fbf. Ainsi soit l'ensemble $E$ qui contient les termes ou fbf $A_1, A_2, \ldots, A_n$. La substitution $\sigma = (x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ permet d'avoir la fbf $E\sigma$ qui est le résultat de l'application $\sigma$ à $E$ et qui est obtenu en remplaçant chaque occurrence dans $E$ de $x_i$ par $A_i$ , $i = 1, \ldots, n$. $E\sigma$ est appelé une *exemplification* (*instance*) de $E$.

DÉFINITION 6.2.2 *Soient* $\sigma = (x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ *et* $\tau = (y_1/B_1, y_2/B_2, \ldots, y_m/B_m)$ *deux substitutions. La composition* $\sigma\tau$ *de* $\sigma$ *et* $\tau$ *est obtenue à partir de l'ensemble* $\{x_1/A_1\tau, x_2/A_2\tau, \ldots, x_n/A_n\tau, y_1/B_1, y_2/B_2, \ldots, y_m/B_m\}$ *en supprimant*

- *tous les* $x_i/A_i\tau$ *pour lesquels on a* $x_i = A_i\tau$ , $i = 1, \ldots, n$
- *tous les* $y_j/B_j$ *pour lesquels on a* $y_j \in \{x_1, x_2, \ldots, x_n\}$ , $j = 1, \ldots, m$.

EXEMPLE 6.2.1 $\sigma = (x/Z, y/W)$ , $\tau = (x/a, Z/b, W/y) \Rightarrow \sigma\tau = (x/b, Z/b, W/y)$

La composition des substitutions n'est pas en général commutative.

ASCÈSE 6.9 *Vérifier la non commutativité de la substitution en utilisant l'exemple ci–dessus.*

Les propriétés des substitution sont les suivantes :

PROPRIÉTÉ 6.2.1 $E(\sigma\tau) = (E\sigma)\tau$

PROPRIÉTÉ 6.2.2 $(\rho\sigma)\tau = \rho(\sigma\tau)$

PROPRIÉTÉ 6.2.3 $\varepsilon\theta = \theta\varepsilon = \theta$

PROPRIÉTÉ 6.2.4 *Si* $\models E$, *alors* $\models E\sigma$

## 6.3 Interprétation sémantique - Modèles

La première tâche du calcul des prédicats est de donner un sens aux fbf, c'est-à-dire d'établir la valeur de vérité de ces formules. Néanmoins il n'est pas possible à chaque fbf d'associer une valeur de vérité, car, pour une formule atomique, il faut en même temps donner une valeur aux arguments du prédicat et au prédicat lui-même. Or la valeur du prédicat dépend des valeurs de ses arguments. Pour cette raison nous avons besoin d'une fonction d'interprétation qui donnera la valeur de vérité au prédicat en fonction des valeurs de ses arguments.

Nous donnons d'abord une présentation informelle de l'interprétation.

On considère un langage $\mathcal{L}$ du 1er ordre et un univers du discours $\mathcal{U}$. Nous allons essayer de mettre en association les éléments du langage d'une part et les éléments de l'univers du discours, d'autre part. Cette correspondance entre éléments du langage et éléments de l'univers du discours constitue ce qu'on appelle une *interprétation* $I$ du langage ou, encore, *structure*. Nous pouvons envisager que cette interprétation $I$ ne couvre pas la totalité de l'univers du discours $\mathcal{U}$ mais une partie seulement, notée $\mathbb{D}(I) \subset \mathcal{U}$ et qui est le *domaine* de l'interprétation.

La correspondance $I$ que nous venons de présenter n'est pas une application au sens habituel du terme mais plutôt un ensemble d'applications, chacune de ses applications étant spécifique à un type d'éléments de $\mathcal{L}$.

Ainsi pour les constantes $a$ de $\mathcal{L}$ il faut trouver des constantes $a_{I.}$ dans $\mathcal{U}$ pour les faire appliquer.

Pour un foncteur $f$ de $\mathcal{L}$ il faut trouver une relation $f_I$ dans $\mathcal{U}$ qui peut être utilisée comme une fonction à valeurs dans $\mathcal{U}$. Dans ce cas, si l'arité de $f$ est $n$, alors l'interprétation de $f(t_1,\ldots,t_n)$, où $t_i$ ce sont des termes, est $f_{I.}((t_1)_{I.},\ldots,(t_n)_I)$ avec $(t_i)_I$ interprétation du terme $t_i$.

Pour un prédicat $p$ dans $\mathcal{L}$ il faut trouver une relation $p_I$ dans $\mathcal{U}$ qui peut être utilisée comme une fonction ayant deux valeurs : 0 ou 1. Dans ce cas, si l'arité de $p$ est $n$, alors l'interprétation de $p(t_1,\ldots,t_n)$, où $t_i$ ce sont des termes, est $p_{I.}((t_1)_{I.},\ldots,(t_n)_I)$ avec $(t_i)_I$ interprétation du terme $t_i$.

Nous obtenons ainsi la définition suivante :

DÉFINITION 6.3.1 (INTERPRÉTATION) *Une* interprétation *$I$ sur un langage $\mathcal{L}$ est un domaine non vide $\mathbb{D}(I)$ (qui parfois est noté $\mid I \mid$), appelé* domaine de l'interprétation, *et une application qui associe :*

- *chaque constante $a \in \Xi$ avec un élément $a_I \in \mathbb{D}(I)$ ;*
- *chaque foncteur $f \in \mathbf{F}$ d'arité n avec une fonction $f_I : \mathbb{D}(I)^n \to \mathbb{D}(I)$ ;*
- *chaque prédicat $p \in \mathbf{P}$ d'arité n avec une relation $p_I \subseteq \mathbb{D}(I)^n$.*

Remarquons que la dernière association peut se comprendre comme étant une application de $\mathbb{D}(I)^n$ dans l'ensemble $\{0,1\}$ (faux, vrai).

ASCÈSE 6.10 *Considérons un langage $\mathcal{L}$ et un prédicat unaire $p \in \mathcal{L}$. Supposons que le domaine de définition de ce prédicat est l'ensemble $D = \{1,2\}$. Établissez pour le prédicat $p$ les quatre interprétations $I_i$ ; $i = 1,\ldots,4$ possibles et distinctes en utilisant comme domaine d'interprétation l'ensemble $D$.*

Grâce à cette définition de l'interprétation, on peut attribuer des valeurs de vérité aux fbf. En effet la valeur de vérité d'une fbf sera définie en fonction des valeurs de vérité de ses composantes qui sont soit des fbf à leur tour, soit des termes. Il faut donc pouvoir établir l'interprétation des termes. Dans la mesure où les termes contiennent des variables, il faut pouvoir les associer au domaine. Nous voyons donc que l'interprétation des variables du langage $\mathcal{L}$ pose un problème. Car pour pouvoir interpréter une variable $x$ il faudrait savoir quel objet désigne exactement $x$. Mais dans ce cas $x$ ne serait plus une variable. On s'en sort de cette situation embarrassante par un artifice du type suivant : les variables du langage sont interprétées comme étant des éléments variables de l'univers du discours ! Et pour formaliser cette belle interprétation on utilise ce qu'on appelle l'*assignation* des variables par rapport à une interprétation, qui est une application $\overline{\varphi}_I : V \to \mathbb{D}(I)$ où $V$ l'ensemble des variables du langage $\mathcal{L}$ c'est-à-dire formellement :

DÉFINITION 6.3.2 (SÉMANTIQUE DES VARIABLES) *On appelle assignation d'un ensemble des variables $W \subset V$ relativement à une interprétation $I$ , une application $\overline{\varphi}_I : W \to \mathbb{D}(I)$.*

EXEMPLE 6.3.1 *Considérons un langage $\mathcal{L}$ avec une constante $a$, une variable $x$, un foncteur unaire $f$ et un prédicat binaire $p$. On peut envisager la formule $p(f(a), a)$. La transformation de cette formule par l'interprétation $I$ serait $p_I(f_I(a_I), a_I)$. Mais qu'en est-il de la transformation de la formule $p(f(x), a)$ ? Si on procède à l'assignation $x_I$ de la variable $x$, où $x_I$ une variable indiquant un élément quelconque de l'univers du discours, alors on peut écrire pour l'interprétation : $p_I(f_I(x_I), a_I)$. Bien sûr si on pose $x = a$, alors les deux interprétations sont identiques.*

L'assignation des variables nous permet maintenant de définir la signification d'un terme.

DÉFINITION 6.3.3 (SÉMANTIQUE DES TERMES) *La* signification $\varphi_I$ *d'un ensemble des termes $\mathbb{T}$ relativement à une interprétation $I$ , est définie comme suit :*

- *si $t \in \mathbb{T}$ est une constante, alors $\varphi_I(t) = t_I$ ;*
- *si $t \in \mathbb{T}$ est une variable, alors $\varphi_I(t) = \overline{\varphi}_I(t)$ ;*
- *si $t \in \mathbb{T}$ est un terme de la forme $f(t_1, t_2, \ldots, t_n)$, alors $\varphi_I(t) = f_I(\varphi_I(t_1), \varphi_I(t_2), \ldots, \varphi_I(t_n))$.*

Notons que la terminologie n'est pas stabilisée en français. Certains auteurs utilisent le terme *assignation* pour signification, tandis que d'autres auteurs préfèrent parler d'*interprétation*. Il est curieux de constater que personne n'utilise la traduction du terme anglais *meaning* (= signification) qui est, à mon avis, très éclairante ici.

ASCÈSE 6.11 *Considérons un ensemble des termes $\mathbb{T}$ qui contient la constante zéro (l'élément neutre de l'addition), le foncteur unaire s (l'élément successeur) et le foncteur binaire plus (l'addition de deux valeurs). On cherche à établir une signification relativement à une interprétation $I$ dont son domaine $\mathbb{D}(I)$ est l'ensemble des entiers non négatifs muni de l'opération de l'addition $+$.*

*(1) Quelle doit-être, selon vous, l'interprétation des termes zéro , s et plus ?*

*(2) Calculer la signification du terme $plus(s(zero), X)$ où X une variable avec assignation*

$$\overline{\varphi}_I(X) = \begin{cases} 0 & si\ X \neq \ nombre\ entier \\ \\ |\,X\,| & si\ X = \ nombre\ entier \end{cases}$$

On peut maintenant, grâce aux deux définitions précédentes, calculer la valeur de vérité d'une fbf. Formellement nous avons la définition suivante :

DÉFINITION 6.3.4 (SÉMANTIQUE DES FBF) *La* valeur de vérité de la signification d'une fbf $F$ relativement à une interprétation $I$ *(ou, de façon plus succincte, la valeur de vérité de $F$ relativement à $I$), est définie comme suit :*

- *Si la fbf est de la forme $F = p(t_1, t_2, \ldots, t_n)$ , alors $I(F)$ est égale à la valeur de vérité de $p_I(\varphi_I(t_1), \varphi_I(t_2), \ldots, \varphi_I(t_n))$.*
- *Si la fbf $F$ a une des formes $\neg G$ , $G \vee H$ , $G \wedge H$ , $G \to H$ , $G \leftrightarrow H$, alors $I(F)$ est égale à la valeur de vérité de la forme correspondante.*

- *Si la fbf $F$ est de la forme $\forall x G(x,y,z,\ldots)$, alors $I(F)=1$ si $\forall\sigma=(x/a)$ avec $a\in\mathbb{D}(I)$ nous avons $I(G\sigma)=1$ (vraie). Sinon $I(F)=0$ (fausse).*
- *Si la fbf $F$ est de la forme $\exists x G(x,y,z,\ldots)$, alors $I(F)=1$ si $\exists\sigma=(x/a)$ avec $a\in\mathbb{D}(I)$ nous avons $I(G\sigma)=1$ (vraie). Sinon $I(F)=0$ (fausse).*

EXEMPLE 6.3.2 *Si $\mathbb{D}(I)$ est l'ensemble des nombres réels avec la relation d'égalité "=" qui correspond au prèdicat* eg *du langage $\mathcal{L}_1$, alors la formule atomique $F=\text{eg}(a,b)$ a comme valeur de vérité $I(F)$, la valeur de vérité de la relation $a=b$.*

De cette définition on déduit que la valeur de vérité d'une fbf close, par rapport à une interprétation, dépend seulement de cette interprétation et elle est indépendante de l'assignation des variables. Si nous avons une fbf avec des variables libres, alors on utilise la clôture universelle qui nous permet d'obtenir une fbf close et on applique ensuite la définition précédente. L'utilisation de la clôture universelle est tout à fait concevable car, comme nous venons de le dire, l'assignation des variables n'intervient pas à la détermination de la valeur de vérité d'une fbf.

ASCÈSE 6.12 *Considérons l'ascèse 6.10. Donner, pour une interprétation, la valeur de vérité de fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

### 6.3.1 Satisfiabilité et modèles

Nous donnons dans la suite la définition d'une fbf satisfiable.

DÉFINITION 6.3.5 *Une fbf $F$ est* satisfiable *ou* sémantiquement consistante *s'il existe une interprétation $I$ telle que la valeur de vérité de $F$ par rapport $I$ est ègale à* 1. *L'interprétation est alors un* modèle *de $F$ et l'on note par $I\models F$.*
*Une fbf qui ne possède pas de modèle est appelée* sémantiquement inconsistante *ou* insatisfiable.

ASCÈSE 6.13 *En reprenant l'ascèse 6.10, trouver un modèle pour la fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

Nous arrivons ainsi à la notion de la fbf valide.

DÉFINITION 6.3.6 *Une fbf $F$ qui est vraie pour toute interprétation est appelée* formule valide *et sera notée par $\models F$ .*

ASCÈSE 6.14 *Vérifier si la fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

*est une formule valide.*

Nous allons maintenant examiner brièvement la nature de deux quantificateurs $\forall$ et $\exists$. Considérons un prédicat $p$ quelconque, que nous prendrons pour la circonstance et sans perte de généralité, unaire. La fbf $\forall x p(x)$ a comme valeur de vérité pour toute interprétation d'un domaine quelconque $\mathbb{D}(I)$ la valeur $\min\{p_I/x_I\in\mathbb{D}(I)\}$, c'est-à-dire la valuation d'une fbf universellement quantifiée sur la variable $x$ est la valuation minimale de cette formule pour toutes

les interprétations $x_I$ de $x$ appliquées à l'interprétation $p_I$ du prédicat $p$. Dans la mesure où $\min\{p_I/x_I \in \mathbb{D}(I)\} = \bigwedge\{p_I/x_I \in \mathbb{D}(I)\}$ on peut dire que le quantificateur universel $\forall$ est une généralisation du connecteur $\bigwedge$. De même la valuation de la fbf $\exists x p(x)$est donnée par la valeur $\max\{p_I/x_I \in \mathbb{D}(I)\} = \bigvee\{p_I/x_I \in \mathbb{D}(I)\}$, on peut dire que le quantificateur existentiel $\exists$ est une généralisation du connecteur $\bigvee$.

Examinons maintenant la notion de la conséquence sémantique.

DÉFINITION 6.3.7 *Soit la fbf close $B$ et un ensemble des fbf closes $A = \{A_1, A_2, \ldots, A_n\}$. $B$ est une* conséquence sémantique *des $A_1, A_2, \ldots, A_n$ , que l'on note par $A \models B$ ou $A_1, A_2, \ldots, A_n \models B$, si pour toute interprétation $I$ telle que $I(A_i) = 1 \forall i = 1, \ldots, n$, on a $I(B) = 1$.*

En d'autres termes, on peut dire que $B$ doit être vérifiée pour tout modèle de $A$.

Il faut prendre dorénavant l'habitude d'interpréter un ensemble $\{A_1, A_2, \ldots, A_n\}$ de fbf comme étant une conjonction $A_1 \wedge A_2 \wedge \ldots \wedge A_n$.

ASCÈSE 6.15 *Considérons les deux fbf closes*

$$A = \{\forall X (\forall Y ((p(X) \wedge q(Y,X)) \rightarrow r(X,Y))), p(toto) \wedge q(koko, toto)\}$$

*Montrer que la fbf close*

$$r(toto, koko)$$

*est une conséquence sémantique de A. (Démonstration avec l'utilisation du modus ponens, cf. infra).*

En général il est difficile de vérifier si une fbf close $B$ est une conséquence logique d'un ensemble de fbf closes $A$, car il faut vérifier $B$ pour tout modèle de $A$. Une autre façon de démontrer que $A \models B$ est de montrer que $\neg B$ est fausse pour tout modèle de $A$ ou, ce qui revient au même, que l'ensemble de fbf $A \cup \{\neg B\}$ est insatisfiable, c'est-à-dire n'a pas de modèle. Ce procédé est plus facile parce qu'il suffit de trouver un modèle de $A$ qui n'est pas un modèle pour $B$. Formellement nous avons :

THÉORÈME 6.3.1 (DE L'INSATISFIABILITÉ) *Soient $A$ ensemble de fbf closes et $B$ fbf close. Alors $A \models B$ si et seulement si $A \cup \{\neg B\}$ est insatisfiable.*

Une notion importante pour la sémantique des fbf est celle de l'équivalence :

DÉFINITION 6.3.8 *Deux fbf $F$ et $G$ sont* logiquement équivalentes *si et seulement si elles ont la même valeur de vérité pour toute interprétation $I$.*

EXEMPLE 6.3.3 *Les formules suivantes sont logiquement équivalentes :*

- $\neg\neg A \equiv A$
- $A \rightarrow B \equiv \neg A \vee B$
- $A \rightarrow B \equiv \neg B \rightarrow \neg A$
- $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$
- *Lois de de Morgan*
  - $\neg(A \vee B) \equiv \neg A \wedge \neg B$
  - $\neg(A \wedge B) \equiv \neg A \vee \neg B$

### 6.3.2 Examen des quantificateurs

Considérons un prédicat $p$ quelconque, que nous prendrons pour la circonstance et sans perte de généralité, d'arité un. La fbf $\forall x p(x)$ a comme valeur de vérité pour toute interprétation d'un domaine quelconque $\mathbb{D}(I)$ la valeur $\min\{p_I/x_I \in \mathbb{D}(I)\}$, c'est-à-dire la valuation d'une fbf universellement quantifiée sur la variable $x$ est la valuation minimale de cette formule pour toutes les interprétations $x_I$ de $x$ appliquées à l'interprétation $p_I$ du prédicat $p$. Dans la mesure où $\min\{p_I/x_I \in \mathbb{D}(I)\} = \bigwedge\{p_I/x_I \in \mathbb{D}(I)\}$ on peut dire que le quantificateur universel $\forall$ est une généralisation du connecteur $\bigwedge$. De même la valuation de la fbf $\exists x p(x)$est donnée par la valeur $\max\{p_I/x_I \in \mathbb{D}(I)\} = \bigvee\{p_I/x_I \in \mathbb{D}(I)\}$, on peut dire que le quantificateur existentiel $\exists$ est une généralisation du connecteur $\bigvee$.

(1) $\neg\forall x A(x) \equiv \exists x \neg A(x)$

(2) $\forall x A(x) \equiv \neg\exists x \neg A(x)$

(3) $\neg\exists x A(x) \equiv \forall x \neg A(x)$

(4) $\exists x A(x) \equiv \neg\forall x \neg A(x)$

(5) $\forall x(A \vee B(x)) \equiv A \vee \forall x B(x)$ s'il n'y a pas d'occurrences de $x$ dans $A$.

Le théorème suivant fournit des formules supplémentaires.

THÉORÈME 6.3.2 *Pour la distributivité des quantificateurs nous avons les relations suivantes :*

*(1)* $\models \forall x(\varphi \wedge \psi) \leftrightarrow \forall x\varphi \wedge \forall x\psi$

*(2)* $\models \exists x(\varphi \vee \psi) \leftrightarrow \exists x\varphi \vee \exists x\psi$

*(3)* $\models \forall x(\varphi(x) \vee \psi) \leftrightarrow \forall x\varphi(x) \vee \psi$ *si x n'est pas une variable libre de* $\psi$.

*(4)* $\models \exists x(\varphi(x) \wedge \psi) \leftrightarrow \exists x\varphi(x) \wedge \psi$ *si x n'est pas une variable libre de* $\psi$.

Attention : Les formules suivantes :
– $\forall x(\varphi(x) \vee \psi(x)) \rightarrow \forall x\varphi(x) \vee \forall x\psi(x)$
– $\exists x\varphi(x) \wedge \exists x\psi(x) \rightarrow \exists x(\varphi(x) \wedge \psi(x))$
ne sont pas vraies.

## 6.4 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. Ainsi considérée, l'évaluation syntaxique est une théorie de la démonstration. Comme toute démarche déductive, la démonstartion en logique est fondée sir des axiomes qui sont les principes fondamentaux de la logique. La première notion de la théorie de démonstration est le théorème.

DÉFINITION 6.4.1 *Une fbf A est un* théorème, *et l'on note* $\vdash A$ *, si A est un axiome ou si A est une formule obtenue par application des règles d'inférence sur d'autres théorèmes.*

Notons que les règles d'inférence sont celles utilisées par le mécanisme de démonstration en logique propositionnelle, plus des règles spécifiques aux quantificateurs. Pour le calcul des prédicats nous avons donc comme règles d'inférence les suivantes :

**R1.-** Modus ponens (règle du détachement)

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

**R2.-** Modus tollens (l'inverse de modus ponens)

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

**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.-** Généralisation

$$\frac{\vdash A}{\vdash \forall x A}$$

**R6.-** Exemplification universelle (instanciation universelle)

$$\frac{\vdash \forall x A}{\vdash A\sigma}, \ \sigma = (x/a)$$

**R7.-** Exemplification existentielle (instanciation existentielle)

$$\frac{\vdash \exists x A}{\vdash A\sigma}, \ \sigma = (x/s(x))$$

où $s(\cdot)$ est une constante qui est le résultat de l'application d'une fonction $s$ à $x$.

Le calcul des prédicats possède les trois axiomes du calcul propositionnel, à savoir :

**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.-** Réalisation de contradiction

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

où, encore

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

plus deux axiomes supplémentaires :

**A4.-** Exemplification (instanciation) universelle

$$\forall x A(x) \rightarrow A(c)$$

**A5.-** Généralisation universelle

$$((A \rightarrow B) \rightarrow (A \rightarrow \forall x B))$$

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

DÉFINITION 6.4.2 *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.*

DÉFINITION 6.4.3 *Une fbf close A est une* déduction *de l'ensemble de fbf closes* $B_1, B_2, \ldots, B_n$*, 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*.

THÉORÈME 6.4.1 (DE LA DÉDUCTION) *Soit A une fbf close. Si* $A \vdash B$*, alors* $\vdash A \rightarrow B$ *et vice versa.*

On introduit maintenant la notion de la théorie.

DÉFINITION 6.4.4 *Une* théorie $\mathcal{T}$ *est une collection des théorèmes avec la propriété* $\mathcal{T} \vdash p \rightarrow p \in \mathcal{T}$.

Deux théories peuvent être en relations selon la définition suivante.

DÉFINITION 6.4.5 *Soient* $\mathcal{T}$ *et* $\mathcal{T}'$ *deux théories sur les langages* $\mathcal{L}$ *rt* $\mathcal{L}'$ *respectivement.*

- $\mathcal{T}'$ *est une* extension *de* $\mathcal{T}$ *si* $\mathcal{T} \subseteq \mathcal{T}'$
- $\mathcal{T}'$ *est une* extension conservative *de* $\mathcal{T}$ *si* $\mathcal{T}' \cap \mathcal{L} = \mathcal{T}$*, i.e. tous les théorèmes de* $\mathcal{T}'$ *dans le language* $\mathcal{L}$ *sont aussi des théorèmes de* $\mathcal{T}$.

Nous terminons ce paragraphe par la notion de la consistance.

DÉFINITION 6.4.6 *Une logique est* syntaxiquement consistante *s'il n'existe aucune formule du langage telle que nous avons en même temps* $\vdash A$ *et* $\neg \vdash A$ .

Nous avons le

THÉORÈME 6.4.2 *Le calcul des prédicats est syntaxiquement consistant.*

## 6.5 Équivalence entre modèles et théorie de démonstration

Nous allons voir que les modèles et la théorie de démonstration sont équivalentes en calcul des prédicats comme c'était déjà le cas en calcul propositionnel. Cette équivalence s'établit à l'aide de deux notions : adéquation et complétude.

DÉFINITION 6.5.1 *Une logique est* adéquate *si tout théorème $\vdash A$ est une formule valide $\models A$.*
*Une logique est* complète *si toute formule valide est un théorème, i.e.* $(\models A) \rightarrow (\vdash A)$.

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

THÉORÈME 6.5.1 *Le calcul des prédicats est adéquat, i.e.*$(\vdash A) \rightarrow (\models A)$.

THÉORÈME 6.5.2 *Le calcul des prédicats est (fortement) complet, i.e.*
$(\models A) \rightarrow (\vdash A)$.

THÉORÈME 6.5.3 (de complétude) $A \vdash p \leftrightarrow A \models p$, *pour tout fbf p avec* $p \in \mathcal{L}$.

## 6.6 Quelques méta-théorèmes

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

THÉORÈME 6.6.1 (DE LA RÉFUTATION) *Une fbf A est* conséquence valide *d'un ensemble E de fbf, i.e.* $E \vdash A$, *si et seulement si* $E \cup \{\neg A\}$ *est insatisfiable.*
*Une autre forme équivalente (et utile) de ce théoreme est la suivante : Si* $E \cup \{A\}$ *est inconsistante, alors* $E \vdash \neg A$.

*Notons qu'il faut bien interpréter le symbole* $E \cup \{A\}$. *Si E est un ensemble des fbf* $f_1, \ldots, f_n$, $E = \{f_1, f_2 \ldots, f_n\}$, *alors* $E \cup \{A\}$ *signifie que nous avons* $f_1$ ET $f_2$ ET ... ET $f_n$ ET $A$, *c'est-à-dire la virgule dans la notation de l'ensemble E joue le rôle du connecteur* $\bigwedge$. ).

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 en calcul des prédicats. Elle est fondée sur le fait que si on accepte la négation d'une fbf et on aboutit à une contradiction, alors la fbf originale est vérifiée. Notons qu'un ensemble de fbf $E$ contient une contradiction si et seulement s'il existe une fbf $A$ telle que nous avons à la fois $E \vdash A$ et $E \vdash \neg A$.

Nous donnons ci-après quelques métathéorèmes supplémentaires du calcul des prédicats.

THÉORÈME 6.6.2 (RÈGLE T) *Si* $E \vdash A_1, E \vdash A_2, \ldots, E \vdash A_n$ *et*
$\{A_1, A_2, \ldots, A_{n-1}\} \vdash B$, *alors* $E \vdash B$.

THÉORÈME 6.6.3 (DE CONTRAPOSITION) $E \cup \{A\} \vdash \neg B$ *si et seulement si* $E \cup \{B\} \vdash \neg A$.

THÉORÈME 6.6.4 (DE LA GÉNÉRALISATION) *Si* $E \vdash B$ *et x est une variable qui n'a pas d'occurrences libres dans E, alors* $E \vdash \forall x B$.

## 6.7 Formes clausales

Nous avons déjà introduit au chapitre précédent la notion de la clause. Quand on passe au calcul des prédicats il est possible qu'une clause contient des quantificateurs. Afin d'obtenir une normalisation des différents types des clauses nous introduisons la notion de la clause sous forme prénexe.

DÉFINITION 6.7.1 *Une* clause sous forme prénexe *est une clause de la forme* $\Diamond x_1, \Diamond x_2, \ldots \Diamond x_n C$, *où* $\Diamond$ *désigne un quantificateur et C est une fbf sans quantificateurs.*
*La clause vide sera notée par* $\perp$. *Il s'agit d'une fbf toujours fausse.*

Nous avons le théorème suivant :

THÉORÈME 6.7.1 *Toute fbf du calcul des prédicats admet une forme prénexe équivalente*

La démonstration de ce théorème peut se faire, de manière constructive, par l'algorithme suivant de la transformation d'une fbf en une forme prénexe équivalente :

(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) Application des substitutions d'une variable par une autre de sorte que chaque variable apparaît sous la portée d'un seul quantificateur.

(4) Suppression des quantificateurs non opérationnels, i.e. dont la variable quantifiée n'apparaît pas sous leur portée.

(5) Transfert de la négation au niveau le plus intérieur (i.e. devant les atomes) par utilisation des règles :
   - 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.
   - $\neg\forall x A \equiv \exists x \neg A$
   - $\neg\exists x A \equiv \forall x \neg A$

(6) Transfert des quantificateurs au début de la fbf en utilisant les règles :
   - $(\forall x A \wedge B) \equiv \forall x (A \wedge B)$ si $B$ ne contient pas $x$.
   - $(\exists x A \wedge B) \equiv \exists x (A \wedge B)$ si $B$ ne contient pas $x$.
   - $(\forall x A \vee B) \equiv \forall x (A \vee B)$ si $B$ ne contient pas $x$.
   - $(\exists x A \vee B) \equiv \exists x (A \vee B)$ si $B$ ne contient pas $x$.

ASCÈSE 6.16 *Calculer la forme prénexe équivalente de la fbf suivante :*

$$\forall x A(x) \wedge \exists y B(y) \rightarrow \exists y (A(y) \wedge B(y))$$

Dans une forme prénexe la présence des quantificateurs universels n'est pas significative. En effet on considère que "$\forall x\, A(x)$ *est vraie*" et "$A(x)$ *est vraie*" sont deux formules par convention équivalentes (En fait la seconde est une exemplification – instance – de la première). Par contre la présence d'un quantificateur existentiel pour une variable d'une fbf, pose le problème de la « construction » d'une telle variable.

Par exemple si nous avons $\exists x\, A(x)$, on doit pouvoir construire une valeur $c$ pour la variable $x$ qui permet que la fbf $A(c)$ soit vraie. Dans ce cas on dit que nous avons remplacé la variable $x$ par une fonction $s$ d'arité 0, c'est-à-dire par une constante. Une autre possibilité est d'avoir la variable d'un quantificateur existentielle sous la portée d'un (ou plusieurs) quantificateur(s) universel(s) relatif(s) à d'autre(s) variable(s). Ainsi considérons la fbf $\forall x \exists y\, A(y,x)$. Dans cette formule pour chaque $x$ on postule l'existence d'un $y$ tel que $A(y,x)$ soit vérifiée. Pour résoudre ce problème on doit avoir une fonction $s(\cdot)$ d'arité 1, qui permet pour chaque $x$ donné, de fabriquer un $y$. La fbf devient ainsi $\forall x\, A(s(x),x)$.

Bien évidemment on ne cherche pas à construire cette fonction $s$ ni, a fortiori, à lui donner une interprétation. On se contente de lui donner un nom. Une telle fonction s'appelle fonction de Skolem. Nous avons :

DÉFINITION 6.7.2 *Une* fonction de Skolem *est une fonction qui, dans une fbf close, remplace la variable d'un quantificateur existentiel et prend comme arguments les variables des quantificateurs universels sous la portée desquels était la variable du quantificateur existentiel.*
*Si la variable qui est replacée par une fonction de Skolem n'était pas sous la portée d'un quantificateur universel, alors l'arité de la fonction de Skolem serait nulle, c'est-à-dire la fonction de Skolem serait une constante.*

La formule qu'on obtient si on remplace une variable sous un quantificateur existentiel par une fonction de Skolem n'est pas logiquement équivalente avec la formule initiale. Seules les consistances de deux formules sont identiques, ce qui est suffisant quand, en utilisant le théorème de réfutation, on procède par preuve de l'inconsistance.

Ainsi la forme de Skolem d'une fbf close n'a que des quantificateurs universels. Ces quantificateurs peuvent être supprimés. On obtient ainsi une *exemplification* (ou une *instance*) de la forme de Skolem.

DÉFINITION 6.7.3 *Une conjonction de clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ du calcul des prédicats dans lesquels*
- *les variables sous des quantificateurs existentiels sont remplacées par des fonctions de Skolem, et*
- *les variables sous des quantificateurs universels sont omises*

*est une* forme conjonctive normale (fcn) *ou* forme standard.
*Une fcn est parfois notée sous forme ensembliste $\{C_1, C_2, \ldots, C_n\}$.*

Soit une fbf close $F$. L'algorithme pour aboutir à une forme standard 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 règles :
- 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 reécriture.
- de la loi de l'involution $\neg\neg A \equiv A$ qui permet la suppression des doubles négations.
- $\neg \forall x A \equiv \exists x \neg A$
- $\neg \exists x A \equiv \forall x \neg A$

(4) Application des substitutions d'une variable par une autre de sorte que chaque variable apparaî t sous la portée d'un seul quantificateur.

(5) Suppression des quantificateurs non opérationnels, i.e. dont la variable quantifiée n'apparaî t pas sous leur portée.

(6) Suppression des quantificateurs existentiels par application de la fonction de Skolem.

(7) Conversion en forme prénexe, i.e. transfert des quantificateurs au début de la fbf en utilisant les règles :
- $(\forall xA \wedge B) \equiv \forall x(A \wedge B)$ si $B$ ne contient pas $x$.
- $(\exists xA \wedge B) \equiv \exists x(A \wedge B)$ si $B$ ne contient pas $x$.
- $(\forall xA \vee B) \equiv \forall x(A \vee B)$ si $B$ ne contient pas $x$.
- $(\exists xA \vee B) \equiv \exists x(A \vee B)$ si $B$ ne contient pas $x$.

(8) Suppression des quantificateurs universels.

(9) Conversion en forme standard par application de la distributivité
- de $\vee$ par rapport à $\wedge$ : $A \vee (B \wedge C) = (A \vee B) \wedge (B \vee C)$
- de $\wedge$ par rapport à $\vee$ : $A \wedge (B \vee C) = (A \wedge B) \vee (B \wedge C)$

en tant que règles de réécriture.

ASCÈSE 6.17 *Appliquer l'algorithme ci-dessus à la fbf*

$$\exists x(p(X) \wedge \forall y(p(y) \rightarrow \neg q(Y,X))) \wedge \neg\exists x(p(X) \wedge \forall y(p(y) \rightarrow \neg q(Y,X)))$$

De façon analogue nous pouvons définir la *forme disjonctive normale (fdn)* : il s'agit d'une disjonction de conjonctions des littéraux.

Un cas particulier de fdn est le problème SAT de satisfaction d'une fdn. Dans la littérature on trouve les problèmes $k$-SAT qui concernent la satisfiabilité des fdn dont chaque terme a au plus $k$ littéraux. Il y a beaucoup de résultats pour $k = 2$, qui sont des problèmes de classe de complexité $P$ et pour $k = 3$ qui sont des problèmes de classe de copmlexité $NP$-complet. Notons qu'un problème est de classe $NP$ s'il peut être résolu par une machine de Turing non déterministe en temps polynômial, où NP signifie non déterministe polynômial.
Un problème $\Pi$ est *NP-difficile* si pour tout autre problème $\Pi'$ de $NP$ on a un algorithme de complexité inférieure à celle de la classe $NP$ qui transforme toute instance de $\Pi'$ en une instance de $\Pi$.
Un problème est $NP$-complet s'il est dans $NP$ et s'il est $NP$-difficile.. Une étude plus complète de ces problèmes se fait dans le cours de Décidabilité.

En utilisant les règles de de Morgan on constate que la négation d'une fcn est une fdn et vice-versa. Par conséquent ce qui vient d'être dit pour les problèmes $k$-SAT peut être transposé dans le cas des fcn. Ce qui nous intéresse ici est de savoir si un programme logique donné $E$ permet d'induire une fbf $A$, c'est-à-dire si $E \vdash A$. Pour résoudre ce problème on transforme d'abord $E$ et $A$ sous forme conjonctive normale. Ensuite nous avons deux possibilités :

(1) Soit on construit le programme $E \cup \{A\}$ et on cherche à savoir s'il est valide, c'est-à-dire si toutes les interprétations de ce programme sont des modèles pour ce programme. Il s'agit, comme nous venons de voir, d'un problème $NP$-complet.

(2) Soit on construit le programme $E \cup \{\neg A\}$ et on cherche à savoir si $E \cup \{\neg A\} \vdash \bot$ ce qui est plus facile que le précédent car il suffit de trouver une interprétation qui rend fausse la fcn $E \cup \{\neg A\}$.
Il est à noter que Prolog fonctionne selon ce principe.

Il est utile de préciser ici que le calcul de la réponse d'un programme à une question est différent de la validité de ce même programme. Ce dernier problème est *NP*-complet et il n'y a pas actuellement une méthode de test de programme qui soit pleinement satisfaisante.

## 6.8 Exercices

EXERCICE 6.1 *Considérons l'univers du discours* $\{\bigcirc, *\}$ *avec*
- *les constantes* $a, b$
- *les variables* $x, y, z$
- *la relation binaire* $f$

*Considérons une interprétation* $I$ *telle que*
- *l'interprétation des constantes est :* $a_I = \bigcirc, b_I = *$
- *l'assignation des variables est :* $\varphi_I(x) = \bigcirc, \varphi_I(y) = \bigcirc, \varphi_I(z) = *$
- *l'interprétation de la relation est :* $f_I = \{\langle \bigcirc, * \rangle, \langle *, * \rangle\}$

*Calculer la valeur de vérité des termes suivants :*

*(1)* $\neg f(b,a)$

*(2)* $f(a,b) \wedge f(b,a)$

*(3)* $f(a,b) \vee f(b,a)$

*(4)* $f(a,b) \rightarrow f(b,a)$

*(5)* $f(a,b) \leftrightarrow \neg f(b,a)$

EXERCICE 6.2 *Considérons l'ascèse 2.2 et supposons que l'ensemble de termes contient en plus le prédicat p avec interprétation*

$$p_I = \{1, 3, 5, \ldots\}$$

*Calculer la valeur de vérité de la fbf*

$$p(zéro) \wedge p(s(zéro))$$

EXERCICE 6.3 *Traduire enf fbf du calcul des prédicats les propositions suivantes :*

*(1) Les points X, Y, Z sont sur la même droite.*

*(2) Les points X, Y, Z ne sont pas sur la même droite.*

*(3) les lignes l et l′ ont un point commun unique.*

*(4) Deux droites distinctes ont un point commun unique.*

*(5) l, l′ sont deux droites parallèles.*

*(6) (5e postulat d'Euclide.) Pour toute droite L et pour tout X il existe une droite unique qui passe par X est est parallèle à L.*

EXERCICE 6.4 *Considérons deux formules closes $F$ et $G$. Montrer que $F \equiv G$ si et seulement si $\{F\} \models G \wedge \{G\} \models F$.*

EXERCICE 6.5 *Soit $\mathcal{L}$ un langage du 1er ordre. Considérons la formule*

$$\forall x \exists y p(x,y) \rightarrow \exists y \forall x p(x,y)$$

*Est-ce que cette formule peut être satisfaite pour n'importe quelle interprétation de $p(x,y)$ ?*

EXERCICE 6.6 *Soit $\mathcal{L}$ un langage du 1er ordre constitué d'un prédicat unaire $p$ et d'un prédicat binaire $r$.*

*Nous considérons les formules suivantes :*

*(1)* $\exists x \forall y \exists z ((p(x) \rightarrow r(x,y)) \wedge p(y) \wedge \neg r(x,y))$

*(2)* $\exists x \exists z (r(z,x) \rightarrow r(x,z) \rightarrow \forall y r(x,y))$

*(3)* $\forall y (\exists z \forall x r(x,z) \wedge \forall x (r(x,y) \rightarrow \neg r(x,y)))$

*(4)* $\exists x \forall y ((p(y) \rightarrow r(y,x)) \wedge (\forall t (p(t) \rightarrow r(t,y)) \rightarrow r(x,y))$

*(5)* $\forall x \forall y ((p(x) \wedge r(x,y)) \rightarrow ((p(x) \rightarrow \neg r(x,y)) \rightarrow \exists z (\neg r(z,x) \wedge \neg r(y,z))))$

*(6)* $\forall z \forall u \exists x \forall y ((r(x,y) \wedge p(u) \rightarrow (p(y) \rightarrow r(z,x)))$

*Vérifier si ces formules sont satisfaites dans le modèle suivant : Le domaine est $\mathbb{N}$, l'interprétation de $r$ est la relation d'ordre $\leq$ et celle de $p$ est le sous-ensemble des naturels pairs.*

EXERCICE 6.7 *La loi dit qu'il est interdit de posséder des armes à feu non enregistrées. Butch possède plusieurs armes à feu non enregistrées, achétées toutes chez Le Kid. Montrer que Le Kid est un hors-la-loi.*

# 7

# LOGIQUE DES PRÉDICATS ET PROGRAMMATION



COMME en Logique Computationnelle, un programme en Prolog qui n'a pas des variables est un programme dont sa sémantique [1] se limite au programme lui-même. Il n'y a donc pas de nouvelles connaissances. Si par contre nous introduisons des variables, alors il est possible d'en déduire des connaissances nouvelles. Ce point peut être vérifier facilement en se reportant à l'exemple du graphe du chapitre précédent. Si notre programme se limite à la base de données qui représente le graphe, nos connaissances se limitent aux successeurs et prédécesseurs immédiats de chaque sommet. Si nous introduisons des variables qui permettent de définir des nouveaux prédicats, alors nous pouvons avoir des connaissances supplémentaires concernant par exemple les successeurs au sens large d'un sommet donné.

La programmation en Prolog consiste essentiellement à construire des prédicats qui permettront d'inférer des connaissances nouvelles à partir des bases de données existantes. On voit ainsi que la Logique Computationnelle est un outil pour ce que, en langage branché, on appelle *forage de données – data mining* et qui est en réalité une activité aussi vieille que le monde.

1. La sémantique d'un programme $P$ sont toutes les formules atomiques closes qui peuvent être induites par $P$. En d'autre termes c'est toutes les connaissances que nous pouvons obtenir en utilisant les connaissances du programme $P$.

## 7.1 Les prédicats de base de Prolog

Pour construire nos propres prédicats nous pouvons utiliser des prédicats de Prolog. En effet SWI-Prolog a une impressionnante collection des prédicats dont vous trouverez la liste complète et leur utilisation dans le guide de référence du langage. Nous présentons dans ce paragraphe quelques prédicats très utiles avec une explication sommaire de leur signification.

Pour la présentation des prédicats, nous utiliserons les conventions suib=vantes :

- nomPredicat/n où n est l'arité du prédicat nomPredicat.
- nomPredicat(+Arg1, -Arg2, ?Arg3), où +Arg1 signifie que l'argument Arg1 est en entrée, -Arg2 signifie que l'argument Arg2 est en sortie et ?Arg3 signifie que l'argument Arg3 est en entrée-sortie.

### 7.1.1 Entrées-sorties

```
write/1 avec write(+Term)
Exemples :
write(X), write('toto')   afficher le contenu de X ou le mot << toto >>
print/1 avec print(+Term)
Exemples :
print(X), print('toto')   afficher le contenu de X ou le mot << toto >>
tab/1 avec tab(+
tab(+Quantit\'e)            affiche quantit\'e espaces blancs
nl/0                      saut d'une ligne
read(-X)                  lecture d'une valeur et stockage dans X
```

### 7.1.2 Opérations arithmétiques en Prolog

Voici une liste non exhaustive des opérateurs arithmétiques de Prolog

```
X = Y     X et Y sont les m\^emes nombres. Si une des variables n'est pas
          instaci\'ee, alors elle prend la valeur de l'autre par unification
          (et non par affectation). Remarquez que 4 = 1+3 \'echoue car 1+3
          est vu par Prolog comme un arbre +(1, 3) (faire display(4, 1+3).
          pour le voir).
X =:= Y   comme avec ''='' mais les variables doivent \^etre instanci\'ees
          au pr\'ealable. Si l'une de variables est instaci\'ee et l'autre non,
          l'op\'erateur \'echoue.)
X \= Y    X et Y sont des nombres diff\'erents. (Attention, si on pose X is 3,
          X\=Y., on obtient false.)
X =\= Y   X est non identique \'a Y.
X < Y
X =< Y
X >= Y
X > Y
X == Y    \'egalit\'e sans unification
X =/= Y   X est non identique \'a Y
```

Examinons à l'aide d'un exemple la différence entre les trois types d'égalité :

```
?- 3 = 1+2.
false.

X is 3, Y is 1+2, X=Y.
X = 3,
```

```
Y = 3.

?- X is 1+2, Y = X.
X = 3,
Y = 3.

?- X is 1+2, Y =:=X.
ERROR: =:=/2: Arguments are not sufficiently instantiated

?- X is 1+2, Y = 3, X =:=Y.
X = 3,
Y = 3.

?- X is 1+2, Y = 3, X =\=Y.
false.
```

Dans le premier cas, la représentation par Prolog de deux membres de l'égalité n'est pas identique et on donc échec. Dans le deuxième cas, le contenu de deux variables est identique et le test réussit. Dans le troisième cas, Y est instancié à la valeur de X et le test réussi. Notons que ce test réussira aussi si les deux variables ne sont pas instanciées. Dans le quatrième cas le contenu Y n'a pas de valeur et elle ne peut pas être instanciée avec X. Donc le test échoue. Dans le cinquième cas le contenu de deux variables est le même et le test réussit et par la suite le test du sixième cas échoue.

### 7.1.3 Fonctions arithmétiques

Il y a les quatre opérations arithmétiques – `+, -, *, /` – ainsi que les fonctions `sqrt, abs, sign, max, min`. Nous avons aussi les fonctions trigonométriques `sin, cos, tan asin, acos, atan` et le s fonctions logarithmiques `log10, log, exp`. Notons aussi les constantes `pi = 3.14159` et `e = 2.71828`.

Notons encore trois procédures Prolog, à savoir

- `between(+L1, +L2, ?Val)` qui réussit si `L1 =< L2`. `Val` est instaciée successivement aux valeurs `L1, ...L2`.
- `succ(?I1, ?I2)` qui réussit si `I2 = I1+1`. Au moins un argument doit être instacié.
- `plus(?I1, ?I2, ?I3)` si `I3 = I1 + I2`. Au moins deux arguments doivent être instaciés.

### 7.1.4 Opérateurs avec des termes non arithmétiques

Voici une liste non exhaustive d'opérateurs qui s'appliquent à des termes non arithmétiques.

```
:-          si
X = Y       X et Y sont le m\^eme contenu. Si une des variables n'est pas
            instaci\'ee, alors elle prend la valeur de l'autre par unification
            (et non par affectation)
X == Y      egalite sans unification
X \= Y      X et Y ont de valeurs diff\'erentes
X \== Y     X est non identique \'a Y

            En examinant le code ASCII du contenu de X et Y, on a
            les op\'erateurs suivants
X @< Y      X est plus petit que Y
X @=< Y     X est plus petit ou \'egal \'a Y
X @> Y      X est plus grand que Y
```

```
X @>= Y   X est plus grand ou \'egal \'a Y
X @== Y   X est structurellement identique \'a Y
X \=@= Y  X n'est pas structurellement identique \'a Y
```

L'équivalence structurelle `@==/2` est plus faible que l'équivalence `==/2` mais plus forte que l'unification `=/2` . Ainsi on a

```
a =@= A              faux
A =@= B              vrai
x(A,A) =@= x(B,C)    faux
x(A,A) =@= x(B,B)    vrai
x(A,B) =@= x(C,D)    vrai
```

#### 7.1.4.1 Opérateurs établis par l'utilisateur

En dehors des opérateurs établis par `Prolog` le programmeur peut définir ses propres opérateurs en utilisant la requête « `:-` ». La forme générale d'un opérateur est

```
:- op(priorité, type, nom)
```

avec

- `Priorité` : une valeur entre 1 et 32000 qui indique la priorité de l'opérateur (1 est le plus prioritaire, 32000 le moins prioritaire).
- `Type` :
  - infixe : `xfx, xfy, yfx, yfy`
  - préfixe : `fx, fy`
  - postfixe : `xf, yf`
- `Nom` : le nom de l'opérateur.

En ce qui concerne le type de l'opérateur, le symbole « `x` » interdit les associations tandis que le symbole « `y` » les autorise. Ainsi les quatre opérations numériques sont du type `yfx` et par conséquent une expression comme

```
a + b + c + d
```

sera représentée en interne comme suit

```
(((a + b) + c) + d).
```

La virgule est un opérateur du type `xfy` et donc l'expression

```
a, b, c, d
```

sera interprétée comme suit :

```
(a, (b, (c, d)))
```

### 7.1.5 Prédicats extralogiques

Les prédicats extralogiques sont des prédicats qui soit testent le statut d'un terme, soit induisent une action.

Dans la première catégorie on trouve

```
var(+X)     X est une variable
nonvar(+X)  X n'est pas une variable
atom(+X)    X est une constante non num\'erique
atomic(+X)  X est une constante non num\'erique ou une valeur num\'erique
integer(+X) X est un entier
float(+X)   X ets un flottant
number(+X)  X est un nombre (entier ou flottant)
```

Pour `atom` nous avons les résultats suivants :

```
?- atom('toto').
true.

?- atom(toto).
true.

?- atom(3).
false.

?- atom(pi).
true.

?- X is pi, atom(X).
false.
```

Pour les nombres on a les résultats suivants :

```
atomic(pi).
true.

?- float(pi).
false.

?- X is pi, atomic(X).
X = 3.14159.

?- X is pi, float(X).
X = 3.14159.
```

Dans la deuxième catégorie on trouve d'une part

```
assert(X)   Ajouter \'a la base de donn\'ees le fait X
asserta(X)  Ajouter au debut de la base de donnees le fait X
assertz(X)  Ajouter \'a la fin de la base de donnees le fait X
retract(X)  Supprimer de la base de donn\'ees toutes les occurrences de X
```

et, d'autre part, le prédicat `fail` et le coup-choix (cut) `!` que nous examinerons en détail plus loin.

## 7.2 Un exemple

Considérons de nouveau l'exemple du graphe du chapitre précédent dont voici de nouveau sa représentation graphique :

Nous avons vu que la base de données du graphe permet de répondre à de questions du type : `?gr(a,b,_)`. Pour améliorer nos connaissances on doit utiliser la logique des prédicats qui, grâce à l'existence des quantificateurs, permet de poser des questions du type

`Existe-t-il des sommets X tels que gr(a,X,_) ?`

et qui sous forme clausale s'écrit

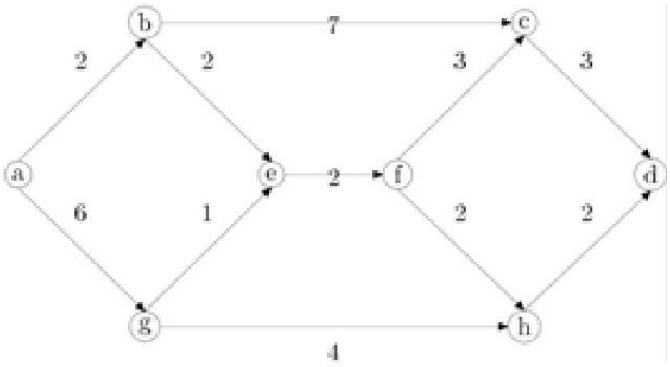



FIGURE 7.1 – Graphe pour un réseau

`?-gr(a,X,_).` [2]

Dans le cadre de la logique des prédicats, les clauses du programme sont considérées comme des prédicats et, par conséquent, leur valeur de vérité dépend de la valeur des arguments.

PROGRAMME 7.2.1

```
successeurImmediat(X,Y) :- gr(X,Y,_).
```

ASCÈSE 7.1 *Examiner le résultat des questions :*

`successeurImmediat(c,Y).`
`successeurImmediat(X,Y).`
`successeurImmediat(X,c).`

*Pourriez-vous faire la liaison avec le modèle minimal de Herbrand ?*

*Pourriez-vous anticiper la réponse de* `Prolog` *aux questions :*

`successeurImmediat(X,a).`
`successeurImmediat(d,Y).`

*En fonction des résultats obtenus est-il possible d'établir deux nouveaux prédicats qui décrivent les propriétés des sommets* `a` *et* `b` *?*

Nous pouvons maintenant envisager de définir la notion du successeur au sens large, à savoir un sommet qu'on peut atteindre d'un sommet fixé en passant par plusieurs sommets intermédiaires. Cette notion peut introduite au programme en y ajoutant les clauses suivantes :

PROGRAMME 7.2.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
```

Notons que ce programme est sous forme recursive parce que la deuxième clause contient un appel à cette même clause. Nous examinerons par la suite les méthodes d'écriture des programmes récursifs.

2. Remarquons que la clause écrite en `Prolog` respecte la convention qui veut que les noms des constantes commencent par une lettre minuscule – ici le sommet `a` – et les noms des variables par une lettre majuscule – ici `X` qui représente les sommets reliés directement à `a` et dont `a` est l'extrémité de départ. De plus on utilise aussi le symbole « _ » qui représente la variable anonyme, c'est-à-dire une variable dont la valeur précise ne nous intéresse pas. Pour plus de détails concernant la variable anonyme, voir dans la suite.

ASCÈSE 7.2 *Examiner le résultat des questions :*
`successeur(c,Y).`
`successeur(X,c).`
*Pourriez-vous faire la liaison avec le modèle minimal de Herbrand ?*
*Établir l'arbre de dérivation SLD pour les questions précédentes.*

Nous allons par la suite revenir plusieurs fois sur cet exemple du graphe.

## 7.3 Sémantique de Prolog

Considérons un programme défini $E$ écrit en Prolog. Sa sémantique est l'ensemble des fbf closes que nous pouvons en déduire des clauses du programme $E$ en appliquant les règles de la logique des prédicats. Ainsi une question – qui, en réalité, est une clause – que nous posons, constitue un but à vérifier par $E$ et si ce but est dans la sémantique de $E$, alors la réponse du programme sera positive.

Remarquons que le comportement d'un programme Prolog depend de l'ordre de clauses – faits et règles – et de l'ordre des prédicats, à l'intérieur de chaque clause. Du fait que, en cherchant à répondre à une question, Prolog examine ces prédicats en les considérant comme des buts, dans la bibliographie l'ordre des prédicats porte le nom d'*ordre des buts*, nom que nous utiliserons par la suite.

### 7.3.1 Ordre des clauses

L'ordre dans lequel sont écrites les clauses d'un programme Prolog détermine l'ordre dans lequel seront trouvées les différentes solutions. En effet Prolog pour trouver toutes les réponses à un but, parcourt l'arbre de dérivation de la résolution SLD selon l'algorithme *en profondeur d'abord*. Chaque clause du programme est placée sur une branche de cet arbre en fonction de la place qu'elle occupe dans l'ordre des clauses du programme. Si donc nous avons deux programmes qui sont identiques mais dans lesquels l'ordre des clauses n'est pas le même, le parcours de l'arbre ne se fera pas de la même façon mais les deux graphes seront isomorphes et par conséquent si l'un de deux n'a pas une branche infinie, l'autre non plus n'a pas de branche infinie.

Il n'y a pas une règle générale pour l'ordre des clauses dans un programme Prolog. L'usage cependant veut que, dans le cas d'un programme récursif, on ait d'abord le(s) fait(s), c'est-à-dire le(s) test(s) d'arrêt, et ensuite les règles récursives.

ASCÈSE 7.3 *Vérifier l'ordre des solutions trouvées pour le but*
`?-successeur(f,X).`
*si à la place du programme 1.3 on utilise le programme*

PROGRAMME 7.3.1

```
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
successeur(X,Y) :- successeurImmediat(X,Y).
```

*Pourriez-vous expliquer les différences ?*

### 7.3.2 Ordre des buts

L'ordre dans lequel sont présentés les premisses ou (sous-)buts dans une règle d'un programme `Prolog` conditionne l'exécution du programme et, donc, il détermine les solutions qui seront trouvées. Car, contrairement à la permutation des clauses qui aboutit à des arbres de dérivation toujours isomorphes entre eux, la permutation des buts donne naissance à des arbres de dérivation différents. De plus en fonction de la place qu'il occupe un appel récursif dans une règle, le programme peut nous fournir des solutions avant d'aboutir à une branche infinie ou même aboutir à cette branche infinie avant de donner la moindre solution.

ASCÈSE 7.4 *Vérifier les solutions trouvées pour le but*
`?-successeur(f,X).`
*si à la place du programme 1.3 on utilise le programme*

PROGRAMME 7.3.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeur(Z,Y), successeurImmediat(X,Z).
```

*Pourriez-vous expliquer les différences ?*
*Pourriez-vous anticiper la réponse aux questions*
`?-successeur(d,X).` *et*
`?-successeur(X,a).`

Comme précédemment avec les clauses, il n'y a pas non plus une méthode pour établir l'ordre des buts dans une règle. L'ascèse ci-dessus suggère d'utiliser un appel récursif à la fin des prémisses d'une règle (récursivité droite) plutôt qu'un appel récursif au début des prémisses (récursivité gauche) mais il ne s'agit pas d'une méthode générale.

ASCÈSE 7.5 *On dira que deux sommets sont au même niveau s'ils sont successeurs immédiats du même sommet.*

(1) *Écrire le programme* `memeNiveau(X,Y).`

(2) *Vérifier en posant la question* `?memeNiveau(b,X)` *que deux variables différentes n'ont pas nécessairement des valeurs différentes. Apporter une solution.*

(3) *Compléter ce programme pour tenir compte du fait que la relation* `memeNiveau(X,Y)` *est symétrique, c'est-à-dire que si, par exemple, on a* `memeNiveau(bmg)`*, alors on a aussi* `memeNiveau(g,b).`

# 8

# LISTES ET RECURSIVITÉ



NOUS avons vu jusqu'ici le stockage des données à l'aide des base de données. Mais le traitement des données à partir de ces bases n'est pas toujours très facile à effectuer. On a envie d'avoir des données en mémoire dynamique facilement manipulables. À vrai dire `Prolog` est très chichement doté en types des données. Comme son grand ancêtre, le `Lisp`, `Prolog` ne dispose comme type des données, en tout et pour tout, que les listes. Bien sûr tout programmeur experimenté sait que la profusion des types de données qui sont l'apanage des plusieurs langages de programmation, en commençant par le plus illustre, le `Fortran`, sont très souvent source de confusions. Il est donc important pour l'élève–ingénieur de comprendre que l'esprit humain doit dompter la machine, qui par construction et par essence est bête, et arriver à faire des choses merveilleuses en utilisant très peu des matériaux. `Prolog` constitue un excellent exercice pour cet objectif.

## 8.1 Les listes et leur représentation

La seule structure des données que `Prolog` reconnaît ce sont les *listes*. Ce qui veut dire qu'en `Prolog` il n'y a pas des tableaux et surtout il n'y a pas des indices de tableau ou des pointeurs.

Une liste est une suite des termes de n'importe quelle nature, separés par des virgules, entourée par deux crochets, [ et ] . Par exemple `[toto, 1, av_du_parc, cergy, la_logique_est_super]`. Bien évidemment une liste peut avoir des sous-listes, une sous-liste peut avoir des sous-sous-listes et ainsi de suite à la manière des poupées russes mais généralisées en ce sens qu'une pou-

pée peut contenir plusieurs sous-poupées de même taille et qui, à leur tour, puissent avoir plusieurs sous-sous-poupées de même taille. Par exemple `[toto,[ 1,[ av_du_parc], [cergy]], la_logique_est_super]`.

Il faut peut être le repéter : on ne peut pas accéder directement à un élément quelconque d'une liste. (Par contre on peut accéder à un élément dont on connaît effectivement son rang dans la liste.) On peut seulement séparer ce qu'on appelle la *tête* d'une liste du *reste* de la liste, en utilisant le symbole du séparateur « | ». Ainsi, si une liste est représentée par la variable `L`, on peut écrire `L=[Tete | Reste]` où `Tete` représente le premier élément de `L` et `Reste` est la liste composée des autres élément de `L`. Par exemple si
`L=[toto, 1, av_du_parc, cergy, la_logique_est_super],`
alors on a `Tete = toto`
et `Reste = [1, av_du_parc, cergy, la_logique_est_super]`.

De ce qui précède on peut en conclure qu'on peut accéder au premier élément d'une liste. Considérons maintenant une liste ayant $n$ éléments `L=[`$x_1,x_2,...,x_n$`]`. Si on veut accéder au $k$-ième terme, où $k$ a une valeur précise et connue, nous avons deux possibilités :

- soit directement en posant pour `L` : `[ T`$_1$`,T`$_2$`,...,T`$_k$ `| Reste ]` avec `T`$_k$ $\longleftarrow x_k$.
- soit d'une façon séquentielle, à la manière de la lecture des enregistrements d'un fichier en accés séquentiel.On construit la représentation `L`$_1$ `= [Tete | Reste]`, où `Tete` $\longleftarrow x_1$. On recupère le `Reste` dans une liste notée $L_2$ et on recommence : $L_2$ =`[Tete | Reste]` avec maintenant `Tete` $\longleftarrow x_2$. En continuant ainsi on arrive, au bout de $k$ itérations, à accéder au $k$-ième élément de la liste.

Même si la première possibilité vous paraît plus facile, rappelez-vous ce qu'on vous a toujours dit concernant les apparences et concentrez-vous sur la deuxième possibilité. C'est celle qui est utilisée en `Prolog` mais dans sa version recursive.

## 8.2 La recursivité

Pour appliquer la recursivité sur les listes il faut savoir que si on introduit une liste, e.g. `[toto, 1, av_du_parc, cergy, la_logique_est_super]`, `Prolog` introduit toujours à la fin de la liste, comme un élément supplémentaire, une liste vide, de sorte qu'on ait `[toto,1,av_du_parc,cergy, la_logique_est_super, []]`. Ainsi quand on progresse à l'intérieur d'une liste, élément par élément, on peut comprendre qu'on est arrivé à la fin de la liste en comparant la liste qui reste chaque fois avec la liste vide. Le test de la liste vide constituera pour beaucoup de programmes recursifs, le test d'arrêt.

En règle générale, un programme récursif est composé de deux parties :

- Une partie concernant le ou les tests d'arrêt.
- Une partie concernant les appels récursifs du prédicat à lui-même.

La programmation recursive est un type particulier de programmation au même titre que la programmation fonctionnelle ou la programmation orienté objet. Il faut savoir que la mise en øeuvre d'un programme est plus facile qu'avec les autres types de programmation et ses résultats sont beaucoup beaucoup plus spectaculaires parce qu'on utilise la puissance de la recursivité. En effet les programmes en `Prolog` expriment seulement ce que le programmeur souhaite réaliser et non pas la manière de faire pour le réaliser comme c'est le cas avec les langages imératifs.

Nous allons examiner en detail le mécanisme des appels récursifs en utilisant la liste `L=[toto, 1, av_du_parc, cergy, la_logique_est_super]`. On cherche à calculer la longueur de cette liste, c'est-à-dire le nombre d'éléments qui la composent. On va donc procéder élément par élément et chaque fois on prendra en compte le premier élément de la liste. Il faut donc pouvoir accéder au premier élément de la liste. Pour accéder au second élément, il faut supprimer de la liste le premier élément et appeler le programme de façon recursive. On a donc le programme :

```
longueur([X | Y],N) :- longueur(Y,N1), N is N1+1.
```

L'appel `longueur(Y,N1)` est un appel recursif. Ce qui signifie qu'avant la réalisation du test d'arrêt : `longueur([], 0)`, les demandes de `N is N1+1` sont empilées et ne sont pas exécutées. Elles vont commencer à être exécutées, et dans l'ordre inverse de leur empilement, après l'exécution du test d'arrêt. Concrètement si on applique ce programme à la liste `L` nous aurons le déroulement suivant :

1er appel : `longueur([toto | 1, av_du_parc, cergy, la_logique_est_super],N)`

- État du programme :
  `longueur([ 1, av_du_parc, cergy, la_logique_est_super ],N1)`
- État de la pile :

| N is N1 + 1. |
|---|

2e appel : `longueur([ 1 | av_du_parc, cergy, la_logique_est_super ],N1)`

- État du programme :
  `longueur([ av_du_parc, cergy, la_logique_est_super],N2)`
- État de la pile :

| N1 is N2 + 1. |
|---|
| N is N1 + 1. |

3e appel : `longueur([ av_du_parc | cergy, la_logique_est_super],N2)`

- État du programme :
  `longueur([ cergy, la_logique_est_super],N3)`
- État de la pile :

| N2 is N3 + 1. |
|---|
| N1 is N2 + 1. |
| N is N1 + 1. |

4e appel : `longueur([ cergy | la_logique_est_super],N3)`

- État du programme :
  `longueur([la_logique_est_super],N4)`
- État de la pile :

| N3 is N4 + 1. |
|---|
| N2 is N3 + 1. |
| N1 is N2 + 1. |
| N is N1 + 1. |

5e appel : `longueur([la_logique_est_super], | []], N4)`

- État du programme :
  `longueur([], N5)`
- État de la pile :

| N4 is N5 + 1. |
|---|
| N3 is N4 + 1. |
| N2 is N3 + 1. |
| N1 is N2 + 1. |
| N is N1 + 1. |

6e appel : `longueur([],N5)`

- État du programme :

  `N5 ⟵ 0`

- État de la pile :

| `N4 is N5 + 1. N3 is N4 + 1.` |
|---|
| `N2 is N3 + 1.` |
| `N1 is N2 + 1.` |
| `N is N1 +1.` |

Le test d'arrêt sert ici comme initialisation de la valeur de la longueur à 0. Les ordres successifs d'addition de la valeur 1 aux différents valeurs de `N` n'ont pas été exécutés mais seulement stockés dans la pile. Dès que le programme a rencontré un test d'arrêt, les ordres stockés dans la pile commencent à être exécutés. Ainsi la suite du programme se fera par de depilement successifs et exécution des commandes depilées. Nous avons donc :

7e étape : `N5 = 0`

- État du programme :

  `N4 ⟵ N5 +1 = 0+1 = 1`

- État de la pile :

| `N3 is N4 + 1.` |
|---|
| `N2 is N3 + 1.` |
| `N1 is N2 + 1.` |
| `N is N1 + 1.` |

8e étape : `N4 = 1`

- État du programme :

  `N3 ⟵ N4 +1 = 1+1 = 2`

- État de la pile :

| `N2 is N3 + 1.` |
|---|
| `N1 is N2 + 1.` |
| `N is N1 + 1.` |

9e étape : `N3 = 2`

- État du programme :

  `N2 ⟵ N3 +1 = 2+1 = 3`

- État de la pile :

| `N1 is N2 + 1.` |
|---|
| `N is N1 + 1.` |

10e étape : `N2 = 3`

- État du programme :

  `N1 ⟵ N2 +1 = 3+1 = 4`

- État de la pile :

| `N is N1 + 1.` |
|---|

11e étape : `N1 = 4`

- État du programme :

  `N ⟵ N1 +1 = 4+1 = 5`

- État de la pile :

| `< pile vide >` |
|---|

En examinant le déroulement du programme, on constate que le test d'arrêt est le dernier a être évoqué lors des appels récursifs mais le premier a être exécuté lors du depilement des ordres

empilés. Par conséquent il faut considérer les tests d'arrêt comme la partie du programme qui initialisent les valeurs des variables utilisées.

## 8.3 Techniques de recursivité

Nous présentons ci-après les techniques de base qui permettent de réaliser des programmes recursifs en `Prolog`.

### 8.3.1 Recursivité pour les fonctions numériques

Bien qu'en `Prolog` nous avons seulement des prédicats, nous pouvons envisager d'avoir des fonctions si leur résultat est stocké dans une variable qui fait partie des arguments de la fonction. Ainsi, par exemple, on peut envisager d'écrire un prédicat qui sera en réalité une fonction qui calcule la somme de $N$ premiers nombres naturels. Ce prédicat peut avoir la forme suivante :

PROGRAMME 8.3.1

```
somme(0,0).

somme(N,Somme) :- N > 0, N1 is N - 1, somme(N1, Somme1),
                                 Somme is Somme1 + N.
```

On constate donc que pour programmer une fonction numérique sous forme récursive, il faut

(1) *Déterminer le(s) test(s) d'arrêt*, c'est-à-dire décider quand la fonction retourne une valeur prédéterminée, sans appel recursif à elle-même.

Le test d'arrêt pour une fonction numérique se fait en comparant la valeur d'une variable, dont son contenu évolue en fonction des appels recursifs avec une valeur fixée par le programme. La valeur prédéterminée que le programme retourne est la valeur d'initialisation de la variable dont nous venons de parler.

(2) *Déterminer le(s) cas recursif(s).* Un appel recursif d'une fonction s'effectue avec un argument plus simple et on utilise le résultat pour calculer la réponse de l'argument courant. Un argument plus simple en recursivité numérique est un argument qui est plus proche de la valeur utilisée pour l'initialisation par le test d'arrêt.

ASCÈSE 8.1 *Calcul du factoriel $n!$.*

ASCÈSE 8.2 *Calcul de la puissance d'un nombre $x^n$.*

ASCÈSE 8.3 *Calcul des nombres de Fibonacci.*

ASCÈSE 8.4 *Calcul du plus grand diviseur commun et du plus petit commun multiplicateur de deux entiers.*

### 8.3.2 Recursivité simple

Ce cas concerne les listes qui n'ont pas des sous-listes ou même si elles en ont, on ne les prendra pas en considération.

Si nous avons à faire un traitement sur un élément quelconque, il faut l'avoir soit stocké au préalable dans une base de données, soit introduit dans une liste. Dans ce dernier cas et étant donné que nous ne pouvons pas indexer les listes par des pointeurs, on est obligé de parcourir la liste jusqu'à arriver à atteindre l'élément voulu. Ce parcours se fera par des appels recursifs où un des arguments sera la liste qui, à chaque appel, sera systématiquement debarassée de son premier élément. La technique donc de la programmation recursive pour les listes avec arrêt simple est identique à celle pour les fonctions numériques, mais les tests d'arrêt se font sur les listes et leurs éléments et non pas sur la valeur d'une variable. On distingue trois types de tests d'arrêt que nous présentons séparement ci-après.

#### 8.3.2.1 Arrêt lorsque la liste est vide

Le programme s'arrête lorsque la liste que nous sommes en train de traiter devient vide. Les programmes que nous pouvons mettre sous ce type sont

- soit des programmes d'énumération : nombre d'éléments qu'une liste contient, nombre d'occurences d'un élément dans une liste, etc.
- soit des programmes d'affichage du contenu d'une liste ou de copie d'une liste dans une autre liste ou, encore, la concaténation de deux listes.

Nous donnons comme exemple le calcul de la longueur d'une liste

PROGRAMME 8.3.2

```
longueur([], 0).

longueur([Tete | Reste], Longueur) :- longueur(Reste,Longueur1),
                                     Longueur is Longueur1 + 1.
```

Longueur is Longueur1 + 1.

ASCÈSE 8.5 *Afficher le contenu d'une liste.*

ASCÈSE 8.6 *Copier une liste dans une nouvelle liste.*

ASCÈSE 8.7 *Concatener deux listes en créant une troisième.*

#### 8.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé

On s'arrête dès qu'un élément spécifique, fixé au préalable a été retrouvé. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur le fait qu'élément particulier appartient à une liste comme, par exemple, le programme `membre`.

PROGRAMME 8.3.3

```
membre(X, [X | _]).
membre(X, [Tete | Reste]) :- membre(X, Reste).
```

Bien sûr un tel programme pose le problème de sa fin dans le cas où l'élément spécifique ne fait pas partie de la liste. Normalement dans ce cas le programme s'arrête en échec. On peut éviter cette sortie en échec, en ajoutant la clause :

```
membre(_, []).
```

Mais en procédant ainsi, on ignore si on a trouvé ou non l'élément spécifique. Si on s'inspirait de la programmation impérative, on serait tenté ici d'ajouter un indicateur qui nous informererait sur le résultat effectif. Mais la bonne solution en Prolog est de distinguer le traitement de deux situations en utilisant l'opérateur " `ou` " comme suit :

```
toto :- (membre(a, [b,a,c]),
          write('L \'el\'ement a fait partie de la liste'));
        (write ('L \'el\'ement a ne fait pas partie de la liste')), nl.
```

Si le traitement particulier dans chaque cas est long, on peut envisager d'utiliser deux clauses qui s'excluent mutuellement :

```
toto :- membre(a, [b,a,c]),
        write('L \'el\'ement a fait partie de la liste')), nl.
toto :- not(membre(a, [b,a,c])),
        write ('L \'el\'ement a ne fait pas partie de la liste')), nl.
```

ASCÈSE 8.8 *Donner la position d'un élément dans une liste.*

ASCÈSE 8.9 *Supprimer un élément d'une liste et obtenir une nouvelle liste sans cet élément.*

ASCÈSE 8.10 *Remplacer dans une liste un élément par un autre et obtenir une nouvelle liste.*

ASCÈSE 8.11 *Insérer dans une liste, après un élément spécifique, un autre élément et obtenir une nouvelle liste.*

#### 8.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte

On s'arrête dès qu'une position spécifique, fixée au préalable a été retrouvée. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur l'élément qui occupe une place particulière dans une liste comme par exemple le programme suivant qui affiche le *n*-ième élément d'une liste

PROGRAMME 8.3.4

```
afficheNth(1, [X|_]) :- write(X), nl.

afficheNth(N, [Tete | Reste]) :- N > 1, N1 is N - 1,
                                 afficheNth(N1, Reste).
```

ASCÈSE 8.12 *Supprimer le n-ième élément d'une liste et obtenir une nouvelle liste.*

ASCÈSE 8.13 *Remplacer dans une liste un élément dans une position donnée par un autre et obtenir une nouvelle liste.*

ASCÈSE 8.14 *Insérer dans une liste, avant un élément qui se trouve à une position spécifique, un autre élément et obtenir une nouvelle liste.*

### 8.3.3 Recursivité multiple

On doit travailler avec des récursivités multiples si la liste que nous sommes en train de traiter contient des sous-listes. Si e.g. on cherche à savoir si un élément donné fait partie d'une liste et si cette liste contient des sous-listes, on doit examiner séparement ses sous-listes.

La programmation des tests d'arrêt pour la recursivité multiple est identique à celle de la programmation simple. Par contre la programmation des cas recursifs est différente. On utilisera la technique de *recursivité Tête – Reste* dont nous présentons ci-après un exposé succint.

Programmation selon la technique de récursivité Tête – Reste. On suppose que nous avons une fonction qui a comme argument une liste représentée par `[Tete | Reste]` et que nous allons appeler cette fonction de façon recursive en modifiant l'argument représenté par la liste.

(1) Déterminer le(s) cas de reste – recursivité.

Ce sont les cas où la tête `Tete` de la liste est un atome et nous appelons recursivement la fonction avec comme argument pour la liste son `Reste`.

Il y a deux type de reste – recursivité.

(a) On retourne simplement les résultats de la fonction appelée récursivement sur le reste de la liste.

(b) On associe les résultats de reste – recursivité avec une valeur dérivée de la tête de la liste.

(2) Déterminer les cas de tête – reste recursivité.

Ce sont les cas où la tête `Tete` de la liste est une liste. Dans ce cas il faut appeler recursivement la fonction avec comme argument la tête de la liste et, ensuite, appeler recursivement la fonction avec comme argument le reste de la liste. À la fin il faut associer les résultats de deux appels, pour obtenir le résultat correct pour la liste entière.

On présente comme exemple d'application de la recursivité Tête – Reste la recherche d'un élément dans une liste contenant des sous-listes.

PROGRAMME 8.3.5

```
membreT(X, [X | _]).   % Test d'arr\^et

membreT(X, [Tete | Reste]) :- atom(Tete),
                              membreT(X, Reste). % Reste - recursivit\'e

membreT(X, [Tete | Reste]) :- not(atom(Tete)),
                              (membreT(X,Tete);
                              membreT(X, Reste)).% Test - reste recursivit\'e
```

On utilise aussi la recursivité multiple dans des cas où on doit traiter en même temps plusieurs listes. Examinons, à titre d'exemple, le programme qui opère un tri d'une liste numérique selon ses valeurs ascendantes .

PROGRAMME 8.3.6

```
tri([X|Xs],Y) :- partition(Xs,X,Petits,Grands),
                 tri(Petits,Ps),
                 tri(Grands,Gs),
                 conc(Ps,[X|Gs],Y).
```

```
tri([],[]).

partition([X|Xs],Y,[X|Petits],Grands) :- X < Y,
                                    partition(Xs,Y,Petits,Grands).

partition([X|Xs],Y,Petits,[X|Grands]) :- X >= Y,
                                    partition(Xs,Y,Petits,Grands).

partition([],Y,[],[]).
```

Bien évidement on n'utilise pas la technique de tête – reste recursivité mais le programme `tri` est appelé deux fois, comme en recursivité simple, avec deux listes différentes.

ASCÈSE 8.15 *Étant donnée une liste qui contient des sous-listes, obtenir un nouvelle liste qui a les mêmes éléments que la première liste mais sans sous-listes.*

ASCÈSE 8.16 *Faire un tri selon l'ordre alphabétique d'une liste qui contient des noms.*

## 8.4 Exercices

EXERCICE 8.1 *Structurer une base de connaissances de sorte que si nous avons comme faits qu'un cheval est plus rapide qu'un chien qui, à son tour, est plus rapide qu'un lapin, alors on peut répondre, pour des animaux particuliers, lequel est le plus rapide.*

EXERCICE 8.2 *Disséquer une liste.*

*Application :* `disseq([b,a,c])?; disseq([])?; disseq([b, [a, c]])?`

# 9

# TECHNIQUES DE PROGRAMMATION EN PROLOG



Ce chapitre est consacré à l'étude des techniques de programmation qui sont particulières en Prolog.

En premier lieu on présente le mapping, qui permet, à partir d'une liste, d'obtenir une nouvelle liste ayant des caractéristiques particulières. Puis les opérations avec des listes numériques et les matrices sont examinées comme une extension avec du mapping. Ensuite on présente la notion de l'accumulateur qui permet d'exécuter une opération et stocker le résultat dans l'accumulateur avant l'appel récursif du prédicat. Nous abordons par la suite la différence de listes qui est une technique de représentation des données unique en Prolog. Vient après l'étude d'un élément particulier de la programmation en `Prolog` et qui est utilisé pour le contrôle dans un programme. La coupure, notée par « ! » en `Prolog`, sert à réduire l'espace de recherche de solutions par la stratégie utilisée par `Prolog`. Nous rappelons que la stratégie de résolution utilisée consiste à développer, pour un but donné, un arbre de recherche – dit *arbre de résolution SLD* – dont la racine est le but. Cet arbre est exploré selon la méthode « en profondeur d'abord ». L'utilisation de la coupure dans une clause sert à élaguer une partie des branches de l'arbre dont la clause est la racine. Nous étudions aussi l'utilisation de la négation – `not` en `Prolog` – dans le corps d'une clause. Cette utilisation permet de décrire des situations de succés qui seront condi-

tionnés par des situations d'échec d'autres prédicats. Nous terminons On termine en présentant des prédicats concernant les ensembles et le traitement des structures.

## 9.1 Mapping

Le *mapping* est une version attenuée de la notion de relation en ce sens que le mapping est une relation unidirectionelle, là oùu la relation peut être bidirectionelle. En effet soient deux ensembles $E$ et $F$. $r$ est une relation entre $E$ et $F$ si $\forall X \in E$ et $Y \in F$ tel que $r(X,Y)$ est soit vraie, soit fausse. Par exemple on a $E,F \subset \mathbb{N}$ et $r$ est la relation "$\geq$". Par contre un mapping est une fonction $f$ entre $E$ et $F$ telle que $\forall X \in E \; \exists \, Y = f(X) \in F$. Par exemple $E \subset \mathbb{R}$ ,$F = [-1,1]$ et le mapping est la fonction $\cos(X)$.

Le mapping est utilisé en Prolog pour transformer une liste à une autre liste suivant une règle donnée.

On présente dans la suite les principaux types de mapping.

### Mapping complet

La totalité de la liste est transformée. Par exemple d'une liste de nombres entiers on crée la liste qui contient les carrés de ces nombres et on a

```
listeCarree([],[]).
listeCarree([T|R],[X|L]) :- X is T*T, listeCarree(R,L).

?- listeCarree([2,4,5],L).
L = [4, 16, 25].
```

Ascèse Considérons une liste composée des nombres réels. Écrire un programme en Prolog qui permet d'obtenir une liste qui contient les valeurs entières, immédiatement inférieures aux valeurs de la liste.

N.B. Le foncteur `floor` permet d'obtenir la valeur entière $\lfloor x \rfloor$ pour tout $x \in \mathbb{R}$.

### Mapping partiel

La transformation concerne certains éléments de la liste. Par exemple dans une liste qui contient des entiers et des mots, on veut obtenir la liste qui contient les mots et remplace les entiers par leur carré. On a :

```
listeCarreePart([],[]).
listeCarreePart([T|R],[T|L]) :- not(integer(T)), listeCarreePart(R,L).
listeCarreePart([T|R],[X|L]) :- integer(T),X is T*T, listeCarreePart(R,L).

listeCarreePart([2, 4, toto, 1, 6],L).
L = [4, 16, toto, 1, 36] .
```

ASCÈSE 9.1 *Écrire en Prolog un programme qui permet d'obtenir la liste des nombres pairs d'une liste d'entiers.*

### Mapping multiple

La transformation produit deux listes en fonction de la nature des éléments de la lsite d'origine. Par exemple d'une liste qui contient des mots et des entiers, on veut obtenir une liste qui contient les carrés des entiers et une autre liste qui contient les mots. On a

```
listeCMult([],[],[]).
listeCMult([T|R],Ln,[T|L]) :- not(integer(T)), listeCMult(R,Ln,L).
listeCMult([T|R],[X|L],La) :- integer(T), X is T*T, listeCMult(R,L,La).

?- listeCMult([2, 4, toto, 1, 6],Ln, La).
Ln = [4, 16, 36],
La = [toto, 1] .
```

ASCÈSE 9.2 *Écrire en Prolog un programme qui permet, à partir d'une liste des nombres entiers, d'obtenir deux listes, une contenant les nombres impairs et une autre contenant les nombres pairs.*

### Mapping complet avec état

On peut envisager une transformation où la transformée d'un élément dépend non seulement de l'élément en question mais aussi de l'état du calcul. Cet état de calcul sera représenté par un ou plusieurs prédicats. Par exemple étant donnée une liste avec des nombres, on cherche à obtenir une nouvelle liste qui contient pour chaque élément la somme cumulative de tous les éléments qui le précèdent dans la liste plus l'élément en question. On a

```
sommeC(L,LR) :- etatSC(0,L,LR).
etatSC(_,[],[]).
etatSC(NCumul,[T|R],[NCumul1|L]) :- NCumul1 is NCumul+T, etatSC(NCumul1,R,L).

?- sommeC([1, 2, 3, 5],L).
L = [1, 3, 6, 11]
```

Le prédicat auxiliaire `etatSC/3` contient le programme Prolog pour le calcul de l'état.

ASCÈSE 9.3 *Soit une liste sans sous-listes. Écrire en Prolog un programme qui produit une liste contenant des sous-listes de deux éléments chacune et dont le premier élément est un élément de la première liste et le deuxième le numéro d'ordre dans la liste de cet élément. Exemple la liste* `[tot,koko,lolo,momo]` *donne* `[[tot, 1], [koko, 2], [lolo, 3], [momo, 4]]`.

### Mapping séquentiel avec état

Considérons une liste qui contient plusieurs repetitions du même mot à la suite, comme par exemple la liste `[1,1,4, toto, toto, toto,w, w, W,W,4,4]`. On veut obtenir une liste compactée qui contient les éléments précédés du nombre de repétitions successives. On a

```
nbRepet([T|R], LR) :- nbRepet([T|R], T, 0, LR).
nbRepet([], X, N, [N*X]).
nbRepet([T|R], T, N, LR) :- N1 is N +1, nbRepet(R,T,N1,LR).
nbRepet([T|R], X, N, [N*X|Z]) :- T\== X, nbRepet(R,T,1,Z).

?- nbRepet([1,1,4, toto, toto, toto,w, w, 4,4],LR).
LR = [2*1, 1*4, 3*toto, 2*w, 2*4] ;
```

Comme précédement la description de l'état est contenu dans le prédicat `nbRepet/4`.

#### Mapping de regroupage avec état

Soit un prédicat `p/2` dont le premier argument représente une quantité d'éléments du second argument. On forme une liste avec des prédicats `p` ayant des argument différents, par exemple la liste `[p(10, pieces1), p(2,pieces2), p(3,pieces1),p(5,pieces05),p(4,pieces2)]` où chaque prédicat représente le nombre de pièces de 1 euro, 2 euros et 0.5 euro. On veut regrouper ces éléments par groupe de même pièce, c-à-d. avoir la réponse `[p(13, pieces1), p(6, pieces2), p(5, pieces05)]`.On a :

```
collect([],[]).
collect([p(N,X)|R],[p(T,X)|R1]) :- lebesgue(X,N,R,Q,T), collect(Q,R1),!.
lebesgue(_,N,[],[],N).
lebesgue(X,N,[p(N1,X)|R],Q,T) :- M is N + N1, lebesgue(X,M,R,Q,T).
lebesgue(X,N,[Q|R],[Q|Rs],T) :- lebesgue(X,N,R,Rs,T).

? collect([p(10, pieces1), p(2,pieces2), p(3,pieces1),p(5,pieces05),p(4,pieces2)],L).
L = [p(13, pieces1), p(6, pieces2), p(5, pieces05)].
```

Ici c'est le prédicat `lebesgue/5` qui représente l'état.

## 9.2 Opérations numériques avec les listes

En s'inspirant des techniques du mapping, on peut envisager de créer une librairie d'opérations arithmétiques avec des listes numériques. Il s'agit, étant données deux listes, d'effectuer une opération numérique élément par élément entre ces deux listes et stocker le résultat dans une troisième liste.

On précise d'abord les opérations numériques que nous allons utiliser à l'aide du programme Prolog suivant :

```
oper(add, X, Y, R) :- R is X + Y.
oper(soust, X, Y, R) :- R is X-Y.
oper(mult, X, Y, R) :- R is X * Y.
oper(div, X, Y, R) :- R is X / Y.
oper(max, X, Y, R) :- (X > Y, R is X); R is Y.
oper(min, X, Y, R) :- (X < Y, R is X); R is X.
oper(sqrt, X, _, R) :- R is sqrt(X).
oper(abs, X, _, R) :- R is abs(X).
oper(sign, X, _, R) :- R is sign(X).
oper(sin, X, _, R) :- R is sin(X).
oper(cos, X, _, R) :- R is cos(X).
oper(tan, X, _, R) :- R is tan(X).
oper(asin, X, _, R) :- R is asin(X).
oper(acos, X, _, R) :- R is acos(X).
oper(atan, X, _, R) :- R is atan(X).
oper(log, X, _, R) :- R is log(X).
oper(log10, X, _, R) :- R is log10(X).
oper(exp, X, _, R) :- R is exp(X).
oper(floor, X, _, R) :- R is floor(X).
oper(ceiling, X, _, R) :- R is ceiling(X).
```

```
    oper(truncate, X, _, R) :- R is truncate(X).
    oper(integer, X, _, R) :- R is integer(X).
    oper(float, X, _, R) :- R is float(X).
    oper(float_fractional_part, X, _, R) :- R is float_fractional_part(X).
    oper(sqrt, X, _, R) :- R is sqrt(X).
```

Le programme qui permet de faire ces opérations entre les éléments de deux listes est :

```
opListes([],[],[],_):-!.
opListes([T1|R1],[],[T|R],Oper) :- oper(Oper,T1,_,T),
                                   opListes(R1,[],R,Oper),!.
opListes([T1|R1],[T2|R2],[T|R],Oper) :- length(R1,N1),length(R2,N2), N1 == N2,
                                        oper(Oper,T1,T2,T),
                                        opListes(R1,R2,R,Oper).

?- opListes([1,2,3],[3,2,1],S,add).
S = [4, 4, 4].
?- opListes([1.5,-2.3,3.8],[],S,float_fractional_part),opListes(S,[],C,acos).
S = [0.5, -0.3, 0.8],
C = [1.0472, 1.87549, 0.643501].
```

Nous pouvons aussi envisager, étant donnée une liste numérique, d'effectuer des opérations avec ses éléments. On se limite aux opérations d'addition de tous les éléments d'une liste et de multiplication. On a le programme :

```
fctListe(L,S,Oper) :- (Oper == add,fctListeAccu(L,0,S,Oper));
                      (Oper == mult,fctListeAccu(L,1,S,Oper))   .
fctListeAccu([],S,S,_).
fctListeAccu([T|R],Sp,S,Oper) :- oper(Oper,Sp,T,Spn),
                                 fctListeAccu(R,Spn,S,Oper).

?- fctListe([1.5,-2.3,3.8],S,add).
S = 3.0 ;
?- fctListe([1,-2,3],S,mult).
S = -6.
```

ASCÈSE 9.4 *Écrire en Prolog un programme qui calcule le produit intérieur de deux vecteurs.*

## 9.3 Opérations avec les matrices

Nous pouvons utiliser les dévéloppements précédent pour faire des opérations avec des matrices. Premièrement pour représenter une matrice nous envisageons d'utiliser des listes composées des sous-listes. Chaque sous-liste serait une colonne de la matrice. Ainsi la liste `[[2,1,1], [4,2,4], [7,3,5]]` représente la matrice $\begin{bmatrix} 2 & 4 & 7 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix}$.

Occupons-nous d'abord de la transposée d'une matrice. Nous avons le programme :

PROGRAMME 9.3.1

```
    transpose([[] | _],[]).
    transpose(R,[T | Rs]) :- premiereCol(R,T), suiteCol(R,H),
                             transpose(H,Rs).
```

```
premiereCol([],[]).
premiereCol([[T | R] | Rs],[T | Ls]) :- premiereCol(Rs,Ls).

suiteCol([],[]).
suiteCol([[T | R] | Rs],[R | Ls]) :- suiteCol(Rs,Ls).
```

Pour construire le programme de multiplication de deux matrices, nous procéderons en trois étapes.

**1°** Multiplication d'un vecteur avec une matrice. Le vecteur se présente sous la forme d'une liste simple. Nous avons :

```
prodVecMat(A,[],[]).
prodVecMat(A,[B | R],[Res | Rs]) :- produitInt(A,B,Res),
                                    prodVecMat(A,R,Rs).
```

**2°** Multiplication de deux matrices qui se présentent sous forme des listes contenant des listes mais qu'on suppose qu'elles ont été préalablement arrangées pour que la multiplication soit correcte. Nous avons :

```
prodMatMat([],B,[]).
prodMatMat([A | R],B,[Res | Rs]) :- prodVecMat(A,B,Res),
                                    prodMatMat(R,B,Rs).
```

**3°** Enfin la multiplication des matrices :

```
multMat(A,B,C) :- transpose(A,AT), prodMatMat(AT,B,C).
```

Remarquons qu'il n'est pas nécessaire de faire le test de la conformité des dimensions des matrices, parce qu'il sera fait par le programme `prodInt`. Nous donnons ci-après un programme de calcul du nombre de lignes et des colonnes d'une matrice.

```
dimMat([T | R],NbLigne,NbCol) :- not(atom(T)), length(R,Nb),
                                 NbCol is Nb+1,
                                 length(T,NbLigne).
dimMat([T | R],NbLigne,NbCol) :- (atom(T); number(T)),
                                  NbCol is 1, length(R,Nb),
                                  NbLigne is Nb+1.
```

## 9.4 Accumulateur

La *technique de l'accumulateur* est utilisée pour déterminer des prédicats de manière récursive.

Commençons par un exemple : nous voulons écrire un programme qui calcule la somme des entiers qui sont dans une liste. Il s'agit d'une récursivité numérique et le programme est le suivant :

```
somme([],0).
somme([T|R],S) :- somme(R,S1), S is S1 + T.
```

Par exemple à la question

```
?- numlist(1,100,L), somme(L,S).
```

on obtient la réponse

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
R = 5050
```

Le programme `numlist(1,100,L)` crée la liste `L` contenant les entiers de `1` à `100`.

Le programme somme s'auto-appelle de façon récursive et construit une pile LIFO dans laquelle il stocke tous les éléments de la liste `L` afin de faire les calculs après avoir exécuter le test d'arrêt `somme([],0)`.. On peut imaginer que si la liste `L` contient un nombre important de termes, la longueur de la pile ne sera pas suffisante pour contenir tous les éléments de `L`. En effet nous avons :

```
?- numlist(1,100000,L), somme(L,R).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
R = 5000050000 ;

?- numlist(1,1000000,L), somme(L,R).
ERROR: Out of local stack
```

On pourrait éviter l'explosion de la pile, si on utilise un accumulateur qui, à chaque appel recursif, accumule le résultat intermédiare de la somme au lieu de le stocker dans la pile en vue de leur exploitation ultérieure. En fait au lieu de faire un appel récursif et de différer le calcul de la somme et de l'effectuer, par conséquent, après avoir rencontré le test d'arrêt, on réalise ce calcul chaque fois avant l'appel récursif et on le stocke dans un accumulateur. Dans ce cas le test d'arrêt est utilisé pour terminer les calculs et passer le résultat stocké dans l'accumulateur à la variable de sortie du programme. Nous avons donc le programme suivant :

```
sommeA(L,S) :- somme(L,0,S).
somme([],S,S).
somme([T|R],Accu,S) :- Accu1 is Accu + T,somme(R,Accu1,S).
```

Dans ce cas, on a

```
?- numlist(1,1000000,L), sommeA(L,R).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
R = 500000500000 ;
```

Mais comme toutes les bonnes choses ont une fin, nous avons

```
\texttt{numlist
}(1,10000000,L), sommeA(L,R).
ERROR: Out of global stack
```

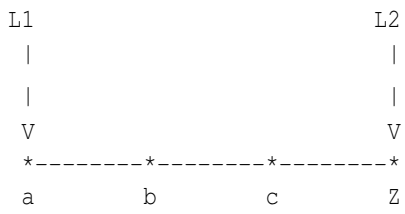
Dans ce cas la solution consiste à augmenter la taille de la pile. Pour ce faire on utilise le prédicat `manpce/0` dans la fenêtre de travail. Sur la fenêtre qui s'ouvre on clique sur `File -> Edit preferences -> Prolog Stack Limits` et augmenter la taille de différentes piles. Néanmoins pour des ordinateurs à 32 bits le résultat n'est pas spéctaculaire.

ASCÈSE 9.5 *Écrire un programme en Prolog qui étant donnée une listte construit une liste qui contient comme sous-listes les décalages à gauche des éléments de la liste.*

## 9.5 Différences de liste

Les différences de liste permet à des listes incomplètes d'être spécifiées et construites de façon incrémentale au fur et à mesure que le programme s'exécute. En ce sens elles constituent une généralisation de l'accumulateur et permettent de réduire l'effort de calcul.

L'idée de la différence de liste est de représenter un segment d'une liste comme une paire de termes, une tête, notée `L1`, et le reste, noté `L2`. Soit un segment de liste qui contient au début les éléments `a,b,c` et le reste est représenté par une variable `Z` comme

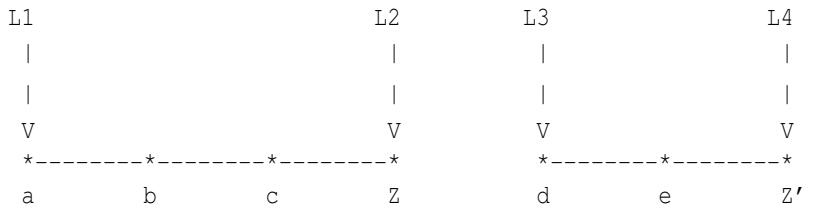



Cette liste a la forme `[a, b, c | Z]`. On unifie ainsi `L1` avec `[a, b, c | Z]` et `L2` avec `Z`. Il est important que les deux variables `Z` se referent au même objet. On construit ainsi la différence de liste `L1` et `L2` qui est unifiée à `[a, b, c]`.

L'utilisation de la différence de liste permet de programmer des appels recursifs à coût constant indépendament de la longueur des listes. Par exemple considérons le prédicat `append` qui fait la concaténation de deux listes `L1` et `L3` et dont le programme en Prolog est le suivant :

```
append([],L3,L3).
append([T|R],L3,[T|L]) :- append(R,L3,L).
```

Ici le nombre d'appels recursifs est fonction de la longueur de la première liste. Supposons maintenant qu'on rajoute à la fin de la première liste `L1` une variable `Z` comme ci-dessus et on définit une deuxième liste `L2` que l'on unifie avec cette variable `Z`. On fabrique ainsi une différence de liste `L1` et `L2`. On fait la même chose avec la liste `L3` et on obtient la différence de liste `L3` et `L4`. On a le schéma ci-après :




On cherche à faire la concaténation de la différence de liste `L1` et `L3` avec la différence de liste `L3` et `L4`. Comme le résultat doit être aussi une différence de liste, on écrira que ce résultat est la différence de liste `X` et `Y`. En utilisant le schéma ci-dessus, on observe que `X` doit être unifié

avec `L1`, `Y` avec `L4` et `L2` avec `L3`. Nous avons donc pour la concaténation avec de différences de liste le programme suivant :

```
appdl(L1,L2,L2,L4,L1,L4).
```

avec appel comme suit

```
?- appdl([a,b,c|Z1],Z1,[d,e|Z2],Z2,X,Y).
Z1 = [d, e|Y],
Z2 = Y,
X = [a, b, c, d, e|Y].
```

Ce programme n'a pas d'appels recursifs et donc son coût en temps d'exécution est constant indépendamment de la longueur de la liste.

Une différence de liste de `L1` et `L2` s'écrit en Prolog `L1-L2`. En utilisant cette notation, on peur écrire le programme précédent comme suit :

```
app(L1–L2, L2–L3, L1–L3).
```

qui exprime le simple fait mathématique `(L1-L2) + (L2-L3) = L1-L3`.

Nous avons les propriétés suivantes :

- `L-L` représente la liste nulle.
- `[a | Z] - Z` est la différence de liste qui contient `a`.
- L'unification de la différence de liste `X` avec `Y-[]` "rectifie" la liste, c'est-à-dire instancie `Y` avec une liste sans reste. Par exemple si on unifie `[a,b,c | Z] - Z` avec `Y-[]`, alors on instacie `Y` avec `[a,b,c]`.

ASCÈSE 9.6 *Écrire un prédicat prolog qui aplatit une liste (c'est-à-dire qu'il supprime les indiactions de sous-listes)*

- *en utilisant le prédicat appende, et*
- *en utilisant la différence de liste.*

ASCÈSE 9.7 *Écrire un prédicat prolog qui construit une liste contenant tous les décalages circulaires gauche d'une liste. Utiliser la différence de liste et le prédicat `app` défini ci-dessus.*

## 9.6 Fonctionnement de la coupure

Pour l'enchaînement des buts et des clauses, `Prolog` dispose d'une facilité en ce qui concerne le contrôle de la circulation de l'information. Il s'agit du symbole `cut`, noté par `!`. Il peut être insérer dans le programme comme un but mais ses effets sont à retardement. L'exemple suivant présente le comportement du cut. Soit une primitive exprimée à l'aide de deux clauses suivantes :

```
B :– C1, C2, . . . ,Ci, !, C(i+1), . . ., Cn .
B :– D1, D2 . . . , Dm .
```

Si une de conditions $C_1$, $C_2$, . . . , $C_i$ n'est pas satisfaite, alors le contrôle du programme passe à la clause suivante. Si par contre toutes les conditions $C_1$, $C_2$, . . . , $C_i$ sont satisfaites, alors le cut est atteint et, en tant que but, est réalisé. Ce qui aura comme conséquence l'interdiction partielle du backtracking sur cette primitive. Ainsi le backtracking sur les conditions $C_{i+1}$, . . . , $C_n$ est permis mais il n'est plus possible sur les conditions $C_1$, $C_2$, . . . , $C_i$, ni d'ailleurs sur les conditions des clauses qui s'en suivent, c'est-à-dire sur les conditions $D_1$, $D_2$, . . . , $D_m$.

La coupure modifie, en fait, le comportement du programme et elle peut même le rendre logiquement erroné. L'exemple suivant fournit l'équivalent logique d'un programme sans ou avec coupure.

```
p :- a, b.
p :- c.
```

Ce programme est équivalent à la fbf $(a \wedge b) \vee c$. Si on place une coupure à la première clause, on aura le programme

```
p :- a, !, b.
p :- c.
```

La fbf équivalente est maintenant $(a \wedge b) \vee (\neg a \wedge c)$. Si on change l'ordre des clauses, on a

```
p :- c.
p :- a, !, b.
```

qui est équivalent à la fbf $(c \vee a \wedge b)$, c'est-à-dire est équivalent au programme initial.

L'exemple suivant fournit un traitement exhaustif de cut.

PROGRAMME 9.6.1

```
(1) p(a).

(2) p(X):- q(X),r(X).
(3) p(X):-u(X).

(4) q(a)
(5) q(b).
(6) q(c).

(7) r(a).
(8) r(b).
(9) r(d).

(10) u(d).
```

Nous savons que dans le cas normal la réponse à la requête `?-p( X)` donne `X = a, X = a, X = b, X = d`. Si nous changeons la première clause par la clause :

`p(a) :-!.`

la seule réponse fournie sera `X = a`, car la seule branche développée par l'arbre de résolution serait la branche (1). Remplaçons maintenant la deuxième clause par :

`p(X) :-!, q(X), r(X).`

Les réponses obtenues dans ce cas seront `X = a, X = a, X = b`, car les branches (3) et (10) de l'arbre de résolution serraient élaguées. Si nous déplaçons la coupure vers la droite :

```
p(X) :- q(X), !, r(X).
```

nous obtenons les réponses `X = a, X = a`, car dans ce cas ce sont les branches (3) et (5) à (10) qui seront élaguées.

Les règles qu'il faut suivre pour l'écriture des programmes avec cut sont les suivantes :

- Chaque clause doit refléter une règle de logique qui détermine la vérité des faits qui filtre le but. Dans ce cas, nous pouvons ajouter des cuts pour éviter de parcourir, dans l'arbre de la résolution, des branches de calculs inutiles ou superflus.
- Placer le cut le plus proche possible du début de la clause.
- Il ne faut pas utiliser des cuts à la dernière clause d'une primitive.

### 9.6.1 Coupure rouge et coupure verte

Il y a en réalité deux sortes de cut : le *cut vert* et le *cut rouge*. Le premier sert à élaguer des branches sur lesquelles on sait qu'il n'y ait pas de solutions mais il ne modifie pas la sémantique du programme. Par exemple soit le programme qui trouve le maximum de deux nombres.

PROGRAMME 9.6.2

```
minimum(X,Y,X) :- X=<Y.
minimum(X,Y,Y) :- X>Y.
```

Étant donné que les deux clauses sont exclusives, on sait que si la première réussit, la seconde échouera. Par conséquent nous pouvons éviter son examen par `Prolog` en utilisant une coupure comme ci-après.

PROGRAMME 9.6.3

```
minimum(X,Y,X) :- X=<Y, !.
minimum(X,Y,Y) :- X>Y.
```

Cette coupure ne change en rien le sens du programme, elle accélère seulement son exécution.

Par contre le cut rouge peut élaguer des branches sur lesquelles il y a des solutions et, par conséquent, modifie le comportement du programme, c'est-à-dire change la semantique du programme.

Reprenons l'exemple précédent. Du fait de l'introduction de la coupure à la première clause, on sait que si on arrive à la seconde clause, on a forcement `X > Y`. Donc, nous pouvons penser, à tort comme on le verra par la suite, que le test `X > Y` effectué par cette clause est superflu. Ainsi on peut avoir le programme

PROGRAMME 9.6.4

```
minimum(X,Y,X) :- X=<Y, !.
minimum(X,Y,Y) .
```

Ici le sens du programme est modifié et à tel point que sa réponse n'est pas toujours correcte comme, par exemple, à la question `?minimum(2,5,5)`. Ceci est dû au fait que la première clause n'est pas satisfaite, mais la deuxième par contre est satisfaite.

ASCÈSE 9.8 *Compléter la 1e clause du programme précédent afin qu'il devient correct sans modifier la seconde clause.*

On verra mieux l'usage du cut rouge avec le programme suivant qui détermine une liste qui est l'intersection, au sens ensembliste du terme, de deux autres listes.

PROGRAMME 9.6.5

```
intersection([],_,[]).
intersection([H|T],L,[H|T2]) :- membre(H,L),
                                intersection(T,L,T2).
intersection([H|T],L,T2) :-     intersection(T,L,T2).
```

À la question `?- intersection([a,b,c], [a,b,c], L).`, nous avons les réponses :

```
L = [a, b, c] ;
L = [a, b] ;
L = [a, c] ;
L = [a] ;
L = [b, c] ;
L = [b] ;
L = [c] ;
L = []
```

ce qui, bien entendu, n'est pas souhaitable.

L'utilisation d'un cut rouge peut rectifier la situation, comme ci-après :

PROGRAMME 9.6.6

```
intersection([],_,[]) :- !.

intersection([H|T],L,[H|T2]) :- membre(H,L), !,
                                intersection(T,L,T2).

intersection([_H|T],L,T2) :- intersection(T,L,T2).
```

Le cut au test d'arrêt est obligatoire si on ne veut pas que l'appel avec des listes qui contiennent aussi des variables dégénère.

Ici il s'agit d'un cut rouge car il modifie les résultats, c'est-à-dire la sémantique du programme.

### 9.6.2 Arrêt après la première solution

Le coupure peut être utilisée pour que `Prolog` s'arrête dès qu'il a trouvé une solution.

ASCÈSE 9.9 *Écrire le programme `membre` qui s'arrête dès qu'il trouve le premier membre de la liste.*

La première solution d'un but peut être expressément construit à l'aide du prédicat `premiereSolution/1` comme suit :

PROGRAMME 9.6.7

```
premiereSolution(But) :- call(But), !.
```

Ainsi pour le programme de l'ascèse précédente on a

`premiereSolution(membre(X,[a, b, c])).`

### 9.6.3 Coupure et clause conditionnelle

Nous avons utilisé dans le programme 9.6.6 la coupure pour construire l'intersection ensembliste de deux listes. Néanmoins ce programme, malgré les lettres de noblesse qu'il dispose, n'est pas dépourvu des faiblesses. Par exemple la réponse à la question

`?-intersection([a, b ,c],[d, b, a],[a]).`

est `oui` !

Le problème vient du fait que l'intersection est construite de façon implicite à la deuxième clause. Une construction explicite nous permettra d'éviter ce problème. On aura donc

PROGRAMME 9.6.8

```
intersection1([],_,[]) :- !.

intersection1([H|T],L,T2) :- membre(H,L), !,
                             T2 = [H|Z],
                             intersection1(T,L,Z).

intersection1([_H|T],L,T2) :- intersection1(T,L,T2).
```

En regardant bien le programme, on constate que nous avons une construction conditionnelle du type `SI ... ALORS ... SINON` que nous pouvons exprimer à l'aide du programme suivant

PROGRAMME 9.6.9

```
siAlorsSinon(X,Y,Z) :- call(X), !, call(B).
siAlorsSinon(X,Y,Z) :- call(Z).
```

ASCÈSE 9.10 *Écrire le programme `intersection2` en utilisant le prédicat `siAlorsSinon`.*

ASCÈSE 9.11 Étude du cut rouge.

*Soit la base de données :*

```
ferie(mercredi, premierMai).
temps(mercredi,beau).
temps(samedi,beau).
temps(dimanche,beau).
weekend(samedi).
weekend(dimanche).
promenade(Jour) :- temps(Jour, beau),
                   weekend(Jour).
promenade(Jour) :- ferie(Jour,premierMai).
```

*Pourriez-vous expliquer le comportement de `Prolog` dans chacune des situations suivantes :*

*(1) Si on pose la question `?-promenade(Quand).` on obtient comme réponses*

```
Quand = samedi ;
Quand = dimanche ;
Quand= mercredi ;
```

*Explications.*

(2) *Si maintenant on change la primitive* `promenade` *comme suit*

```
promenade1(Jour) :- temps(Jour, beau),
                    weekend(Jour),!.
promenade1(Jour) :- ferie(Jour,premierMai).
```

*on a comme réponse*

```
Quand = samedi;
```

*Explications.*

(3) *Si on utilise*

```
promenade2(Jour) :- temps(Jour, beau), !,
                    weekend(Jour).
promenade2(Jour) :- ferie(Jour,premierMai).
```

*on n'obtient aucune réponse. Pourquoi ?*

(4) *Si on écrit*

```
promenade3(Jour) :- !, temps(Jour, beau),
                    weekend(Jour).
promenade3(Jour) :- ferie(Jour,premierMai).
```

*on obtient la réponse*

```
Quand = samedi;
Quand = dimanche;
```

*Explications.*

## 9.7 La négation comme échec

Le coupe-choix permet d'introduire la négation. Par exemple le programme suivant introduit le prédicat `different` :

```
dif(X, X) :-!, fail.
dif(X,Y).
```

Nous pouvons aussi introduire la négation non :

```
non(But) :- call(But),!, fail.
non(But).
```

En réalité la négation introduite de cette façon n'est pas une vraie négation logique car elle est fondée sur le succés de l'échec du but ou, en d'autres termes, un but est faux si nous pouvons démontrer que nous ne pouvons pas le démontrer. Le problème a déjà été évoqué en Logique Computationnelle, chapitre 7, où nous avons notamment écrit :

« Considérons un programme défini $E$ et soit $\mathcal{B}_E$ la base de Herbrand de $E$. [...] Comme $E$ est un programme défini, c'est-à-dire qui contient des connaissances positives, la base de Herbrand contient seulement des faits positifs. Par conséquent on n'a pas la possibilité d'induire des connaissances négatives. Malgré tout nous avons besoin de savoir, par des méthodes d'induction logique, qu'un fait n'existe pas. Il s'agit essentiellement d'un problème d'interprétation de la négation, c'est-à-dire trouver une méthode qui permet d'obtenir de l'information négative. Il existe plusieurs méthodes mais

pour ce cours introductif nous allons nous restreindre aux trois plus anciennes parues simultanément dans le livre *Logic and Data Bases*, édité par H. Gallaire et J. Minker en 1978.

– *Hypothèse du monde fermé* (CWA – Closed World Assumption) qui fut introduite par R. Reiter. Selon cette hypothèse – qui est, en fait, une règle – toute connaissance qui ne fait pas partie explicitement d'une base de données ou qu'il ne peut pas être induite logiquement de cette même base de données, n'existe pas et donc c'est sa négation qui a la valeur de vérité « vraie ». Il s'agit d'une hypothèse très naturelle dans les bases de données relationnelles. Mais pour les bases de données logiques, que sont tous les programmes logiques, il en va tout autrement. En effet, formellement l'hypothèse du monde fermé introduit la règle d'inférence suivante :

**(R-MF)** $$\frac{\vdash \neg(E \vDash A)}{\vdash \neg A}$$

En toute rigueur on aurait dû conclure que $E \vDash \neg A$ si on avait la preuve que $A$ n'est pas une conséquence logique du programme $E$. Or cette affirmation n'est licite que si on fait fi de la base de Herbrand $\mathcal{B}_E$ de $E$ et qu'on tient compte seulement de la base de données du programme.En effet, en général la base de Herbrand est un ensemble infini et de ce fait le problème de savoir si une fbf est une conséquence logique d'un programme est indécidable ([1]). Nous pouvons donc en conclure que l'hypothèse du monde fermé n'est pas, en général, applicable.

– *La négation considérée comme un échec*. Il s'agit d'une idée introduite par K. L. Clark. Nous pouvons la présenter comme étant une restriction de la règle (R-MF) dans le cas fini. En effet la règle (R-MF) stipule en substance que

$\neg A$ réussit ssi $A$ ne peut pas être prouvé

Clark a suggeré que, dans cette proposition, la négation soit consïdérée comme le résultat d'un échec fini, c'est-à-dire d'un échec qu'on peut obtenir, à l'aide de l'arbre de dérivation SLD, en un nombre fini d'étapes. Dans ce cas nous avons l'interprétation affaiblie suivante :

$\neg A$ réussit ssi $A$ échoue de façon finie

De façon formelle nous avons la définition suivante :

DÉFINITION 9.7.1 *Soit $E$ un programme défini et $B$ un but défini. Un arbre de dérivation SLD d'échecs fini pour $E \cup \{B\}$ est un arbre qui est fini et ne contient pas de branches de succés.*

Si nous voulons établir que le programme $E$ induit $\neg A$, il faut montrer que l'arbre de dérivation pour le but $B : \leftarrow \neg A$ est un arbre d'échecs finis. Nous introduisons ainsi un nouveau type de résolution, la résolution SLDNF qui est une résolution SLD avec « Negation » comme « Failure ». Elle complète la résolution SLD lorsque le but est sous forme d'un atome négatif, i.e. $B : \leftarrow \neg A$, en introduisant les règles suivantes :

$\neg A$ réussit ssi $A$ donne un arbre SLD d'échecs fini
$\neg A$ donne un arbre SLD d'échecs fini ssi $A$ réussi

Ainsi si $\neg A$ réussit, alors il est supprimé de la question et on passe à l'examen de la suite de la question. Si par contre il échoue de façon finie, alors la question est considérée comme aboutissant à un échec. Nous avons donc, en utilisant la résolution SLDNF, les résultats suivants :

– $E \vdash B \circ \theta$ s'il existe un arbre de dérivation SLDNF de $E \cup \{B\}$ avec réponse calculée $\theta$.

– $E \vdash \neg B$ s'il existe un arbre de dérivation SLDNF d'échecs fini pour $E \cup \{B\}$.

Telle quelle la résolution SLDNF pose le problème de sa justesse et aussi de l'équivalence entre implication sémantique et implication syntaxique. [...] »

Ainsi le fonctionnement de ce type de négation ne va pas sans quelques inconvénients. Par exemple, soit la base de données suivante :

---

1. ce qui signifie qu'il n' y a pas un algorithme qui, pour un programme donné $E$ et un but $B$, peut répondre si $E \vDash B$ dans un temps fini. Notons que l'élève cultivé peut, avec profit, relier ce problème avec celui de l'arrêt dans les machines de Turing en se reportant à la bibliographie spécialisée.

```
innocent(toto).
occupation(jojo, ailleurs).
coupable(koko).
innocent(X) :- occupation(X, ailleurs).
coupable(X) :- occupation(X, ici).
```

Si nous posons la question `innocent(fourier).` la réponse sera `non`. Pour améliorer le programme on pense ajouter la clause suivante :

```
coupable(X) :- non(innocent(X)).
```

Alors à la question `coupable(fourier).` nous aurons comme réponse `oui`.

Avec la base de données suivante, nous avons des résultats encore plus extravagants :

```
voiture(rouge).
voiture(verte).
voiture(bleue).
decapotable(rouge).
decapotable(bleue).
berline(X) :- non(decapotable(X)).
```

À la question `?-voiture(X), berline(X).` nous avons comme réponse `oui` et à la question `?-berline(X), voiture(X).` la réponse est `non`.

La raison de ce comportement est simple. Dans la première question, lorsque le prédicat `berline(X)` est activé, la variable `X` est déjà unifiée avec une valeur, e.g. `rouge` et donc le `non` s'applique sur un prédicat dont l'argument est une constante. Dans la deuxième question le `non` s'applique sur une variable non unifiée. Prolog cherchera dans sa base de données à trouver une clause `non(decapotable(_))`, mais il n'en trouvera pas, car d'habitude dans une base des données nous n'indiquons pas les propriétés dont il est dépourvu un objet. Ainsi la recherche échoue et la réponse est non.

Nous pouvons donc en déduire une règle pour l'utilisation du `non` : Il faut utiliser le `non` uniquement avec des prédicats dont les arguments sont des constantes ou des variables déjà instaciées (exemplifiées).

## 9.8 Prédicats ensemblistes

Nous allons examiner des prédicats qui permettent de créer des ensembles à partir d'une liste d'éléments. Rappelons que l'occurrence d'un élément dans un ensemble se limite à une fois au plus. Nous pouvons donc simuler un ensemble par une liste qui ne contient pas plusieurs occurrences d'un même élément. Mais cette liste, contrairement à un ensemble, elle est ordonnée.

ASCÈSE 9.12 *Écrire le programme `list2set(L,S)` qui transforme la liste `L` en un ensemble `S`. Exemple : `?- list2set([b,a,c,b,d,c,f,a,d], S).` donne comme réponse `S=[b,c,f,a,d]`.*

Si, à la place d'une liste, nous avons une base de données, Prolog fournit trois prédicats extra-logiques qui permettent de construire soit des ensembles, soit des multi-sets [2], en utilisant les éléments de la base.

2. c'est-à-dire des ensembles où des éléments peuvent apparaître plusieurs fois

Considérons d'abord la bases de données suivante :

```
nom(toto).
nom(koko).
nom(jojo).
nom(toto).

boisson(toto, the).
boisson(toto, lait).
boisson(toto, biere).
boisson(koko, lait).
boisson(koko, the).
boisson(koko, vin).
boisson(jojo, the).
boisson(jojo, vin).
boisson(pierre, lait).

boisson(toto, the, chaud).
boisson(toto, lait, chaud).
boisson(toto, biere, fraiche).
boisson(toto, vin, froid).
boisson(koko, lait, froid).
boisson(koko, the, chaud).
boisson(koko, vin, chaud).
boisson(jojo, the, chaud).
boisson(jojo, vin, chambre).
boisson(pierre, lait, froid).
```

Les prédicats qui fabriquent des ensembles à partir des éléments de cette base sont les suivants :

### 9.8.1 setof

C'est un prédicat d'arité 3 : `setof(X, nomBdD(..., X, ...),S)` où `X` est un élément de la base de données `nomBdD` et `S` est l'ensemble des éléments `X` de la base, regroupés en fonction des valeurs des autres éléments de la base et triés selon l'ordre alphabétique.

EXEMPLE 9.8.1 `?- setof(X,nom(X),S).`
*fournit comme réponse*
`S = [jojo, koko, toto].`

EXEMPLE 9.8.2 `?- setof(X,boisson(X,Y),L).`
*donne les réponses suivantes*

```
X = biere,
S = [toto];
X = lait,
S = [koko, pierre, toto];
X = the,
S = [jojo, koko, toto];
X = vin,
S = [jojo, koko]
```

*c'est-à-dire pour chaque valeur du second élément de la base, S contient l'ensemble des personnes qui boivent cette boisson.*

*Si on veut obtenir l'ensemble de personnes qui boivent du lait, c'est-à-dire l'ensemble des éléments qui sont en première position dans la base de données* `boisson/2`*, on doit poser la question*

```
?- setof(X,boisson(X,lait),S).
```
*qui donnera comme réponse*
```
S = [koko, pierre, toto]
```
*Enfin si on veut avoir l'ensemble de toutes le personnes de la base boisson/2, on doit poser la question*
```
?- setof(X,Y^boisson(X,Y),S).
```
*qui donnera comme réponse*
```
S = [jojo, koko, pierre, toto]
```
*Ici l'opérateur ^ dans la notation* `Y^boisson(X,Y)` *teste s'il existe un* `Y` *tel que* `boisson(X,Y)` *est vrai.*

ASCÈSE 9.13 *Écrire un programme qui fournit l'ensemble de toutes les boissons de la base de données* `boisson/2`.

ASCÈSE 9.14 *Écrire un programme qui fournit l'ensemble de toutes les personnes de la base de données* `boisson/3` *qui boivent du lait froid.*

ASCÈSE 9.15 *Écrire un programme qui fournit l'ensemble de toutes les personnes de la base de données* `boisson/3` *qui boivent une boisson froide.*

Il est à noté que s'il n'est pas possible de former un ensemble, le prédicat `setof` échoue. Tester, par exemple la clause `setof(X,boisson(X,champagne),S).`

ASCÈSE 9.16 *En utilisant* `setof/3` *et* `member/2` *écrire les programmes Prolog qui permettent :*

*(1) de tester si un ensemble est sous-ensemble d'un autre ensemble.*

*(2) de tester l'égalité entre deux ensembles.*

*(3) de faire l'union de deux ensembles.*

*(4) de faire l'intersection de deux ensembles.*

### 9.8.2 bagof

Prédicat d'arité 3 qui a le même comportement que `setof` mais il retourne des multi-sets au lieu des ensembles. Ces multi-sets ne sont pas ordonnés.

EXEMPLE 9.8.3 `?- bagof(X,nom(X),S).`
*fournit comme réponse*
```
S = [toto, koko, jojo, toto].
```

ASCÈSE 9.17 *Poser les questions suivantes et comparer les réponses avec celles obtenues avec* `setof`.

*(1)* `?- bagof(X,boisson(X,Y),L).`

*(2)* `?- bagof(X,boisson(X,lait),S).`

*(3)* `?- bagof(X,Y^boisson(X,Y),S).`

ASCÈSE 9.18 *Écrire un programme qui fournit le multi-set de toutes les boissons de la base de données* `boisson/2`.

ASCÈSE 9.19 *Écrire un programme qui fournit le multi-set de toutes les personnes de la base de données* `boisson/3` *qui boivent du lait froid.*

ASCÈSE 9.20 *Écrire un programme qui fournit le multi-set de toutes les personnes de la base de données* `boisson/3` *qui boivent une boisson froide.*

Comme avec `setof`, s'il n'est pas possible de former un ensemble, le prédicat `bagof` échoue. Tester, par exemple la clause `bagof(X,boisson(X,champagne),S).`

### 9.8.3 `findall`

Prédicat d'arité 3 qui fournit la liste de tous les éléments d'une base de données qui se trouvent à une position fixée.

EXEMPLE 9.8.4 `?- findall(X,boisson(X,Y),S).`
*fournit comme réponse tous les éléments qui sont en première position dans boisson/2 :*
`S = [toto, toto, toto, koko, koko, koko, jojo, jojo, pierre]`

L'opérateur ^ n'est pas pris en considération par `findall`

EXEMPLE 9.8.5 `?- findall(X,lait^boisson(X,Y),S).`
*fournit la même réponse que précédemment*
`S = [toto, toto, toto, koko, koko, koko, jojo, jojo, pierre]`

Contrairement aux deux prédicats précédents, `findall` s'il n'est pas possible de former un ensemble, retourne la liste vide []. Tester, par exemple la clause
`findall(X,boisson(X,champagne),S).`

ASCÈSE 9.21 *Écrire un programme pour la négation* `not` *en utilisant le prédicat* `findall`.
*Pouvons-nous écrire le même programme en utilisant le prédicat* `setof`*? Si oui, écrivez et tester le programme. Si non, expliquer les raisons.*

## 9.9 Prédicats relatifs aux structures

Parfois il est utile de pouvoir accéder aux éléments d'une structure, par exemple au nom d'un prédicat ou à l'arité d'un prédicat ou, encore, aux noms de ses arguments.

### 9.9.1 `functor`

Le prédicat `functor` appliqué à un prédicat détermine le nom du prédicat et son arité. Ainsi

```
?- functor(somme(X,Y,R), NomFoncteur, Arite).
```

donne comme réponse

```
NomFoncteur = somme,
Arite = 3
```

Bien sûr le même prédicat permet de construire de façon dynamique le nom du prédicat somme `?- functor(P, somme, 3).`
donne comme réponse

```
P = somme(_G955, _G956, _G957).
```

Notons que `functor` sort en erreur si au moins deux de ces arguments ne sont pas instanciés.

### 9.9.2 arg

`arg` permet d'accéder aux noms des arguments d'un prédicat. Ainsi, on a

```
 ?- arg(2, somme(2,3,S),X).
X = 3.
```

et

```
?- arg(3, somme(2,3,S),X).
S = X.
```

Il faut faire attention d'utiliser ce prédicat avec ses deux premiers arguments instanciés. Sinon au lieu de sortir en erreur, fournit des réponses erronées, comme on peut voir à l'aide de deus exemples suivants :

```
?- arg(Y, somme(2,3,S),3).
Y = 2 ;
Y = 3,
S = 3.

?- arg(Y, somme(2,3,S),D).
Y = 1,
D = 2 ;
Y = 2,
D = 3 ;
Y = 3,
S = D.
```

### 9.9.3 name

Ce prédicat permet d'obtenir une liste avec les codes ASCII d'un nom ou le nom correspondant à une liste des codes ASCII comme indiquer aux exemples suivants :

```
?- name(toto,X).
X = [116, 111, 116, 111].

?- name(X,[116,111,116,111]).
X = toto.

?- name(toto,"toto").
true.
```

Notons que le premier argument de `name` doit être un atome.

### 9.9.4 univ

Le prédicat `univ`, noté `=..`, qui sert à composer ou décomposer les termes d'un prédicat, est la réunion de deux prédicats `functor` et `arg`. Ainsi

```
?- somme(X,Y,R) =.. L.
```

fournit comme résultat

```
L = [somme, X, Y, R]
```

et

```
?- T =.. [somme,X,Y,R].
```
a comme réponse `T = somme(X, Y, R)`

L'exemple suivant montre deux utilisations de ce prédicat.

EXEMPLE 9.9.1 *`?- a*b+c =..[F,X,Y].` a comme réponse*

```
F = +,
X = a*b,
Y = c
```

*et `?- a*(b+c) =..[F,X,Y].` a comme réponse*

```
F = *,
X = a,
Y = b+c
```

# 10

# MODÈLES DE HERBRAND. UNIFICATION



Le calcul des prédicats est utilisé dans des problèmes concernant la satisfaction des contraintes ou les bases de données déductives ou, encore, la vérification formelle du logiciel. Ce calcul a toujours la même forme : étant donné un ensemble $E$ des fbf, vérifier si, pour une fbf $A$, on a $E \models A$. Nous avons déjà rencontré et résolu ce problème dans le cas du calcul des propositions. En effet dans ce cas précis le problème de la satisfiabilité des fbf propositionnelles est décidable. Mais dans le cas du calcul des prédicats nous pouvons montrer que le même problème est indécidable [1] !

Afin de surmonter ces difficultés nous avons déjà établi quelques simplifications syntaxiques : passage d'une fbf en fcn et transformation de cette dernière en clause de Horn. De plus le théorème de la réfutation du chapitre précédent nous montre que si on veut démontrer que l'ensemble des clauses de Horn $E$ fournit la preuve de la clause $A$, il suffit de démontrer que l'ensemble des clauses $E \cup \{\neg A\}$ est insatisfiable. Autrement dit il suffit de montrer qu'il n'y a pas d'interprétation qui est un modèle pour $E \cup \{\neg A\}$. Il est évident que sous cette forme la tâche de démonstration est quasiment impossible, sauf à réduire la taille de l'espace de recherche.

L'objectif de ce chapitre est de présenter la démarche pour opérer cette réduction et qui est composée de deux étapes :

1. c'est-à-dire étant donné un ensemble $E$ des fbf on ne peut pas, dans tous les cas, conclure qu'il soit satisfiable ni qu'il soit insatisfiable

– Établir un lien entre la logique des prédicats et celle des propositions. Ce lien est obtenu grâce à l'univers et la base de Herbrand.
– En utilisant l'univers et la base de Herbrand, construire une énumération récursive des modèles de $E$.

## 10.1 La preuve dans la logique des prédicats

Considérons l'ensemble de fbf closes $E = \{B_1, \ldots, B_n\}$.

Le problème de la déduction est de montrer que la fbf close $A$ est une conséquence logique de $E$.

En reformulant ce problème en termes de validité on a le problème suivant :

*Est-ce que la formule*

$$(\omega) \quad B_1 \wedge \ldots \wedge B_n \rightarrow A$$

*est une formule valide ?*

On peut aussi reformuler le même problème en termes de satisfiabilité.

*Est-ce que l'ensemble*

$$E \cup \{\neg A\}$$

*est un ensemble insatisfiable ?*

Par conséquent, la preuve, du point de vue sémantique, dans la logique des prédicats consiste à établir que chaque interprétation qui est un modèle pour l'ensemble $E$ des fbf closes, l'est aussi pour la fbf close $A$. En général il y a une infinité d'interprétations et donc l'approche sémantique ne peut pas s'appliquer. Sauf si on arrive à réduire de manière drastique le nombre d'interprétations à prendre en considération lors d'une procédure de déduction.

Une méthode pour effectuer cette reduction a été introduite par Löwenheim en 1915. Elle était fondée sur la possibilité de substituer dans une formule logique les variables par des constantes, de sorte que la formule ne contenait plus de variables. Herbrand en 1930 a élaboré une extension de cette méthode, pour faire la démonstration des théorèmes du point de vue syntaxique. Dans ce chapitre nous présentons d'abord la méthode de Herbrand. On continue avec la présentation de l'unification et on termine avec la discussion du modèle minimal de Herbrand.

## 10.2 Interprétations de Herbrand

On rappelle une définition que nous avons vu en calcul des prédicats et on en profite pour faire une extension.

DÉFINITION 10.2.1 *Une fbf sans variables est appelée* fbf filtrée.
*Un terme sans variables est appelé* terme filtré *(ou* terme de base *ou* terme clos*).*
*Une formule atomique ou atome sans variables est appelé* atome filtré *(ou* atome de base *ou* atome clos*).*

Comme on cherche à réduire le nombre d'interprétations possibles pour un ensemble donné des fbf, il est évident qu'il faut modifier notre fade voir l'univers du discours. Pour ce faire, nous allons d'abord construire un univers du discours à partir d'un langage de la logique du 1er ordre.

Soit $\mathcal{L} = (A, \mathbb{T}, \mathbb{F})$ un langage du premier ordre. Considérons un ensemble des fbf $E$ qui a comme langage sous-jacent $\mathcal{L}$, c'est-à-dire que ses prédicats, foncteurs, variables et constantes font partie de $\mathcal{L}$. Dans la suite on supposera toujours que $E$ est un ensemble des clauses sans éléments contradictoires et contenant seulement de la connaissance positive, c'est-à-dire contenant seulement des règles et des faits et ne contenant pas des questions. Un tel programme s'appellera *programme défini*. L'importance des programmes définis vient du fait qu'ils ont au moins un modèle. Pour établir ce modèle nous allons introduire l'univers et la base de Herbrand.

DÉFINITION 10.2.2 (UNIVERS DE HERBRAND) *Soit un ensemble de clauses $E$ qui a comme langage sous-jacent $\mathcal{L}$. L'*univers de Herbrand *$\mathcal{U}_E$ de l'ensemble $E$ est l'ensemble de tous les termes filtrés que l'on peut former en utilisant les constantes et les foncteurs de $E$.*
*Si $E$ ne contient pas un symbole de constante, alors on y ajoute arbitrairement un symbole de constante $a_H$ (la constante de Herbrand), afin de pouvoir former des termes filtrés.*

Ainsi défini, l'univers de Herbrand est composé des tous les termes filtrés que nous pouvons construire à partir de $E$.

EXEMPLE 10.2.1 *Calcul de l'univers de Herbrand pour les trois exemples suivants :*

*(1)* $E = \{p(a), \neg p(X) \vee q(X)\} \Rightarrow \mathcal{U}_E = \{a\}$.

*(2)* $E = \{\neg p(X) \vee q(X)\} \Rightarrow \mathcal{U}_E = \{a_H\}$ *où $a_H$ estla constante de Herbrand.*

*(3)* $E = \{p(a), \neg p(X) \vee q(s(X))\} \Rightarrow \mathcal{U}_E = \{a, s(a), s(s(a)), \ldots\} = \{s^n(a), n \geq 0\}$.

DÉFINITION 10.2.3 (Base de Herbrand) *Soit un ensemble de clauses $E$ qui a comme langage sous-jacent $\mathcal{L}$. La* base de Herbrand *$\mathcal{B}_E$ de l'ensemble $E$ est l'ensemble de tous les atomes filtrés que l'on peut former en utilisant les symboles de prédicats de $E$, appliqués aux termes filtrés de $\mathcal{U}_E$, en prenant ces derniers comme des arguments.*

On voit donc que la base de Herbrand est constitué de tous les prédicats filtrés que nous pouvons construire en utilisant les éléments de $E$.

De faprécise l'univers de Herbrand $\mathcal{U}_E$ se construit comme suit :

- À chaque constante $a$ de $E$ correspond la même constante dans $\mathcal{U}_E$.
- S'il n'existe aucune constante dans $E$, alors on introduit dans $\mathcal{U}_E$ la constante de Herbrand $a_H$.
- À chaque foncteur $f$ d'arité $n$, correspond dans $\mathcal{U}_E$ le même foncteur avec comme arguments des termes clos de $\mathcal{U}_E$ , e.g. $f(t_1, \ldots, t_n)$ , $t_i \in \mathcal{U}_E$.

Pour la construction de la base de Herbrand $\mathcal{B}_E$, nous avons

– À chaque prédicat $p$ d'arité $n$ correspond dans $\mathcal{B}_E$ le même prédicat avec comme arguments des termes clos de $\mathcal{U}_E$ , e.g. $p(t_1,\ldots,t_n)$ , $t_i \in \mathcal{U}_E$ [2].

Nous voyons ainsi que $\mathcal{U}_E$ et $\mathcal{B}_E$ sont définis pour un programme $E$ et contiennent exactement les symboles qui sont dans le programme.

EXEMPLE 10.2.2 *On calcule la base de Herbrand pour les programmes de l'exemple 10.2.1.*

*(1)* $\mathcal{B}_E = \{p(a), q(a)\}$

*(2)* $\mathcal{B}_E = \{p(a_H), q(a_H)\}$

*(3)* $\mathcal{B}_E = \{p(a), p(s(a)), \ldots, q(a), q(s(a)), \ldots\} = \{p(s^n(a)), q(s^n(a)); n, m \geq 0\}$

ASCÈSE 10.1 *Soit le programme*

$$E = \{plus(zero, N, N),\ plus(s(K), L, s(M)) \leftarrow plus(K, L, M)\}$$

*Déterminer l'univers et la base de Herbrand pour E.*

ASCÈSE 10.2 *Soit le programme*

$$E = \{\ impair\ (X) \rightarrow\ impair\ (s(s(X)))\}$$

*où $s(X)$ est le successeur de $X$.*
*Déterminer l'univers et la base de Herbrand pour E.*
*Quelle peut être la constante $a_H$ ?*

ASCÈSE 10.3 *Soit le programme*

$$\begin{array}{l} E = \{\ chante\ (chanteur\ (chanson)\ , chanson), \\ \qquad\quad chante\ (X\ , chanson)\ \rightarrow joyeux\ (X) \end{array}$$

*Déterminer l'univers et la base de Herbrand pour E.*

En suivant cette construction, on peut comprendre que l'univers de Herbrand est plus « pauvre » qu'un univers de discours tel que nous l'avions examiner au chapitre précédent, car il n'a pas des variables. Il est donc plus facile de calculer la valeur de vérité d'un ensemble de clauses $E$ et de savoir ainsi s'il est satisfiable ou pas, car, en absence des variables, d'une part il n'est pas nécessaire de procéder à une assignation de variables et, d'autre part, la signification des termes se simplifie. Néanmoins il faut savoir que l'univers de Herbrand peut être infini comme le stipule la proposition suivante :

PROPRIÉTÉ 10.2.1 *Si dans éléments d'un programme il y a un foncteur, alors l'univers de Herbrand correspondant est de cardinal infini.*

Il reste maintenant à établir la nouvelle forme de l'interprétation.

2. Il est temps de signaler que lors de deux définitions concernant l'univers et la base de Herbrand, nous avons pris quelques libertés avec la rigueur, afin de mieux faire comprendre au lecteur l'approche de Herbrand au calcul logique. En fait l'univers de Herbrand ne se construit pas sur un ensemble $E$ des formules mais sur le langage $\mathcal{L}$ du 1er ordre dont, d'aiilleurs, l'utilisation permet d'engendrer les formules qui sont dans $E$. Quand on travaille sur un ensemble de formules, on suppose que le langage $\mathcal{L}$ est défini par les constantes, les foncteurs et les prédicats qui apparaissent dans cet ensemble de formules. L'univers, donc, de Herbrand est construit sur ce langage. Cette mise au point étant faite, on pourra continuer à parler – mais seulement par abus de langage – de l'univers de Herbrand, de la base de Herbrand et de l'interprétation de Herbrand associés à l'ensemble de clauses $E$.

DÉFINITION 10.2.4 (Interprétation de Herbrand) *Soit un ensemble de clauses E qui a comme langage sous-jacent $\mathcal{L}$. L'* interprétation de Herbrand $I_E$ *de l'ensemble E est constituée par l'univers de Herbrand $\mathcal{U}_E$ comme domaine de l'interprétation et une application entre E et $\mathcal{U}_E$ qui associe :*

- *à chaque constante $c \in E$, la même constante $c_{I_E} = c \in \mathcal{U}_E$ ;*
- *à chaque foncteur $f$ de E d'arité n , le foncteur $f_{I_E}\left((t_1)_{I_E}, \ldots, (t_n)_{I_E}\right) = f(t_1, \ldots, t_n) \in \mathcal{U}_E$ ;*
- *à chaque prédicat $p$ de E d'arité n , une relation quelconque $p_{I_E} \subseteq \mathcal{U}_E^n$ [3] (i.e. une application quelconque de $\mathcal{U}_E^n$ dans l'ensemble $\{0,1\}$ - faux, vrai).*

Ainsi pour cette interprétation l'univers du discours est l'univers de Herbrand $\mathcal{U}_E$. Pour la sémantique des termes la signification d'une constante est la constante elle-même. Pour la signification des foncteurs on utilise la signification prédéfinie des foncteurs lors de l'interprétation. En somme pour toute interprétation les termes représentent des éléments de $\mathcal{U}_E$. Nous pouvons donc envisager une interprétation pour laquelle chaque terme représente exactement un élément du domaine de l'univers du discours et chaque élément du domaine est représenté. Enfin, en ce qui concerne l'interprétation d'un prédicat, il suffit de spécifier la relation à associer avec chaque symbole de prédicat, c'est-à-dire la valeur des arguments pour lesquelles le prédicat retourne la valeur "vraie". Dans la mesure où tous ces prédicats se trouvent, sous forme filtré, dans la base de Herbrand $\mathcal{B}_E$, il suffit d'en extraire un sous-ensemble $\mathcal{B}'_E$ des prédicats qui retournent la valeur "vraie" pour l'interprétation.

EXEMPLE 10.2.3 *Soit le programme $E = \{p(a), \neg p(X) \vee q(s(X))\}$. Considérons le sous-ensemble $A = \{p(a), p(s(a)), q(a), q(s(a))\}$. Une interprétation est la suivante : $I_E = \{F, V, V, F\} = \{\neg p(a), p(s(a)), q(a), \neg q(s(a))\}$.*

En conclusion, une interprétation de Herbrand peut s'identifier à un sous-ensemble de la base de Herbrand. En effet étant donné que, pour un ensemble de clauses $E$, l'interprétation des constantes et des foncteurs soit fixée, une interprétation de Herbrand est la détermination du sous-ensemble de la base de Herbrand $\mathcal{B}_E$ dont les éléments ont comme valeur de vérité 1 (vrai) pour cette interprétation. Réciroquement si on considère un sous-ensemble quelconque $B$ de la base de Herbrand $\mathcal{B}_E$ et si pour chaque prédicat de $E$ qui est aussi dans $B$ avec les mêmes arguments, on associe la valeur de vérité 1 (vrai), alors on obtient une interprétation de Herbrand issue de $B$. Nous avons donc que si $I_E$ est une inteprétation de Herbrand, alors elle est équivalente au sous-ensemble $\left\{B_i \in \mathcal{B}_E \;/\models_{I_E} B_i\right\} \subseteq \mathcal{B}_E$. L'interprétation, donc, de Herbrand est l'interprétation la plus générale que nous pouvons donné au langage $\mathcal{L}$.

DÉFINITION 10.2.5 (Modèle de Herbrand) *Soit un ensemble de clauses E qui a comme langage sous-jacent $\mathcal{L}$. Considérons une interprétation de Herbrand $I_E$ . Si cette interprétation est un modèle pour chaque clause de E, alors elle est un* modèle de Herbrand *pour E.*

Notons aussi que toute interprétation de Herbrand est une interprétation au sens de la définition donnée au chapitre précédent mais le contraire n'est pas toujours vrai.

ASCÈSE 10.4 *Donner quelques interprétations pour le programme de l'ascèse 10.1.*

3. qui est l'ensemble des tous les $n$-tuples de termes filtrés.

Grâce au théorème suivant nous avons que si un ensemble de clauses $E$ a un modèle, alors il a aussi un modèle de Herbrand.

THÉORÈME 10.2.1 *Soit $I'_E$ un modèle pour l'ensemble de clauses $E$. Alors l'ensemble $I_E = \left\{ B \in \mathcal{B}_E \;\middle/\; \models_{I'_E} B \right\}$ est un modèle de Herbrand pour $E$.*

La conséquence immédiate de ce théorème est que nous pouvons associer des interprétations de Herbrand avec des sous-ensembles de la base de Herbrand et décrire ainsi l'interprétation par les éléments de $\mathcal{B}_E$ qui sont vrais pour cette interprétation.

ASCÈSE 10.5 *Parmi les interprétations établies à l'ascèse 10.1, quelles sont celles qui sont un modèle pour $E$ ?*

Nous avons aussi les deux corollaires suivants :

COROLLAIRE 10.2.1 *Si un ensemble de clauses $E$ est satisfiable, alors cet ensemble possède un modèle de Herbrand.*

D'après ce corollaire, pour prouver qu'un ensemble de clauses $E$ est satisfiable, on doit chercher de trouver un modèle de Herbrand, c'est-à-dire il faut examiner les sous-ensembles de la base de Herbrand $\mathcal{B}_E$ de $E$. Nous voyons que de cette façon l'espace de recherche se réduit fortement.

COROLLAIRE 10.2.2 *Si l'ensemble de clauses $E$ n'a pas de modèle de Herbrand, alors il est insatisfiable.*

ASCÈSE 10.6 *Soit le programme*

$$E = \left\{ \begin{array}{l} p(f(f(a))). \\ p(f(X)) \leftarrow p(X). \end{array} \right\}$$

*Préciser parmi les trois interprétations suivantes, celles qui sont des modèles de Herbrand pour $E$ :*

*(1)* $I_1 = \{p(a)\}$

*(2)* $I_2 = \{p(f(f(a))), p(f(f(f(a)))), \ldots\}$

*(3)* $I_3 = \{p(f^n(a)) \mid n \text{ est premier}\}$

Nous pouvons donc obtenir le théorème suivant :

THÉORÈME 10.2.2 (THÉORÈME DE HERBRAND) *Un ensemble $E$ de clauses est insatisfiable si et seulement si n'a pas de modèle de Herbrand.*

Rappelons avant de terminer ce paragraphe que les résultats obtenus ici ne sont valables que si $E$ est un ensemble de clauses.

ASCÈSE 10.7 *Soit l'ensemble de fbf*

$$E = \{\neg p(a)\ ,\ \exists X\ p(X)\}$$

*Déterminer l'univers et la base de Herbrand pour $E$.*
*Donner les interprétations de Herbrand et les modèles de Herbrand.*
*Donner d'autres modèles de $E$.*

## 10.3 Modèle minimal de Herbrand

Nous avons vu à la fin du § 10.2 qu'un ensemble des clauses peut ne pas avoir un modèle de Herbrand. Le théorème suivant montre que tout modèle non-Herbrand d'un programme défini peut produire un modèle de Herbrand.

THÉORÈME 10.3.1 *Soit $E$ un programme défini*[4] *et $I'$ un modèle non-Herbrand de $E$. Alors l'ensemble*

$$I = \{A \in \mathcal{B}_E \,/\models_{I'} A\}$$

*est un modèle de Herbrand.*

Comme conséquence de ce théorème nous avons un fait déjà connu :

COROLLAIRE 10.3.1 *Soit $E$ un programme défini. La base de Herbrand $\mathcal{B}_E$ de $E$ est un modèle de Herbrand de $E$.*

Considérons une exemplification (instanciation) d'une clause de $E$ en utilisant uniquement des constantes $A \leftarrow B_1, \ldots, B_n$ avec $n \geq 0$. Nous avons que $B_i \in \mathcal{B}_E$, $i = 1, \ldots, n$ et $A \in \mathcal{B}_E$. Donc la clause entière est vraie. Bien évidemment ce modèle n'est pas très intéressant car il enfonce des portes ouvertes, c'est-à-dire il affirme que tout prédicat de $E$ peut être interprété comme une relation sur l'ensemble de termes filtrés de $E$. Ce qui serait ici intéressant est d'avoir des modèles qui contiennent davantage des termes filtrés que $E$. Avant d'examiner cette possibilité, nous allons établir la structure de la famille des modèles de Herbrand. Pour cela on rappelle que tout modèle de Herbrand est un sous ensemble de $\mathcal{B}_E$. Nous pouvons donc envisager, étant donnés différents modèles de Herbrand, leur intersection qui forme un autre modèle de Herbrand. On a donc le théorème suivant :

THÉORÈME 10.3.2 *Soient $E$ un programme défini, $\mathcal{M}_E = \{I_1, I_2, \ldots\}$ une famille non vide de modèles de Herbrand de $E$. L'intersection $I = I_1 \cap I_2 \cap \cdots$ est aussi un modèle de Herbrand pour $E$*[5].

La famille $\mathcal{M}_E$ existe pour tout $E$, car il y a au moins un modèle de Herbrand pour $E$ à savoir la base $\mathcal{B}_E$ de Herbrand. Tout modèle de Herbrand est un sous-ensemble de la base de Herbrand. Si $\mathcal{M}_E$ est la famille de tous les modèles de Herbrand de $E$, alors l'intersection $I = \bigcap_{I_i \in \mathcal{M}_E} I_i$ représente le *modèle minimal de Herbrand* (ou encore le *plus petit modèle de Herbrand*) pour $E$. Ce modèle sera noté par $\widetilde{I}_E$.

THÉORÈME 10.3.3 *Le modèle minimal de Herbrand $\widetilde{I}_E$ d'un programme défini $E$ est l'ensemble de toutes les conséquences logiques filtrées :*

$$\widetilde{I}_E = \{A \in \mathcal{B}_E \,/ E \models A\}$$

---

4. Un *programme défini* est un programme qui contient uniquement des *clauses définies*, c'est-à-dire des règles ou, encore, des *clauses de Horn strictes*, et aussi des faits, c'est-à-dire ce qu'on appelle des *clauses de Horn positives*.

5. Il faut se garder de toute interprétation abusive de ce théorème. En effet il est inexact d'affirmer que toute intersection de modèles pour un programme défini est un modèle.

Ce théorème nous montre que pour un programme défini donné $E$, le modèle minimal de Herbrand constitue une interprétation du programme. Mais il y a plus. On avait vu précédemment que les modèles de Herbrand sont, dans un certain sens, des interprétation les plus générales, c'est-à-dire des modèles les plus généraux. Donc le modèle minimal de Herbrand contient les clauses qui sont vraies pour chaque modèle de $E$, c'est-à-dire les clauses filtrés qui sont des conséquences logiques de $E$. On pourrait donc aller plus loin et suggérer qu'un programmeur aura tout intérêt à ce que l'interprétation qui fait de son programme soit exactement le modèle minimal de Herbrand.

Toutefois il faut se garder d'extrapoler ces résultats pour n'importe quelle fbf.

ASCÈSE 10.8 *Soit la fbf $p(a) \vee q(a)$. Trouver ses modèles de Herbrand. Vérifier qu'il n'y a pas de modèle minimal de Herbrand.*

L'existence d'un modèle minimal permet de structurer la famille de toutes les interprétations de $E$. Cette famille est en fait l'ensemble $\mathcal{P}(\mathcal{B}_E)$ des parties de $\mathcal{B}_E$. On peut démontrer que cet ensemble, muni de l'ordre partiel $\subseteq$ , est un treillis complet [6]. Pour ce treillis nous avons $\perp = \emptyset$ et $\top = \mathcal{B}_E$. Pour tout ensemble d'interprétations de Herbrand la borne inférieure est l'interprétation de Herbrand qui est l'intersection de toutes les interprétations de Herbrand de l'ensemble et la borne supérieure est la réunion de ces mêmes interprétations.

Pour une clause $A$ de $E$ on peut déterminer des substitutions particulières $\sigma_f$, appelées *substitutions filtrantes*, telles que $A\circ\ \sigma_f$ soient des clauses filtrées, c'est-à-dire sans occurrence des variables. En réalité $A\circ\ \sigma_f$ est une exemplification (instaciation) du fait $A$.

Considérons un élément du treillis, c-à-d. un modèle de Herbrand $I \in \mathcal{P}(\mathcal{B}_E)$.

Pour tout fait $A$ de $E$ nous avons que $A\circ\ \sigma_f$ appartient à $I$ et ceci pour toute substitution filtrante $\sigma_f$. Car sinon $A$ ne serait pas vrai dans $I$ et, par conséquent, $I$ ne serait pas un modèle pour $E$.

Soit maintenant une règle $A \leftarrow B_1, \ldots, B_n$ de $E$ avec $n > 0$ qui signifie, entre autre, que si les faits $B_1, \ldots, B_n$ sont vrais, il en de même pour $A$. Par conséquent si nous avons une substitution filtrante $\sigma_f$ pour $B_1, \ldots, B_n$, $A$ est vrai pour la même substitution filtrante, i.e. $A\circ\sigma_f \leftarrow B_1\circ\sigma_f, \ldots, B_n\circ\ \sigma_f$. Donc si $I$ contient $B_1\circ\sigma_f, \ldots, B_n\circ\ \sigma_f$, il doit aussi contenir $A\circ\sigma_f$ sinon il ne serait pas un modèle. Notons que $A\circ\sigma_f \leftarrow B_1\circ\sigma_f, \ldots, B_n\circ\ \sigma_f$ est une exemplification (instanciation) de la règle $A \leftarrow B_1, \ldots, B_n$ . Cette discussion permet de voir la manière dont nous pouvons enrichir en termes filtrés le programme $E$. En effet soit $I_1(E)$ l'ensemble des faits filtrés de $E$. Il est possible de lui adjoindre des nouveaux éléments en utilisant toutes les exemplifications (instanciations) des toutes les règles de $E$. On engendre ainsi l'ensemble $I_2(E)$ sur-ensemble de $I_1(E)$. Ce processus peut être répété tant qu'on peut engendrer des nouveaux éléments. Ces nouveaux éléments s'ajoutent à l'ensemble $I_k(E)$ afin de former l'ensemble $I_{k+1}(E)$ qui suit immédiatement $I_k(E)$.

Afin de formaliser ce processus on donne d'abord la définition suivante :

DÉFINITION 10.3.1 (Opérateur de la conséquence immédiate) *Soit $E$ un programme défini. L'application $\mathcal{T}_E : \mathcal{P}(\mathcal{B}_E) \rightarrow \mathcal{P}(\mathcal{B}_E)$ définie par*

6. Un *treillis* ou *ensemble réticulé* est un ensemble muni d'un ordre partiel et disposant d'un plus petit et d'un plus grand élément, notés $\perp$ et $\top$ respectivement. Le treillis est *complet* si pour chacun de ses sous-ensembles $X$ il y a une borne inférieure et une borne supérieure.

$$\mathcal{P}(\mathcal{B}_E) \ni I \mapsto \mathcal{T}_E(I) = \left\{ A\circ\sigma_f \,/\, A\circ\sigma_f \leftarrow B_1\circ\sigma_f,\ldots,B_n\circ\sigma_f \right.$$
$$\text{où } (A \leftarrow B_1,\ldots,B_n) \in E$$
$$\left. \text{et } \left\{B_1\circ\sigma_f,\ldots,B_n\circ\sigma_f\right\} \subseteq I \right\}$$

*c'est-à-dire $\mathcal{T}_E(I)$ est composé de toutes de conclusions filtrées $A\circ\sigma_f$ des règles $(A \leftarrow B_1,\ldots,B_n)$ de $E$ et telles que les prémisses filtrées $\left\{B_1\circ\sigma_f,\ldots,B_n\circ\sigma_f\right\}$ soient dans $I$.*

*$\mathcal{T}_E$ est appelé* opérateur de la conséquence immédiate *de $E$.*

Les propriétés de cet opérateur sont données par le

THÉORÈME 10.3.4 *Soit $E$ un programme défini. L'opérateur de la conséquence immédiate de $E$ a les propriétés suivantes :*

*(1) $\mathcal{T}_E$ est une application monotone, i.e. si $I' \subseteq I$ , alors $\mathcal{T}_E(I') \subseteq \mathcal{T}_E(I)$.*

*(2) $T_E$ est une application continue, i.e. si $I \in \mathcal{P}(\mathcal{B}_E)$ tel que $\sup(I) \in I$ , alors $\mathcal{T}_E(\sup(I)) = \sup(\mathcal{T}_E(I))$.*

*(3) Soit $I$ une interprétation de Herbrand. $I$ est un modèle de Herbrand si et seulement si $\mathcal{T}_E(I) \subseteq I$.*

ASCÈSE 10.9 *Soit le programme $P = \{p(f(X)) \leftarrow p(X), q(a) \leftarrow p(X)\}$. Calculer $\mathcal{T}_E(\emptyset)$, $\mathcal{T}_E(\mathcal{B}_E)$ et $\mathcal{T}_E(\mathcal{T}_E(\mathcal{B}_E))$.*

Pour pouvoir poursuivre la construction des ensembles $I_k(E)$ il est obligatoire d'introduire la notion des puissances ordinales de l'application $\mathcal{T}_E$.

Rappelons que les nombres ordinaux se déterminent de farécurrente comme suit :
Le premier nombre ordinal est $\emptyset$ et il est noté par 0, c'est-à-dire $0 = \emptyset$.
Ensuite nous avons :
$1 = \{\emptyset\} = \{0\}$ ;
$2 = \{\emptyset, \{\emptyset\}\} = \{0,1\}$ ;
$3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0,1,2\}$ ;
… …
$\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots\} = \{0,1,2,\ldots\}$ qui est l'ensemble des nombres naturels. $\omega$ est le plus petit ordinal infini. Si nous poursuivons, nous avons la suite des ordinaux infinis :
$\omega + 1 = \omega \cup \{\omega\}$ ;
$\omega + 2 = (\omega + 1) + 1$ ;
… …

Dans la suite on notera les ordinaux infinis par des lettres grecques $\alpha, \beta, \cdots$ .

Sur les ordinaux nous pouvons définir un ordre, noté « $<$ » comme suit : $\alpha < \beta$ si $\alpha \in \beta$. En général si $\alpha$ est un nombre ordinal, alors son successeur est l'ordinal $\alpha + 1 = \alpha \cup \{\alpha\}$. C'est le plus petit ordinal qui est plus grand que $\alpha$. Un ordinal $\alpha$ est un *ordinal limite* si'il n'est successeur d'aucun ordinal.

Il est indispensable, pour pouvoir opérer avec les ordinaux, de pouvoir introduire[7] pour ces nombres l'équivalent du principe d'induction de Peano pour les nombres naturels. Il s'agit de l'*induction transfinie* pour les ordinaux. Du fait qu'il n'existe pas l'ensemble de tous les nombres ordinaux (tout au plus nous pouvons parler de collection de tous les nombres ordinaux), nous ne pouvons pas avoir une définition générale pour l'induction transfinie. Nous utilisons donc des

7. au grand dam des constructivistes, dont je fait modestement partie.

définitions qui sont valables jusqu'à un certain ordinal. Il y a plusieurs formulations équivalentes de l'induction transfinie. Nous présentons la plus simple (cf. E. W. Beth, pp.372-373).

DÉFINITION 10.3.2 (Induction transfinie) *Considérons une propriété $p$. Supposons que nous avons démontré (ou admis) que pour tout ordinal $\alpha$ nous avons que*

$$\forall \beta \text{ ordinal } (\beta < \alpha \rightarrow p(\beta)) \rightarrow p(\alpha)$$

*alors tout ordinal $\alpha$ a la propriété $p(\alpha)$.*

Pour introduire les opérations d'addition et de multiplication sur les ordinaux il faut aussi pouvoir définir la *recursion transfinie* afin de pouvoir appliquer la notion du successeur de manière analogue à celle utilisée pour les nombres naturels.

DÉFINITION 10.3.3 (Recursion transfinie) *Supposons que nous avons une fonction $f$ qui associe à tout ensemble $X$ d'ordinaux, un ordinal $\alpha = f(X)$. Alors il existe une unique fonction $F$ qui à chaque ordinal $\alpha$ associe un ordinal $F(\alpha)$ et qui satisfait à la condition*

$$\forall\, F(\alpha) = f(X_\alpha)$$

*où $X_\alpha = \{F(\beta)/\beta < \alpha\}$.*

Nous pouvons maintenant envisager d'introduire l'opérateur de la conséquence immédiate aux nombres ordinaux.

Nous avons vu que $\emptyset \in \mathcal{P}(\mathcal{B}_E)$ . Nous convenons de noter

$\mathcal{T}_E \uparrow 0 = \emptyset$

Ensuite nous posons :

$\mathcal{T}_E \uparrow (\alpha+1) = \mathcal{T}_E(\mathcal{T}_E(\alpha))$ si $\alpha+1$ est l'ordinal successeur de $\alpha$, i.e. si $\alpha+1 = \alpha \cup \{\alpha\}$.

$\mathcal{T}_E \uparrow \omega = \bigcup \{\mathcal{T}_E \uparrow \alpha / \alpha < \omega\}$.

Cette construction permet d'utiliser l'opérateur de la conséquence immédiate avec des nombres ordinaux $\alpha$ à la place des ensembles $I_\alpha(E)$.

THÉORÈME 10.3.5 (Caractérisation du modèle minimal de Herbrand) *Soient $E$ un programme défini et $\widetilde{I}_E$ son modèle minimal de Herbrand. Alors*

**(i)** *$\widetilde{I}_E$ est un point fixe de l'opérateur de la conséquence immédiate : $\mathcal{T}_E\left(\widetilde{I}_E\right) = \widetilde{I}_E$.*

**(ii)** *$\widetilde{I}_E = \mathcal{T}_E \uparrow \omega$.*

ASCÈSE 10.10 *Donner le modèle minimal de Herbrand pour le programme de l'ascèse 10.1*

## 10.4 Unification

Avant de pouvoir examiner si un ensemble de clauses $E$ est satisfiable, il faut essayer de réduire le nombre de ses éléments. On cherchera donc de trouver dans $E$ des clauses qui bien qu'ayant des formes différentes, sont en réalité équivalents logiquement. On effectuera cette recherche en utilisant l'algorithme de l'unification.

DÉFINITION 10.4.1 *Soit $E = \{A_1, \ldots, A_n\}$ un ensemble de clauses. Une substitution $\sigma$ est un* unificateur *de $E$ ssi $A_1\sigma = \cdots = A_n\sigma$ .*
*Dans ce cas on dit que $E$* est unifiable *par $\sigma$.*

On voit ainsi que si $E$ est unifiable par $\sigma$, alors $E\sigma$ est réduit à un singleton.

ASCÈSE 10.11 *Calculer la substitution qui unifie les deux termes*

$p(X, a, f(X, a))$ *et* $p(b, U, V)$

ASCÈSE 10.12 *Calculer la substitution qui unifie les termes suivants :*

$p(X, f(Y), g(b))$ , $p(X, f(c), g(Z))$ , $p(X, f(c), g(U))$ , $p(X, f(V), W)$

DÉFINITION 10.4.2 *Soit $E = \{A_1, \ldots, A_n\}$ un ensemble de clauses. Un unificateur $\theta$ de $E$ est l'*unificateur le plus général (UPG) *si pour tout unificateur $\sigma$ de $E$ il existe une substitution $\tau$ telle que $\sigma = \theta \circ \tau$.*

On peut facilement constater que l'UPG est unique à une substitution de renommage près [8].

ASCÈSE 10.13 *Trouver l'UPG de termes suivants :*

$p(a, Y, Z)$ *et* $p(X, b, Z)$

ASCÈSE 10.14 *Trouver l'UPG de termes suivants :*

$p(X, f(Y))$ *et* $p(Z, f(Z))$

*Est-il unique ?*

L'algorithme de l'unification permet de calculer l'UPG d'un ensemble de clauses s'il existe, ou de signaler l'impossiblité de l'existence d'un tel unificateur. Avant de présenter cet algorithme, nous allons faire quelques remarques.

Les foncteurs et les prédicats sont de structures de la forme suivante : $r(s_1, \ldots, s_n)$ avec $r$ le nom de la structure et $s_i$ la i-ième sous-structure. Cette structure peut se représenter comme une liste $[r, s_1, \ldots, s_n] = [s_0, s_1, \ldots, s_n]$. Chaque $s_i$ représente soit un terme, soit un prédicat. La longueur de la structure est égale au nombre de termes dans la liste. Si on note par $\mid r \mid$ cette longueur, on a $\mid r \mid = n + 1$.

Il est évident que, d'une part, deux structures ne sont unifiables que si leur longueur est identique. D'autre part si deux structures peuvent s'unifier, alors soit au moins l'une de deux est une variable, soit elles ont le même nom. Et la même chose doit être valable pour leurs sous-structures correspondantes.

DÉFINITION 10.4.3 *Soient deux structures $u_1 = [s_0, s_1, \ldots, s_n]$ et $u_2 = [r_0, r_1, \ \ldots, r_n]$. L'* ensemble non apparié *de $u_1$ et $u_2$ est l'ensemble $D(u_1, u_2)$ d'un couple de structures, défini comme suit :*

– *Si $s_0$ et $r_0$ sont différents, alors $D(u_1, u_2) = \{u_1, u_2\}$ .*

8. Une substitution de renommage est une substitution du type $\sigma = (t_1/x_1, t_2/x_2, \ldots, t_n/x_n)$ où $t_1, t_2, \ldots, t_n$ sont des variables et $x_1, x_2, \ldots, x_n$ sont des variables distinctes.

– *Si $s_0$ et $r_0$ sont identiques et si les sous-structures $s_i$ et $r_i$ sont identiques pour $i = 1, \ldots k-1$, tandis que les sous-structures de $s_k$ et $r_k$ sont différentes, alors l'ensemble non apparié de $u_1$ et $u_2$ est*
$D_k(u_1, u_2) = \{t_1 = (s_k, \ldots, s_n), t_2 = (r_k, \ldots, r_n)\}$, *où $t_1$ et $t_2$ les termes respectifs de $u_1$ et $u_2$ commençant au k-ième rang.*

ASCÈSE 10.15 *Calculer les ensembles non appariés de deux structures suivantes :*

$u_1 = p(f(X), h(Y), a)$ *et* $u_2 = p(f(X), Z, a)$

Si $\theta$ est le UPG de $u_1$ et $u_2$ est si $\{t_1, t_2\}$ est un de leurs ensembles non appariés, alors $\theta$ est aussi un unificateur de $t_1$ et $t_2$. Par conséquent l'UPG de $u_1$ et $u_2$ est la composition de l'UPG $\sigma$ de $t_1$ et $t_2$ avec l'UPG de $u_1\sigma$ et $u_2\sigma$.

Nous présentons l'algorithme de l'unification pour un ensemble de deux structures $u_1$ et $u_2$.

(1) Si $u_1 = u_2$ , alors l'UPG de $u_1$ et $u_2$ est $\sigma = \varepsilon$ (substitution identité). Fin.
Sinon, posons $\sigma = \varepsilon$ .

(2) Tant que $u_1\sigma \neq u_2\sigma$ faire

DÉBUT

(a) Trouver la première sous-structure de $u_1$ qui soit différente de la sous-structure correspondante de $u_2$. Soit $D_k(u_1, u_2) = \{t_1, t_2\}$ l'ensemble non apparié à ces deux sous-structures non identiques.

i. *(Échec normal)* Si ni $t_1$ ni $t_2$ n'est une variable, alors sortie en échec.

ii. *(Vérification d'occurrence)* Si l'un de $t_1, t_2$ est une variable contenue dans l'autre terme, alors sortie en échec.

iii. *(Passage à l'itération suivante)* Sinon, supposons que $t_1$ soit une variable. Alors on pose $\sigma = \sigma \circ \tau$ où $\tau = (t_1/t_2)$ (indépendamment de la nature de $t_2$).

FIN

On peut appliquer ce même algorithme à un ensemble contenant $m$ structures, de faitérative, à condition d'appliquer chaque substitution intermédiaire à la totalité des structures.

ASCÈSE 10.16 *Appliquer l'algorithme de l'unification aux deux termes suivants :*

$u_1 = p(f(X, Y), a)$ *et* $u_2 = p(f(Z, Z), Z)$

La validité de cet algorithme est donnée par le théorème suivant :

THÉORÈME 10.4.1 (Th. de l'Unification) *Soit E un ensemble fini de clauses. Si E est unifiable, alors l'algorithme de l'unification termine en donnant l' UPG pour E. Sinon l'algorithme se termine en échec.*

À cause de l'étape de la vérification d'occurrence, l'algorithme de l'unification risque d'être très coûteux en temps, car le temps de l'exécution de cette étape peut être une fonction exponentielle de la longueur de l'entrée. C'est la raison pour laquelle le langage `Prolog` utilise l'algorithme de l'unification sans l'étape de la vérification d'occurrence. Du point de vue théorique c'est une catastrophe. Du point de vue pratique on n'évite pas l'existence des cercles vicieux. Supposons, par exemple, qu'on cherche à unifier $X$ et $f(X)$. Alors on obtient la substitution $\tau = (X/f(X))$. Bien évidemment $\tau$ n'est pas un unificateur car $X \circ \tau = f(X) \neq f(X) \circ \tau = f(f(X))$. Le deuxième appel récursif fournit la même substitution qui n'est pas, pas plus qu'avant, un unificateur, car $f(X) \circ \tau = f(f(X)) \neq f(f(X)) \circ \tau = f(f(f(X)))$. Nous arriverons ainsi à une substitution infinie $f(f(f(\cdots)))$.

## 10.5 Réponse correcte à un programme

Nous terminons ce chapitre par la présentation de la réponse correcte à un programme.

DÉFINITION 10.5.1 *Soient $E$ un programme défini, $B$ un but $\leftarrow B_1, \ldots, B_n$. Une réponse à $E \cup \{B\}$ est une substitution $\sigma_{E,B}$ pour les variables du but $B$.*

Il va de soi que cette substitution ne concerne pas obligatoirement toutes les variables de $B$. De plus si $B$ n'a pas de variables, la seule réponse possible est la substitution identité.

DÉFINITION 10.5.2 *Soient $E$ un programme défini et $B$ un but $\leftarrow B_1, \ldots, B_n$ et une réponse $\sigma_{E,B}$ à $E \cup \{B\}$. On dit que $\sigma_{E,B}$ est une réponse correcte à $E \cup \{B\}$ si $B_k \circ \sigma_{E,B}$ est une conséquence logique de $E$ pour tout $k = 1, \ldots, n$ , i.e.*

$$E \models B_k \circ \sigma_{E,B} \ , \ \forall\, k = 1, \ldots, n$$

Remarquons qu'un programme au lieu de retourner une substitution comme réponse à un but, peut retourner la réponse « non ». Cette réponse est correcte si c'est le programme $E \cup \{\neg B\}$ (et non pas $E \cup \{B\}$) qui est satisfiable.

En utilisant le th. 10.3.2 on peut montrer qu'une substitution $\sigma_{E,B}$ est une réponse correcte si et seulement si $B_k \circ \sigma_{E,B}$ , est vrai pour tout $k = 1, \ldots, n$ dans $\widetilde{I}_E$.

THÉORÈME 10.5.1 *Soient $E$ un programme défini, $B$ un but $\leftarrow B_1, \ldots, B_n$ et une réponse $\sigma_{E,B}$ à $E \cup \{B\}$ telle que $B_k \circ \sigma_{E,B}$ soit filtré pour tout $k = 1, \ldots, n$. Alors les propositions suivantes son équivalentes :*

**(i)** *$\sigma_{E,B}$ est correcte.*

**(ii)** *$B_k \circ \sigma_{E,B}$ , est vrai pour tout $k = 1, \ldots, n$ dans tout modèle de Herbrand de $E$.*

**(iii)** *$B_k \circ \sigma_{E,B}$ , est vrai pour tout $k = 1, \ldots, n$ dans $\widetilde{I}_E$.*

Remarquons que ce théorème est vrai si $B_k \circ \sigma_{E,B}$ soit filtré pour tout $k = 1, \ldots, n$.

En récapitulant nous pouvons dire qu'il y a trois étapes lors de la construction d'un programme :

(1) Trouver un modèle de Herbrand à partir des modèles non-Herbrand.

(2) Trouver le modèle minimal de Herbrand.

(3) Extraire toute la connaissance positive qui est incluse dans le modèle minimal de Herbrand afin de l'utiliser pour la construction du programme. En effet tous les éléments d'un programme qui se vérifient, font partie du modèle minimal de Herbrand.

ASCÈSE 10.17 *Soit le programme* $E = \{p(a) \leftarrow\}$. *Considérons le but* $B = \{\leftarrow p(X)\}$. *Montrer que la substitution identité n'est pas une réponse correcte à* $E \cup \{B\}$.

## 10.6 Exercices

EXERCICE 10.1 *Donner l'univers et la base de Herbrand pour le programme suivant :*

$$q(X, g(X)) \rightarrow p(f(X)) \ , \rightarrow q(a, g(b)) \ , \ \rightarrow q(b, g(b))$$

EXERCICE 10.2 *Même chose pour le programme*

$$p(X, Y, Z) \rightarrow p(s(X), Y, s(Z)) \text{ et } p(0, X, X)$$

EXERCICE 10.3 *Supposons que les constantes d'un univers de Herbrand sont* $a, b, c$ *et* $d$.
*Soit* $I$ *l'interprétation de Herbrand :*

$$\{p(a), p(b), q(a), q(b), q(c), q(d)\}$$

*Trouver parmi les fbf suivantes, celles qui sont vraies pour l'interprétation* $I$ *:*

*(1)* $\forall X \ p(X)$

*(2)* $\forall X \ q(X)$

*(3)* $\exists X \ (q(X) \wedge p(X))$

*(4)* $\forall X \ (q(X) \rightarrow p(X))$

*(5)* $\forall X \ (p(X) \rightarrow q(X))$

EXERCICE 10.4 *Vérifier sur un exemple que la relation « la substitution* $\sigma$ *est plus générale que la substitution* $\tau$ *» n'est pas antisymétrique.*

EXERCICE 10.5 *Soit* $\sigma$ *une substitution de renommage. Montrer que dans ce cas* $\sigma$ *dispose d'une substitution inverse, notée* $\sigma^{-1}$, *telle que* $\sigma^{-1} \circ \sigma = \sigma \circ \sigma^{-1} = \varepsilon$ .

EXERCICE 10.6 *Soit* $\theta$ *l' UPG des termes* $s$ *et* $t$ *et* $\sigma$ *une substitution de renommage. Montrer que* $\theta \circ \sigma$ *est aussi un UPG de* $s$ *et* $t$.

EXERCICE 10.7 *Trouver tous les ensembles non appariés de deux structures suivantes :*

$$u_1 = p(X, f(g(a), b), c) \text{ et } u_2 = p(a, f(Y, b), Z)$$

EXERCICE 10.8 *Appliquer l'algorithme de l'unification aux programmes suivants :*

*(1)* $u_1 = p(f(a), g(X)) \text{ et } u_2 = p(Y, Y)$

*(2)* $u_1 = p(a, X, h(g(Z))) \text{ et } u_2 = p(Z, h(Y), h(Y))$

*(3)* $u_1 = p(X, X) \text{ et } u_2 = p(Y, f(Y))$

EXERCICE 10.9 *Soit le programme* $E = \{impair(s(0)) \leftarrow,\ impair(s(s(X))) \leftarrow impair(X)\}$. *Calculer* $\widetilde{I}_E$.

EXERCICE 10.10 *Soit le programme* $E = \{p(f(X)) \leftarrow p(X),\ q(X) \leftarrow p(X)\}$. *Calculer* $\widetilde{I}_E$.

# Table des matières