UNIFICATION

5.1
Rappels

Soient $\mathcal{L}$ un langage et E ensemble des clauses sur $\mathcal{L}$. On a

- ***Univers de Herbrand*** $\mathcal{U}_E$ est formé par les constantes et les foncteurs de E ayant comme arguments des termes de E.
- ***Base de Herbrand*** $\mathcal{B}_E$ est formé par les prédicats de E ayant comme arguments les éléments de $\mathcal{U}_E$.

Par conséquent ***L'univers de Herbrand contient tous les termes filtrés que nous pouvons construire à partir de*** E et ***La base de Herbrand contient tous les prédicats filtrés issus de*** E**.**

L'interprétation I_E ***de Herbrand*** de l'ensemble E utilise comme univers du discours l'univers de Herbrand.

THÉORÈME 6.0.14 *Si I'_E est un modèle pour l'ensemble des clauses E, alors $\left\{B \in \mathcal{B}_E \,\middle/\, \models_{I'_E} B\right\}$ est un modèle de Herbrand pour E.*

Par conséquent on peut associer des interprétations de Herbrand avec des sous-ensembles de la base de Herbrand et obtenir un modèle de Herbrand.

Nous avons la proposition suivante

PROPOSITION 6.0.1 *Si un ensemble des clauses est satisfiable, alors il est un modèle de Herbrand.*

Par conséquent tout modèle de Herbrand de l'ensemble des clauses E est un sous-ensemble de la base de Herbrand $\mathcal{B}_E$ et la base de Herbrand est un modèle de Herbrand de E.

On obtient ainsi le ***modèle minimal de Herbrand*** pour E :

$$\widetilde{I_E} = \bigcap_{I_i \in \mathcal{B}_E} I_i$$

et on a le

THÉORÈME 6.0.15 *Le modèle minimal de Herbrand $\widetilde{I_E}$ d'un programme défini E est l'ensemble de toutes les conséquences logiques filtrées :*

$$\widetilde{I_E} = \left\{ A \in \mathcal{B}_E \,/\, E \models A \right\}$$

De ce qui précède, on en déduit que :

– Le modèle minimal est une interprétation du programme E.

– Le modèle minimal contient les clauses qui sont vérifiées pour chaque modèle de E.

– Tout ce que un programme peut faire se trouve dans le modèle minimal et vice-versa.

Pour trouver le modèle minimal, il faut une méthode de recherche dans la base de Herbrand. Cette méthode est fondée sur l'opérateur de la conséquence immédiate qui permet en commençant par l'ensemble vide de trouver de façon itérative, tous les éléments du modèle minimal de Herbrand. La définition de cet opérateur, $\mathcal{T}_E : \mathcal{P}(\mathcal{B}_E) \to \mathcal{P}(\mathcal{B}_E)$ est la suivante :

Première application de l'opérateur :

$$\mathcal{P}(\mathcal{B}_E) \ni I = \{\emptyset\} \mapsto \mathcal{T}_E(I) = \left\{ A \circ \sigma_f \,/\, A \in E \right\}$$

Applications suivantes :

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

L'utilité de cet opérateur provient du théorème suivant :

THÉORÈME 6.0.16 (CARACTÉRISATION DU MODÈLE MINIMAL) *Pour tout programme défini E le modèle minimal de Herbrand $\widetilde{I_E}$ satisfait aux deux propriétés suivantes :*

– *Le modèle minimal est le point fixe de l'opérateur de la conséquence immédiate*

$$\mathcal{T}_E\left(\widetilde{I_E}\right) = \widetilde{I_E}$$

– *Si l'opérateur n'a pas de point fixe, on prend comme modèle minimal la limite de l'opérateur à l'infini :*

$$\widetilde{I_E} = \lim_{n \to \infty} \mathcal{T}_E^n(\emptyset)$$ [1]

1. Notons que cette écriture n'est pas rigoureuse. Pour une écriture correcte, se reporter au poly, théorème 5.3.5, page 74

5.2

Le principe de l'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 6.0.31 *Soit $E = \{A_1, \ldots, A_n\}$ un ensemble de clauses. Une substitution σ est un* unificateur *ou* unifieur] de E ssi $A_1\sigma = \cdots = A_n\sigma$.
Dans ce cas on dit que E est unifiable par σ.

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

EXEMPLE 6.0.6 *Soient les deux termes*

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

La décomposition en listes de deux fbf ci-dessus est $u_1 = [p\,[]\,X, a, [f\,[]\,X, a]]$ *et* $u_2 = [p\,[b, U, V]]$.
En appliquant l'algorithme on obtient la substitution $\sigma = (X/b, U/a, W/f(b, a))$.

On peut définir l'*unificateur (ou unifieur) le plus général (UPG)* θ tel que s'il existe un autre unificateur σ de E, alors il existe une substitution τ avec $\sigma = \theta \circ \tau$. On peut facilement constater que l'UPG est unique à une substitution de renommage près [2].

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.

Pour appliquer cet algorithme on considère que les foncteurs et les prédicats sont de structures de liste de la forme suivante :
$[s_1, [s_1, \ldots, s_n]$ avec s_1 le nom de la structure et s_i la i-ième sous-structure. 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 $| s |$ cette longueur, on a $| s | = n$.

DÉFINITION 6.0.32 *Soient deux structures $u_1 = [s_0, s_1, \ldots, s_n]$ et $u_2 = [r_0, r_1, \ \ldots, r_n]$. L'* ensemble des éléments non appariés *(ou* ensemble des éléments discordants *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 $D(u_1, u_2) = \{u_1, u_2\}$.*
- *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.

---

2. 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.

5.3

L'algorithme de l'unification

L'algorithme de l'unification pour un ensemble de deux fbf p_1 et p_2 est le suivant :

Soient $u_1 = [s_k, \ldots]$ et $u_2 = [v_k, \ldots]$ les listes correspondantes à la décomposition de deux fbf.

(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

Supposons que le k-ième élément de la liste $u_1\sigma$ est différent de l'élément correspondant de la liste $u_2\sigma$. On pose $u_1 = [s_k, \ldots] = \left[s_1^k, \ldots\right]$ et $u_2 = [v_l, \ldots] = \left[v_1^l, \ldots\right]$

(a) *(Échec normal)* Si ni s_1^k, ni v_1^l n'est une variable, alors sortie en échec.

(b) *(Vérification d'occurrence)* Si l'un de s_1^k, v_1^l est une variable contenue dans l'autre terme, alors sortie en échec.

(c) *(Passage à l'itération suivante)* Sinon, supposons que s_1^k soit une variable. Alors on pose $\sigma = \sigma \circ \tau$ avec $\tau = \left(s_1^k / v_1^l\right)$ (indépendamment de la nature de v_1^l).

FIN

On peut appliquer ce même algorithme à un ensemble contenant m fbf, de façon itérative, à condition d'appliquer chaque substitution intermédiaire à toutes les listes issues des fbf.

La validité de cet algorithme est donnée par le théorème suivant :

THÉORÈME 6.0.17 (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 τ 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 $f(f(f(\cdots)))$.

EXEMPLE 6.0.7 *On reprend l'exemple précédent :*

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

On pose $\sigma = \varepsilon$.

1ère étape : $u_1 = [X,a,[f,[X,a]]]$ *et* $u_2 = [b,U,V]$. *On a* $\tau = (X/b)$ *et* $\sigma = \sigma \circ \tau$. *Donc* $u_1 \circ \sigma = [b,a,[f,[b,a]]]$ *et* $u_2 \circ \sigma = [b,U,V]$.

2e étape : $u_1 = [a,[f,[b,a]]]$ *et* $u_2 \circ \sigma = [U,V]$. *On a* $\tau = (U/a)$ *et* $\sigma = \sigma \circ \tau$. *Donc* $u_1 \circ \sigma = [a,[f,[b,a]]]$ *et* $u_2 \circ \sigma = [a,V]$.

3e étape $u_1 = [[f,[b,a]]]$ *et* $u_2 = [V]$. *On a* $\tau = (V/[f,[b,a]])$ *et* $\sigma = \sigma \circ \tau$. *Donc* $u_1 \circ \sigma = [a,[f,[b,a]]]$ *et* $u_2 \circ \sigma = [a,[f,[b,a]]]$.

4e étape $u_1 = u_2$.

UPG = $(X/b, U/a, V/[f,[b,a]])$.

5.4

Réponse correcte à un programme

DÉFINITION 6.0.33 *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 6.0.34 *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. **??** 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 6.0.18 *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 $k = 1,\ldots,n$.

5.5

Les trois étapes pour l'écriture 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.

5.6

Exercices de Logique Computationnelle

EXERCICE 6.1 *Appliquer l'algorithme de l'unification aux programmes suivants :*

(1) $u_1 = p(f(a), g(X))$ *et* $u_2 = p(Y, Y)$

(2) $u_1 = p(a, X, h(g(Z)))$ *et* $u_2 = p(Z, h(Y), h(Y))$

(3) $u_1 = p(X, X)$ *et* $u_2 = p(Y, f(Y))$

EXERCICE 6.2 *Soit le programme* $E = \{impair(s(0)) \leftarrow,\ impair(s(s(X))) \leftarrow impair(X)\}$. *Calculer* $\widetilde{I_E}$.

5.7

Utilisation de l'unification par Prolog

Deux prédicats peuvent s'unifier si

(1) leurs noms sont identiques, et

(2) les variables de ces deux prédicats peuvent être substituées par des termes de sorte que, après substitution, les deux prédicats sont identiques.

Cette définition peut être appliquée à plus de deux prédicats.
Exemple : Soit le programme Prolog composé par le fait

```
\texttt{date(10, X, 2007).}
```

Alors à la question

```
\texttt{?-date(Jour, Mois, Annee).
```

Prolog repondra

```
\texttt{Jour = 10,}
\texttt{Annee = 2007}
```

c'es-à-dire il a fait la substitution suivante : `(Jour/10, X/Mois, Annee/2007)`. Comme la substitution `(X/Mois)` est une substitution de renommage (d'une variable par une autre variable), Prolog nous épargne de son affichage.

Remarquons que le fait est incomplet car à la place du mois il contient une variable, mais Prolog n'a pas été empeché pour fournir une réponse, bien sûr incomplète elle aussi.

5.8

Fonctionnement (un peu moins simplifié) de Prolog

De ce qui précède nous pouvons comprendre qu'un programme en Prolog est un ensemble des règles de Horn. Il s'agit soit des règles de Horn strictes du type `p` ← `q.` , soit des faits `a.` , qu'on peut assimiler à `a` ← `.` . La conséquence de la règle s'appelle *tête de la règle*.

Une question `?` ← `r.` est la formulation d'une hypothèse qui demande à être vérifiée par les règles du programme.

Pour ce faire, Prolog examinera les têtes des règles. Il retiendra les règles pour lesquelles leurs têtes peuvent s'unifier avec le prédicat de la question. S'il n'y a pas une telle règle, Prolog repondra par la négative. S'il existe au moins une telle règle, il ya deux possibilités :

(1) soit la règle en question est un fait et, donc, la réponse de Prolog serait immédiate et positive ;

(2) soit la règle en question est stricte. Dans ce cas Prolog fera les substitutions nécessaires pour unifier la question et la tête de la règle et il propagera ces substitutions aux hypothèses de la règle. Ces hypothèses, sous leur forme nouvelle après substitution, deviendront les nouvelles questions à vérifier à la place de la question du départ.

Exemple. Soit le programme

```
p(X) :- q(X).
q(a).
q(b).
```

Supposons qu'on pose la question

```
?- p(b).
```

Prolog fonctionnera comme suit :

(1) unifiera la tête de la règle `p(X) :- q(X).` avec la question à l'aide de la substitution `(X/b)` qui dès lors deviendra `p(b) :- q(b).` ;

(2) remplacera la question `?- p(b).` de départ par la question `?- q(b).` issue de l'hypothèse de la règle et recommencera ;

(3) trouvera le fait `q(b).` qui repond à la question posé, et

(4) donnera la réponse `X=b`.

5.9
Modèle minimal de Herbrand

Un programme en Prolog est un programme défini, c'est-à-dire un programme qui ne comporte pas des clauses de Horn négatives. L'intérêt d'un tel programme vient du fait qu'il a toujours un modèle, à savoir la base de Herbrand du programme. Par conséquent un programme défini ne peut pas être insatisfiable.

En réalité ce qui est intéressant ici est la connaissance du plus petit modèle de Herbrand que nous pouvons construire et que nous avons appelé modèle minimal de Herbrand. Cette connaissance permettra de vérifier ce que réellement fera le programme pendant son exécution.

Considérons, en effet, un programme en Prolog E et une clause p. Soit $\widetilde{I}_E$ le modèle minimal de Herbrand pour E. Alors nous avons $E \models p$ si et seulement si $p \in \widetilde{I}_E$. La partie "seulement si" implique que tout ce que un programme Prolog peut produire comme résultat se trouve dans le modèle minimal de Herbrand.

Exemple. Soit le programme

$$E = \left\{ \begin{array}{l} \texttt{boisson(toto,grenadine).} \\ \texttt{boisson(jojo,pastis).} \\ \texttt{boisson(lolo,pastis).} \\ \texttt{paire(X,Y) :-boisson(X,Z),boisson(Y,Z),X==Y.} \end{array} \right\}$$

L'univers de Herbrand est $\mathcal{U}_E = \{\texttt{toto}, \texttt{jojo}, \texttt{lolo}, \texttt{grenadine}, \texttt{pastis}\}$. La base de Herbrand est

$B_E = \{\texttt{boisson(jojo,jojo)}, \texttt{boisson(jojo,lolo)}, \texttt{boisson(jojo,pastis)}, \ldots,$
$\texttt{paire(jojo,jojo)}, \ldots\}$.

En utilisant l'opérateur de la conséquence immédiate, on a

- $\mathcal{T}_E \uparrow 0 = \emptyset$
- $\mathcal{T}_E \uparrow 1 = \{\texttt{boisson(toto,grenadine)}, \texttt{boisson(jojo,pastis)}, \texttt{boisson(lolo,pastis)}\}$
- $\mathcal{T}_E \uparrow 2 = \mathcal{T}_E \uparrow 1 \cup \{\texttt{paire(jojo,lolo)}, \texttt{paire(lolo,jojo)}\}$
- $\mathcal{T}_E \uparrow 3 = \mathcal{T}_E \uparrow 2$, c'est-à-dire $\mathcal{T}_E \uparrow 2$ est un point fixe pour l'opérateur de la conséquence immédiate. Par conséquent $\widetilde{I}_E = \mathcal{T}_E \uparrow 2$.

Nous pouvons vérifier, en utilisant Prolog, que les réponses données par le programme sont celles contenues dans le modèle minimal.

5.10

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 poupée 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]`.

Considérons 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` : `[` $T_1, T_2, \ldots, T_k$ `|` `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.

5.11

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.

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 jusqu'au moment où on aura atteint la liste vide dont la longueur est 0. On a donc le programme :

```
 longueur([],0).
longueur([X $\mid$ Y],N) :- longueur(Y,N1), 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.

5.12

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 :

```
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.

5.13
Exercices de Prolog

EXERCICE 6.3 *Considérons le graphe orienté dont la base de données correspondante en Prolog est*

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

(1) Écrire en Prolog le prédicat `chemin(X,Y)` *qui réussit s'il y a un chemin dans le graphe qui mène de* `X` *à* `Y`.

(2) Vérifier, en utilisant le modèle minimal de Herbrand, la validité de votre programme.

EXERCICE 6.4 *Calcul du factoriel* $n!$.

EXERCICE 6.5 *Calcul de la puissance d'un nombre* x^n.

EXERCICE 6.6 *Calcul des nombres de Fibonacci.*