Programmation Logique

Dans ce chapitre nous allons présenter le style de programmation le plus avancé dans le paradigme 'Non Procédural' (La programmation logique).
Un rappel sur le langage de la logique du 1er ordre est d'abord présenté, ensuite on décrit le fonctionnement des démonstrateurs automatiques de théorèmes utilisés en programmation logique et on termine avec une présentation résumée du langage Prolog. Dans cette dernière section on verra quelques exemples de programmes logiques manipulant des objets simples et complexes (décrits par des équations à point fixe).

1. Introduction

La programmation logique fait partie du style de la programmation non procédurale. Elle est aussi appelée programmation déclarative, car un programme logique se résume en une déclaration de certaines propriétés du problème à résoudre. La description de ces propriétés se fait à l'aide d'énoncés logiques (formules logiques).

L'exécution d'un programme logique est réalisée à l'aide d'un démonstrateur automatique de théorèmes. Ce dernier accepte en entrée un ensemble d'énoncés représentant les hypothèses du programme logique et tente de vérifier si à partir de cet ensemble d'énoncés, on peut déduire une formule donnée aussi en entrée (le but ou la question du programme logique). Durant ce processus de démonstration, le moteur d'inférence résoudra, par effet de bord, le problème décrit par le programme logique.

De ce fait, en programmation logique, le programmeur n'a pas à indiquer les étapes à suivre pour résoudre un problème, mais juste à spécifier certaines caractéristiques du problème, pour que le démonstrateur de théorème puisse être en mesure de résoudre seul et automatiquement le problème. C'est donc bien une caractéristique fondamentale de la programmation non procédurale qui est offerte par la programmation logique.

2. Rappel sur logique du premier ordre

Les éléments de base pour écrire des énoncés, utilisent des symboles de constantes, de fonctions et de prédicats.

2.1 Définition d'un terme

Les termes représentent les constituants de base pour former des formules.
Formellement un terme est défini par :

- toute constante est un terme
- toute variable est un terme
- si f est un symbole de fonction n-aires et t1, t2, … tn des termes, alors f(t1, t2, … tn) est un terme.

2.2 Définition d'un atome

Les atomes représentent les formules les plus simples.

- si p est un symbole de prédicat n-aires et t1, t2, … tn des termes, alors p(t1, t2, … tn) est un atome (ou formule atomique).

2.3 Définition d'une formule (énoncé)

Les formules bien formées représentent les énoncés du langage :

- Tout atome est une formule
- si A et B sont des formules, alors $A \vee B$, $A \wedge B$, $A \Rightarrow B$, $A \Leftrightarrow B$, $\forall x\ A$, $\exists x\ A$, $\neg A$, (A) sont aussi des formules

Souvent dans les énoncés, les quantificateurs universels ($\forall$) les plus englobant sont omis.

L'ensemble de tous les énoncés forme le langage de la logique du premier ordre.

Tout problème calculable (soluble par un ordinateur) peut être formulé dans ce langage.

2.4 Conséquence logique

On dit que l'ensemble de formules {A1, A2,... An } a pour conséquence logique la formule B (et on note A1, A2, … An |= B), si pour tout domaine possible, toute interprétation qui satisfait l'ensemble {A1, A2, … An }, satisfait aussi la formule B.

Le domaine est l'ensemble des valeurs que peuvent prendre les termes (constantes, variables et fonctions). Une interprétation d'une formule A, sur un domaine D, consiste à associé à chaque symbole de fonction n-aires (apparaissant dans A), une fonction de $D^n \rightarrow D$ et à chaque symbole de prédicat n-aires (apparaissant dans A), une fonction de $D^n \rightarrow \{V,F\}$.

De ce fait, dire que A1, A2, … An |= B, revient à dire que (de manière équivalente) :
- l'implication $A1 \wedge A2 \wedge \ldots An \Rightarrow B$ est valide, ou alors,
- l'ensemble {A1, A2,... An, ¬B } est inconsistant (contradictoire), c-a-d qu'il n'existe aucune interprétation qui vérifie en même temps toute les formules de l'ensemble {A1, A2,... An, ¬B }.

3. Démonstration automatique de théorèmes

Pour pouvoir montrer des conséquences logiques sans faire référence aux interprétations (difficilement automatisables), on utilise pour cela la notion d'inférence logique qui se base uniquement sur des manipulations syntaxiques des formules.

L'inférence logique est le processus qui permet de dériver un énoncé B à partir d'un ensemble d'énoncés S en appliquant une ou plusieurs règles d'inférence dans un système formel. On note alors cette dérivation par : S |-- B

Exemples de règles d'inférence :
modus ponens,
modus tollens,
système de Gentzen,
….

Dans ce qui suit on s'intéressera au système formel utilisant la règle de la résolution.

3.1 Clauses

La résolution s'applique à des formules écrites uniquement sous forme de clauses.
Une clause est une formule ayant la forme :
$a1 \vee a2 \vee \ldots an \vee \neg b1 \vee \neg b2 \vee \ldots \neg bm$
où les ai et les bj sont des atomes, les ai sont appelés les littéraux positifs et les bj sont appelés les littéraux négatifs.

La clause vide (notée $\Diamond$) n'a aucun littéral (ni positif ni négatif). Elle représente une contradiction et sera toujours évaluée à FAUX.

3.2 Substitution

Une substitution θ est un ensemble d'affectations de termes à des variables.
Exemple : $\theta = \{ x \leftarrow f(a,b),\ \ y \leftarrow c,\ \ z \leftarrow x \ \}$

Si E est une formule, alors θE est le résultat de l'application des affectations de θ dans la formule E. Le résultat est une formule qu'on appelle : instance substituée de E.
Exemple : Soit la clause $E = p(a,x) \vee q(y,f(x,y)) \vee r(b,c,x)$, si on applique à E la substitution θ donnée dans l'exemple précédent, on obtient la nouvelle formule :
$\theta E = p(a,f(a,b)) \vee q(c,f(f(a,b),c)) \vee r(b,c,f(a,b))$

3.3 Unificateur

Soient E1 et E2 deux clauses.
S'il existe une substitution θ telle que θE1 = θE2, alors θ est appelée : unificateur de E1 et E2.
On dit aussi que E1 et E2 sont unifiables.

Unification de 2 termes : Couple de désaccord

Deux termes t1 et t2 sont unifiables si :

- l'un d'entre eux est une variable ou alors,
- t1 et t2 sont le même symbole de constante, ou alors,
- t1 et t2 sont le même symbole de fonction.
  Dans ce dernier cas les termes argument du symbole de fonction doivent aussi être unifiables deux à deux.

Le couple de désaccord **D(t1, t2)** entre deux termes t1 et t2 est le couple de sous-terme les plus à gauche dans t1 et t2 qui ne soient pas identiques.

Exemple :
t1 = f(g(u), z)
t2 = f(g(h(t)), r(a))
donc D(t1, t2) =(u, h(t))

Unification de 2 termes : algorithme

Soient deux termes t1 et t2. L'algorithme qui suit cherche l'unificateur μ s'il existe.

1. $S_0 = \{\}$; $k \leftarrow 0$
2. SI (S_k t1 = S_k t2) $\mu \leftarrow S_k$; "Arrêt avec Succès" Aller à 4
   // Sinon …

3. Calculer D(S_k t1 , S_k t2) ; Soit (d1, d2) ce couple de désaccord.
   SI (d1 est une variable)
      $S_{k+1} \leftarrow S_k \cup \{ d1 \leftarrow d2 \}$;
      $k \leftarrow k+1$;
      Aller à 2
   SI (d2 est une variable)
      $S_{k+1} = S_k \cup \{ d2 \leftarrow d1 \}$;
      $k \leftarrow k+1$;
      Aller à 2
   Sinon "Arrêt avec Echec"
4. Fin

Exemple de déroulement de l'algorithme pour les termes :
   t1 = f(g(u), r(a))
   t2 = f(g(h(t)), z)
   avec : t,u et z des symboles de variables,
      a un symbole de constante et,
      f,g,h et r des symboles de fonctions.

$S_0 = \{\}$;
D = (u , h(t))
$S_1 = S_0 \cup \{ u \leftarrow h(t) \} = \{ u \leftarrow h(t) \}$
D = (r(a) , z)
$S_2 = S_1 \cup \{ z \leftarrow r(a) \} = \{ u \leftarrow h(t) , z \leftarrow r(a) \}$
S_2 est l'unificateur de t1 et t2.
L'instance substituée dans ce cas est t1 = t2 = f(g(h(t)), r(a))

Deux clauses E1 et E1 sont donc unifiables, si :
1- elles commencent par le même symbole de prédicat, et
2- les arguments (termes) des 2 clauses, s'unifient deux à deux.

Pour une paire de clauses (E1,E2), il peut exister plusieurs unificateurs possibles, on s'intéressera toujours à <u>l'unificateur le plus général</u> (donné par l'algorithme précédent), c-a-d celui imposant le moins de contraintes possible.

Exemples

E1	E2	θ	θE1=θE2
P(5)	P(5)	vide	P(5)
p(x)	p(5)	x ← 5	p(5)
p(6)	p(5)	pas d'unificateur	
p(x)	p(y)	x ← y	p(y)
p(x,x)	p(5,y)	x ← 5, y ← 5	p(5,5)
p(f(x),f(5),x)	p(z,f(y),y)	x ← 5, y ← 5, z ← f(5)	p(f(5),f(5),5)

3.4 Système formel de la résolution

On définit un système formel de preuve, manipulant des énoncés sous forme de clauses.

a) Alphabet = l'ensemble des symboles utilisés (cste, var, …) + { $\wedge$,$\vee$, $\neg$ }
b) Formules bien formées = l'ensemble des clauses
c) Axiome = vide
d) Règles = { res, dim }

- La règle 'res' (résolution) est définie comme suit :

  $E1, E2 \vdash_{(res)} E3$ ssi :

  E1 est de la forme $a \vee A$
  E2 est de la forme $\neg b \vee B$
  E3 est de la forme $\sigma(\theta A \vee B)$, E3 est appelée résolvante de E1 et E2.
  θ est une substitution de renommage de telle sorte que θE1 et E2 ne doivent avoir aucune variable commune.
  σ est l'unificateur le plus général entre θa et b

- La règle 'dim' (diminution) est une simplification faite au niveau d'une seule clause, elle est définie comme suit :

  $E1 \vdash_{(dim)} E2$ ssi :

  E1 est de la forme $a \vee b \vee A$
  E2 est de la forme $\sigma a \vee \sigma A$
  σ est l'unificateur le plus général entre a et b

Exemples :

1- $p(x,c) \vee r(x) , \neg p(c,c) \vee q(x) \vdash_{(res)} r(c) \vee q(x)$

2- $p(x, g(y)) \vee p(i(c), z) \vee r(x,y,z) \vdash_{(dim)} p(i(c), g(y)) \vee r(i(c), y, g(y))$

Détermination des résolvantes : (Explication)

E1 de la forme $X \vee Ai$ (Commutativité du $\vee$)
E2 de la forme $\neg Bj \vee Y$ (Commutativité du $\vee$)
$\sigma E1 = \sigma X \vee \sigma Ai$ (1)
$\sigma E2 = \neg\sigma Bj \vee \sigma Y$ (2)
de(1) : $\neg\sigma X \Rightarrow \sigma Ai$
de(2) : $\sigma Bj \Rightarrow \sigma Y$
Comme $\sigma Ai = \sigma Bj$ (σ unificateur entre Ai et Bj), par transitivité on : $\neg\sigma X \Rightarrow \sigma Y$
ou : $\sigma X \vee \sigma Y$
ou encore $\sigma(X \vee Y)$

Algorithme général de résolution (Stratégie par saturation)

La stratégie par saturation est l'un des nombreux algorithmes connus, d'application de la résolution.

Soit à démontrer la conséquence logique : $C_1, C_2, ..., C_n \models$ But
Il suffit alors de montrer l'inconsistance de l'ensemble A = { $C_1, C_2, \ldots C_n, \neg$But }

C_1, C_2, ...C_n et ¬But sont des clauses.

Pour ce faire, la stratégie par saturation consiste à enrichir l'ensemble A avec toutes les clauses pouvant être générées par les règles 'res' ou 'dim'. Si la clause vide (interprétée comme une contradiction) fait parti de l'ensemble ainsi construit, cela voudrait dire qu'il est inconsistant et donc que C_1, C_2, ..., C_n |== But.

A chaque étape i, toutes les clauses pouvant être générées à partir de l'ensemble de l'étape précédente (i-1) sont rajoutées.
S'il y a une clause vide, on s'arrête avec succès (on a réussi à montrer l'inconsistance de A donc la conséquence logique C_1, C_2, ..., C_n |== But a été démontrée).
S'il n'y a aucune nouvelle clause pouvant être générée, on s'arrête avec échec (l'ensemble A étant consistant, il n'y a pas de conséquence logique entre C_1, C_2, ..., C_n et But).
Dans tous les autres cas, on continu l'exploration des prochaines étapes.

1. i ← 0
2. A_0 ={C_1, C_2, ...C_n, ¬but}
3. A_{i+1} = A_i ∪ { Toutes les clauses obtenues par 'res' ou 'dim' à partir des clauses de A_i}
4. Si (A_{i+1} contient la clause vide)
   Arrêt, But est une conséquence logique de C_1,.., C_n
5. Si (A_{i+1} = A_i)
   Arrêt, But n'est pas une conséquence logique de C_1,.., C_n
6. i ← i+1 ; Aller à 3

Exemple :
Montrons l'inconsistance de l'ensemble de clauses {c1, c2, c3, c4 }
c1 = ¬p(x) ∨ q(x,f(x)) , c2 = ¬p(x) ∨ r(f(x)) , c3 = p(a) , c4 = ¬q(x,y)
avec :
p, q et r : des symboles de prédicats
x et y : des symboles de variables
a : un symbole de constante
f : un symbole de fonction
A0 = {c1, c2, c3, c4 }
A1 = A0 + {c5,c6,c7}
c5 = res(c1,c3) = q(a,f(a))
c6 = res(c1,c4) = ¬p(x)
c7 = res(c2,c3) = r(f(a))
A2 = A1 + {c8,c9}
c8 = res(c3,c6) = ◊ (la clause vide)
c9 = res(c4,c6) = ◊ (la clause vide)
Arrêt avec succès, car A2 contient la clause vide.
Donc l'ensemble de clauses de départ {c1, c2, c3, c4 } est inconsistant.

L'algorithme de la résolution générale (stratégie par saturation) est correct (1) et complet (2)

1. Correct : si l'algorithme par saturation s'arrête avec succès (c-a-d la clause vide a pu être générée) alors l'ensemble initial de clauses est forcément inconsistant.
2. Complet : si l'ensemble de départ est inconsistant, alors l'algorithme par saturation générera au bout d'un temps fini la clause vide, donc s'arrêtera avec succès.

Ne pas oublier que la logique du premier ordre est semi-décidable (Church 1936). Cela implique

que toute procédure de preuve, basée sur la logique du 1er ordre, hérite aussi de cette propriété.

Donc l'algorithme par saturation peut ne pas s'arrêter (boucle à l'infini) pour certains ensembles de départ consistants.

L'algorithme par saturation est généralement très inefficace (vaste espace de recherche).

Quelques cas particuliers de la résolution

A partir de (Non p) $\vee$ q et (Non q) $\vee$ r on peut déduire par résolution : (Non p) $\vee$ r
Écrites sous forme d'implications, on retrouve la règle du chaînage :
p $\Rightarrow$ q , q $\Rightarrow$ r |-- p $\Rightarrow$ r

A partir de (Non p) $\vee$ q et p , on peut déduire par résolution : q
Écrites sous forme d'implications, on retrouve la règle du modus ponens :
p $\Rightarrow$ q , p |-- q

A partir de (Non p) $\vee$ q et Non q, on peut déduire par résolution : Non p
Écrites sous forme d'implications, on retrouve la règle du modus tollens :
p $\Rightarrow$ q , Non q |-- Non p

Variante de la résolution pour les clauses de Horn

Une clause de Horn est une clause qui a au plus, un seul littéral positif.
On les classe en trois sous-catégories :

- implication :	a $\vee \neg$b1 $\vee \neg$b2 $\vee \ldots \neg$bm	(= a $\leftarrow$ b1 $\wedge$ b2 $\wedge \ldots$ bm)
- assertion :	a	(= a $\leftarrow$)
- dénégation :	$\neg$d1 $\vee \neg$d2 $\vee \ldots \neg$dm	(= $\leftarrow$ d1 $\wedge$ d2 $\wedge \ldots$ dm)

Si on se restreint uniquement à l'utilisation de clauses de Horn, il existe alors des algorithmes pour la résolution plus efficaces que celui par saturation, même si dans le cas général, leurs complexité restent toujours exponentielle.

En programmation logique, on utilise les implications et les assertions pour exprimer les propriétés du problème à résoudre (ce sont les clauses définies ayant exactement un seul littéral positif), et une dénégation pour poser une question sur le problème (c'est une clause négative, formée uniquement de littéraux négatifs).

L'interpréteur du langage logique (le démonstrateur automatique de théorèmes) utilise un algorithme particulier pour implémenter la règle de la résolution. La règle 'dim' (la diminution) n'étant pas utile dans le cas des clauses de Horn.

L'algorithme utilisé est appelé « résolution descendante », il se base sur le fait que la clause vide ne peut être trouvée qu'en explorant la branche de l'espace de recherche où la dénégation est utilisée comme une prémisse pour la résolution (c'est une des stratégies linéaires par entrée).

Le programme logique est donc composé de deux parties :
- les hypothèses du problème (assertions et implications)
- la question posée ou But à vérifier (transformée en une dénégation à réfuter)

On convient d'appeler les clauses de la partie hypothèses, des implications de la forme :

conséquent ← antécédents.

Une assertion peut être vue comme étant une implication particulière qui ne possède pas d'antécédents :

conséquent ←

La dénégation D = ¬D1 ∨ ¬D2 ∨ … ¬Dn, peut aussi être vue comme étant une conjonction de littéraux positifs en mettant le symbole de négation en facteur :

D = ¬ (D1 ∧ D2 ∧ … Dn) = ¬ (But)

Algorithme de la résolution descendante :

Cet algorithme cherche à rendre vide la dénégation D en effaçant tous ses littéraux D_i. On obtient ainsi une clause vide, donc un arrêt avec succès.
Si à un moment donné, un des littéraux D_i ne peut pas être effacé, on s'arrête avec échec (la clause vide ne peut pas être dérivée).

1- But = $D_1 \wedge D_2 \wedge \ldots D_n$
2- D = ¬ But = ¬ ($D_1 \wedge D_2 \wedge \ldots D_n$)
3- / Résolution d'une clause négative (issue de la dénégation initiale) avec les clauses des hypothèses */*
S'il n'existe aucune implication ou assertion dans les hypothèses, telle que le conséquent peut être unifié avec D_1 Alors ***Signaler 'échec'****.*
Sinon, soient C_1, C_2, … C_k les conséquents avec lesquels des unifications (θ_i) sont possibles
Pour i = 1, k
- Renommer les variables de la clause i de telle sorte qu'il n'y ait aucune variable commune avec celles de D.
- Remplacer dans D, le littéral D_1 par les éventuelles antécédents de C_i renommée,
- Appliquer l'unificateur utilisé (θ_i) à D on obtient ainsi une nouvelle dénégation D',
- Si D' est vide c-a-d D' = ¬ ( ), Alors ***Signaler 'Succès'***
Sinon Explorer récursivement cette nouvelle branche (Aller à 3 avec D ← D')
Fsi
FP
Fsi

Cet algorithme explore en profondeur l'espace de recherche des résolvantes issues de la dénégation initiale, jusqu'à trouver la clause vide. Il est correct mais pas complet, car l'exploration en profondeur risque d'emprunter une branche infinie dans une des itérations de la boucle 'Pour', avant d'explorer d'autres alternatives pouvant mener vers la clause vide.

Exemple simple sans montrer les unifications :

Prouver la question (le But) : A1 ∧ A2.
avec comme partie Hypothèses les 4 implications suivantes :

f0 : A1 ← B1 ∧ B2.
f1 : B1 ←
f2 : B2 ← B1 ∧ A2
f3 : A2 ←

Il suffit donc de réfuter Non(A1 ∧ A2), c'est la dénégation initiale (D) :

num étape	Dénégations	résolution effectuée :
1	Non(A1 ∧ A2)	(dénégation initiale)
2	Non(B1 ∧ B2 ∧ A2)	(res avec f0, A1 est remplacé par B1 et B2)
3	Non(B2 ∧ A2)	(res avec f1, B1 est remplacé par vide)
4	Non(B1 ∧ A2 ∧ A2)	(res avec f2, B2 est remplacé par B1 et A2)
5	Non(A2 ∧ A2)	(res avec f1, B1 est remplacé par vide)
6	Non(A2)	(res avec f3, A2 est remplacé par vide)
7	Non( )	(res avec f3, A2 est remplacé par vide)

Arrêt avec succès car la dernière dénégation est vide (la clause vide).

L'ensemble des unifications (θ_i) effectuées dans une branche menant vers un succès (la clause vide), représentent finalement une séquence d'affectations ayant été exécutées par le démonstrateur de théorèmes. Cette séquence d'affectations est donc un effet de bord de la procédure de preuve, ayant permis de résoudre effectivement le problème initial.

4. Le langage Prolog

Un programme Prolog est constitué d'une part, par un ensemble d'hypothèses sous forme d'assertions et d'implications, terminées par un point ('.'). D'autre part, une question (ou but) est posée pour lancer l'exécution du programme. La question doit être sous forme conjonctive (des littéraux positifs connectés par le ET logique).

,	veut dire : le ET logique
;	veut dire : le OU logique
:-	veut dire : impliqué par ou « Si »

4.1 Objets manipulés en Prolog

Les objets manipulés par Prolog sont :

- Objets élémentaires (constantes , variables)
- Listes d'objets : [obj1, obj2, … objn]
- Objets structurés : identificateur(obj1, obj2, … objm)

Les variables en Prolog sont des identificateurs commençant soit par une lettre majuscule soit le caractères souligné ('_'). En Prolog, _ (le caractère souligné seul) indique une variable anonyme. Chaque occurrence de _ correspond à une variable différente, même au sein d'une même clause. Partout où une variable est utilisée une seule fois dans la définition d'une clause, il est préférable d'utiliser la variable anonyme.

Les constantes sont soit numériques soit des identificateurs commençant par une lettre minuscule. On peut aussi avoir des constantes de type chaîne de caractères (quelconques) à condition de mettre des guillemets au début et à la fin de la chaîne.
Les constantes ont une portée globale, alors que les variables sont locale à chaque clause où elles apparaissent.

4.2 Fonctionnement du démonstrateur automatique de Prolog

Prolog implémente la stratégie de la résolution descendante avec un parcours en profondeur de l'espace de recherche. C'est à dire, si le premier littéral de la dénégation peut s'unifier avec plusieurs conséquents des hypothèses, les différentes alternatives sont alors explorées en profondeur d'abord, dans l'ordre d'écriture des implications dans le programme source. Cette stratégie n'est donc pas complète, car il est possible qu'une alternative soit explorée en premier alors qu'elle mène vers une branche infinie de l'espace de recherche. Les autres alternatives pouvant éventuellement mener vers la clause vide ne seront donc pas visitées. Ainsi il est possible que Prolog n'arrive pas à répondre à une question (l'évaluation boucle à l'infini) alors que la solution existe.

Pour des raisons d'efficacité et de simplicité, les opérations arithmétiques et les opérateurs relationnel usuels sont prédéfinie de même que la négation (Not). Il existe aussi des mécanismes pour contrôler le parcours en profondeur. Mais leur comportement n'est pas conforme à la programmation logique pure.

Lors de l'exécution d'un programme Prolog, si la question posée ne contenait pas de variables, le parcours de l'espace de recherche s'arrête dès qu'une clause vide est trouvée, auquel cas la réponse sera affirmative, ou alors lorsque toutes les alternatives auront été visitées sans succès (sans trouver la clause vide) et dans ce cas la réponse sera négative.

Dans le cas où la question posée contenait au moins une variable, Prolog tente de trouver toutes les valeurs que peut prendre la (les) variable(s) de la question pour arriver à un succès (clause vide). Il s'agit alors d'un parcours exhaustif de tout l'espace de recherche. A chaque succès, la (les) valeur(s) de cette (ces) variable(s) ayant(s) permis d'atteindre la clause vide, sera (seront) affichée(s).

Si par contre, la question contient des variables anonymes, leur valeurs ne seront pas affichées en cas de succès.

Exemple de programme Prolog simple :

```
1   Parent(aa, bb).          /* aa, bb, cc, dd, ee et ff sont des constantes */
2   Parent(bb, cc).
3   Parent(bb, dd).
4   Parent(dd, ee).
5   Parent(ff, dd).

6   Frere(X, Y) :- Parent(Z, X), Parent(Z, Y), Not(X=Y).      /* X, Y et Z sont des variables */

7   Ascendant(X,Y)  :- Parent(X, Y).
8   Ascendant(X,Y)  :- Parent(X, Z), Ascendant(Z,Y).


/*  On peut aussi écrire Ascendant comme suit :
    Ascendant(X,Y)  :- Parent(X, Y); Parent(X, Z), Ascendant(Z, Y).  */

Question : frere(dd, cc) /* Ici il n'y a pas de variable dans la question, donc  */
Réponse         Yes              /* le processus de recherche s'arrête dès la première solution */

Question : frere(X, dd)  /* Ici la variable X dans la question, veut dire :  */
Réponse         X = cc           /* quels sont tous les X tel que frere(X,dd) est vrai */
```

Question : frere(dd, X)
Réponse *X = cc*

Question : Ascendant(X, ee) */* Quels sont tous les ascendants de ee */*

Réponse
X = dd
X = aa
X = bb
X = ff

Question : Ascendant(X, cc)
Réponse
X = bb
X = aa

A la question Frere(X, dd), Prolog doit réfuter Non (Frere(X, dd)). C'est la première dénégation D. Dans le cas de succès, il imprime la valeur de X.
Soit donc à réfuter D = Non (Frere(X1, dd)), après le renommage automatique des variables.

Donnons quelques pas de résolution :

Frere(X1, dd) s'unifie avec la clause 6 (Frère(X, Y)) , l'unificateur le plus général est :
μ = { X ← X1 ; Y ← dd }
D devient Non (Parent(Z,X1), Parent(Z, dd), Not(X1=dd))
Parent(Z, X1) s'unifie avec les 5 premières clauses.
La première alternative dans l'unification est la clause 1(Parent(aa, bb)), l'unificateur est :
μ = { Z ← aa; X1 ← bb }
D devient Non (Parent(aa, dd), Not(bb=dd))
Comme Parent(aa, dd) ne peut s'unifier avec aucune clause, cette branche est abandonnée (retour arrière) pour tenter la prochaine alternative.
Parent(Z, X1) s'unifie avec la clause 2 (Parent(bb, cc))
μ = { Z ← bb; X1 ← cc }
D devient Non (Parent(bb, dd), Not(cc=dd))
Comme parent(bb,dd) s'unifie avec la clause 3 (parent(bb,dd)), elle sera alors effacé de D, car la clause 3 est une assertion (elle n'a pas d'antécédents).
La nouvelle dénégation D devient donc Non (Not(cc=dd))
Not(cc=dd) est automatiquement effacée par les règles prédéfinies du Not et de =
Ceci conduit à une dénégation vide, donc c'est un succès.
La valeur de X1 ayant mené vers la clause vide est alors affichée :
X1 = cc
Le parcours de l'espace de recherche continue pour trouver s'il y a d'autres valeurs de X1 :
Parent(Z, X1) s'unifie avec la clause 3 (Parent(bb, dd))
μ = { Z ← bb; X1 ← dd }
D devient Non (Parent(bb, dd), Not(dd=dd))
Parent(bb,dd) s'unifie avec avec la clause 3 est sera donc effacé.
Par contre Not(dd=dd) ne pourra pas être effacé car dd=dd est vrai (règles prédéfinies = et Not)
Cette branche est donc abandonnée, le retour arrière permettra d'explorer les deux autres alternatives qui restent , à savoir l'unification de Parent(Z,X1) avec les clauses 4 (Parent(dd,ee)) et 5 (Parent(ff, dd))
Avec la clause 4, la dénégation D devient : Non (Parent(dd, dd), Not(ee=dd)) qui mène vers un échec car Parent(dd,dd) n'est unifiable avec aucune clause.

Avec la clause 5, la dénégation D devient : Non (Parent(ff, dd), Not(dd=dd)) qui mène aussi vers un échec car Not(dd=dd) ne pourra pas être effacé par les règles prédéfinies de = et du Not.

4.3 Les listes en Prolog

Une liste de n objets est représentée par le terme [obj1, obj2, … objn]
La liste vide est représentée par le terme [ ]
Un opérateur particulier (noté |) permet de séparer le début d'une liste du reste de ses éléments (le reste des éléments et toujours une liste).

Ainsi :

X peut s'unifier avec le terme [a,b,c], l'unificateur est alors { X←[a,b,c] }
X peut s'unifier avec le terme [ ], l'unificateur est alors { X←[ ] }
[X,Y,Z] peut s'unifier avec le terme [a,b,c], l'unificateur est alors {X←a , Y←b , Z←c}
[X,Y,Z] ne peut pas s'unifier avec le terme [a,b]
[X|Y] peut s'unifier avec le terme [a,b,c], l'unificateur est alors { X←a , Y←[b,c] }
[X|Y] ne peut pas s'unifier avec le terme [ ]
[X|Y] peut s'unifier avec le terme [a], l'unificateur est alors { X←a , Y←[ ] }
[X,Y|Z] peut s'unifier avec le terme [a,b,c], l'unificateur est alors { X←a , Y←b , Z←[c] }

Exemple :

Définition d'un prédicat 'app(X,L)' qui sera évalué à vrai si X appartient à la liste L et faux sinon.

```
app( X, [X|L] ).            /* X appartient à toute liste commençant par X ([X|L]) */
app( X, [Y|L] ) :- app( X, L ).   /* si X appartient à la queue d'une liste (L), alors
                                     X appartient aussi à la liste globale  [Y|L] */
```

Comme la variable L n'apparaît qu'une seule fois dans la première clause, il est alors préférable (pour des raisons d'efficacité) de la remplacer par la variable anonyme (_).
De même la variable Y apparaît aussi qu'une seule fois dans la deuxième clause, on pourrait alors la remplacer aussi par la variable anonyme.

Ce qui donne, la nouvelle version :

```
app( X, [X|_] ).            /* X appartient à toute liste commençant par X */
app( X, [_|L] ) :- app( X, L ).   /* si X appartient à la queue d'une liste, alors
                                     X appartient aussi à la liste globale */
```

Exemple de la somme des éléments d'une liste :

```
somme( [ ] , 0 ).                /* la somme des éléments d'une liste vide est 0 */
somme( [X|L] ,S ) :- somme( L, M ) ,   /* si M est la somme des éléments de L, alors */
                     S = X+M.          /* X+M est la somme de la liste [X|L] */
```

4.4 Les objet structurés

Les objets structurés servent à représenter les termes composés d'un symbole de fonction ayant comme argument des objets quelconques.

Leur utilisation pourrait par exemple faciliter la manipulation d'objets complexes.

Par exemple, une date de naissance peut être représentée par un terme de la forme :
date(jour, mois, année)

On pourrait l'utiliser dans un prédicat qui représente des informations sur des personnes :
personne(nom, prenom, date(jj,mm,aa)).

Voici un exemple de programme Prolog définissant les personnes :

```
personne( aa, bb, date(18,05,1997) ).
personne( aa, cc, date(03,08,1972) ).
personne( dd, ee, date(26,11,1985) ).
personne( ff, gg, date(10,05,1981) ).
…
```

avec quelques questions :

Question : personne(X, Y, date(_,05,_))
/ quels sont les noms et prénoms des personnes nées un mois de mai */*
Réponse :
X = aa, Y = bb
X = ff, Y = gg

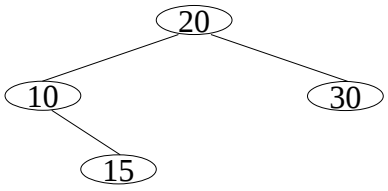
Question : personne(_, X, date(_,_,Y)) , Y < 1980
/ quels sont les prénoms des personnes nées avant 1980 */*
Réponse :
X = cc, Y = 1972

Question : personne(dd, ee , X)
/ quelle est la date de naissance de 'dd ee' */*
Réponse :
X = date(26,11,1985)

On utilise aussi les objets structurés pour représenter des structures de données complexes, particulièrement celles que l'on peut définir à l'aide d'équations à point fixe (les structures récursives).

Par exemple le type Arbre, représentant un arbre binaire d'entiers, peut être défini comme étant la plus petite solution de l'équation à point fixe suivante :
Arb = nil / a(entier, Arb, Arb)
un Arbre est soit le symbole nil, soit un terme structurés à 3 champs 'a(V, A1, A2)' où V est l'information de type 'entier' du nœud racine, alors que A1 et A2 sont respectivement le sous-arbre gauche et le sous-arbre droit.

Ainsi l'expression a(20, a(10, nil, a(15, nil, nil)), a(30, nil, nil)) représente l'arbre binaire suivant :



Un arbre binaire de racine R, avec G comme sous-arbre gauche et D comme sous-arbre droit, est dit « de recherche », s'il vérifie les conditions suivantes :
a. toutes les valeurs de son sous-arbre gauche G sont inférieures à la valeur de R.
b. toutes les valeurs de son sous-arbre droit D sont supérieures à la valeur de R.
c. les arbres G et D sont eux aussi des arbres de recherche.
Un arbre vide est toujours considéré comme étant un arbre de recherche.

L'arbre de l'exemple précédent est un arbre de recherche.

Le programme Prolog suivant, permet d'insérer une valeur dans un arbre de recherche binaire.
Le prédicat 'ins(X,A,B)' sera vrai si l'insertion de la valeur X dans l'arbre de recherche binaire A, donne comme résultat l'arbre de recherche binaire B.

```
ins( X , nil , a( X, nil, nil) ).          // l'insertion de X dans un arbre vide.

ins( X, a( Y, G, D ), a( Y, N, D ) ) :-    // l'insertion de X dans un arbre non vide,
        ins( X, G, N ), X < Y.                  // dans le cas où X < à la valeur de la racine.

ins( X, a( Y, G, D ), a( Y, G, N ) ) :-    // l'insertion de X dans un arbre non vide,
        ins( X, D, N ), X > Y.                  // dans le cas où X > à la valeur de la racine.
```

Question : ins(25, a(20, a(10, nil, a(15, nil, nil)), a(30, nil, nil)) , A)
/ quel est l'arbre A obtenu en insérant 25 dans l'arbre de l'exemple précédent ? */*
Réponses :
A = a(20, a(10, nil, a(15, nil, nil)), a(30, a(25, nil, nil), nil))

Tel que le prédicat 'ins' a été défini (les tests prédéfinis < et > ont été mis en fin de condition) permet de formuler des questions autrement, par exemple :

Question : ins(X, a(20, a(10, nil, a(15, nil, nil)), a(30, nil, nil)) , a(20, a(10, nil, a(15, nil, nil)), a(30, a(25, nil, nil), nil)))
/ quel est la valeur X à insérer dans l'arbre de l'exemple pour obtenir l'arbre a(20, a(10, nil, a(15, nil, nil)), a(30, a(25, nil, nil), nil)) ? */*
Réponses :
X = 25

Même les listes sont représentées en interne par des objets structurés :
L = nil / lst(entier, L)
une liste est soit nil, soit composée d'un premier élément de type 'entier' et d'une sous liste pour le reste des éléments.

Par exemple la liste [1,2,3] sera représentée par le terme structuré : lst(1,lst(2,lst(3,nil))).

Voici un exemple de programme Prolog manipulant ce type de structure, il décrit le prédicat 'concat(L1,L2,L3)' évalué à vrai si la liste L3 est le résultat de la concaténation des listes L1 et L2 :

```
/* la concaténation d'une liste vide avec une liste L, donne la même liste L */
concat( nil, L, L ).

/* la concaténation d'une liste [X|L1] avec une liste L2, donne la liste [X|L3], telle que L3 est le résultat de
   la concaténation de L1 avec L2. */
concat( lst(X,L1), L2, lst(X,L3) ) :-
        concat( L1, L2, L3 ).
```

Question : concat(lst(1, lst(2, nil)) , lst(3,nil), X).
/ Quel est le résultat de la concaténation de [1,2] avec [3] ? /*
X = lst(1, lst(2, lst(3,nil))) / résultat X =[1, 2, 3] */*

Question : concat(lst(1, lst(2, nil)), Y, lst(1, lst(2, lst(3,nil)))).
/ Quelle est la liste Y, telle que concaténée avec [1,2], donnera comme résultat [1,2,3] ? */*
Y = lst(3, nil) / c-a-d [3] */*

Question : concat(X, Y, lst(1, lst(2, lst(3,nil)))).
/ Quelles sont les listes X et Y dont la concaténation produit la liste [1,2,3] ? */*
X = nil, Y = lst(1, lst(2, lst(3,nil)))
X = lst(1,nil), Y = lst(2, lst(3,nil))
X = lst(1,lst(2,nil)), Y = lst(3,nil)
X = lst(1, lst(2, lst(3,nil))), Y = nil

On peut aussi utiliser les objets structurés pour définir et manipuler les entiers à l'aide de prédicats purement logique :

Ent = zéro / succ(Ent)
un entier est soit le symbole 'zéro', soit c'est le successeur d'un entier.

Par exemple, l'entier 3 sera représenté par le terme : succ(succ(succ(zéro)))

Voici un exemple de prédicat purement logique définissant la relation 'inférieure' entre entiers :

inf(zero, succ(_)).
/ zero est inférieur à tout entier qui est de la forme succ(...) */*

inf(succ(X), succ(Y)) :- inf(X, Y).
/ si X est inférieur à Y alors succ(X) est aussi inférieur à succ(Y) */*

On pourra alors poser une question de la forme :

Question : inf(X, succ(succ(succ(zero)))) / quels sont tous les X inférieurs à 3 */*
Réponses :
X = zero
X = succ(zero)
X = succ(succ(zero))

Ce type de question ne sont pas possible, en général, avec les prédicats prédéfinis comme (<, >, =, …) car ils ne peuvent être évalués que si tous leur argument sont connus.