# Programmation logique
# Exercices en Prolog
# (avec solutions)

Année Académique 2013-2014

Professeur : Tom Mens
Chef de Travaux : Olivier Delgrange
Service de Génie Logiciel
Institut d'Informatique, Faculté des Sciences
Université de Mons

3 décembre 2013

# 1 Résolution et unification

**Exercice 1.1** *Simulez le processus de résolution, unification et retour en arrière (*backtracking*) pour trouver une réponse à la requête* `?- seuleChargeeDeFamille(X)`*, étant donné les faits et clauses suivants :*

```
femme(isabel).
femme(anne).
parent(anne,pierre).
parent(anne,filip).
parent(jean,pierre).
parent(michel,filip).
decede(michel).
decede(isabel).

seuleChargeeDeFamille(F) :-
  femme(F),
  parent(F,E),
  parent(H,E),
  H\=F,
  decede(H).
```

**Remarque :** `H\=F` est la négation de `H=F`

**Solution 1.1** *Solution pas donnée.*

**Exercice 1.2** *Utilisez cette analyse/simulation pour établir un* ***arbre de preuve*** *qui démontre qu'il y existe une solution pour la requête* `seulChargeeDeFamille(X)`.

**Solution 1.2** *Solution pas donnée.*

**Exercice 1.3** *Donnez les résultats des requêtes suivantes en Prolog :*

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

**Solution 1.3** *Voici la solution :*

```
?- X=Y.
X = _G157
Y = _G157

?- X is Y
ERROR: Arguments are not sufficiently instantiated.

?- X=Y, Y=Z, Z=1.
X=1
Y=1
Z=1

?- X=1, Z=Y, X=Y.
X=1
Z=1
Y=1

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

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

?- 1+2 == 1+2.
Yes

?- X == Y.
No

?- X == X
X = _G157

?- 1 =:= 2-1
Yes

?- X =:= Y
ERROR: Arguments are not sufficiently instantiated.
```

**Exercice 1.4** *Écrivez un programme Prolog pour calculer les relations de famille frère, soeur, nièce, neveu, oncle, tante, cousin et cousine, étant donné l'arbre génálogique suivant :*

```
homme(adrien).
homme(hugo).      parent(adrien,hugo).
homme(bernard).   parent(hugo,bernard).
```

```
homme(alain).      parent(hugo,alain).
homme(guy).        parent(adrien,guy).
homme(pierre).     parent(guy,pierre).
femme(veronique). parent(guy,veronique).
```

*Le programme doit se comporter comme décrit ci-dessous :*

```
?- frere(X,Y).
X = hugo     Y = guy ;
X = bernard  Y = alain ;
X = alain    Y = bernard ;
X = guy      Y = hugo ;
X = pierre   Y = veronique ;
No

?- soeur(veronique,pierre).
Yes

?- oncle(guy,X).
X = bernard;
X = alain;
No

?- niece(X,hugo).
X = veronique
Yes

?- cousin(alain,X).
X = pierre;
X = veronique;
No
```

**Solution 1.4** *Les règles Prolog ci-dessous forment la solution :*

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.
frere(X,Y) :- homme(X), sibling(X,Y).
soeur(X,Y) :- femme(X), sibling(X,Y).
oncle(X,Y) :- parent(Z,Y), frere(X,Z).
tante(X,Y) :- parent(Z,Y), soeur(X,Z).
neveu(X,Y) :- sibling(Y,Z), parent(Z,X), homme(X).
niece(X,Y) :- sibling(Y,Z), parent(Z,X), femme(X).
cousin(X,Y) :- homme(X),
  parent(XP,X), sibling(XP,YP), parent(YP,Y).
cousine(X,Y) :- femme(X),
  parent(XP,X), sibling(XP,YP), parent(YP,Y).
```

# 2 La récursion

**Exercice 2.1** *Écrivez un programme Prolog pour calculer les nombres de Fibonacci*

1. *en utilisant la définition récursive suivante :*

$$f(0) = 1, f(1) = 1, \forall n > 1 : f(n) = f(n-1) + f(n-2)$$

2. *en optimisant le code par l'utilisation de la récursion terminale (*anglais : tail recursion*). Ceci nécessite l'utilisation d'un paramètre supplémentaire.*
3. *en proposant une solution bidirectionnelle. Ceci nécessite l'utilisation de la programmation logique par contraintes (*anglais : contraint-logic programming*), en important le module* `clpfd`*.*

*Le programme doit se comporter comme décrit ci-dessous :*

```
?- fibonacci(0,X).
X = 1.

?- fibonacci(12,X).
X = 233.

?- fibonacci(5,X).
X = 8.

? - fibonacci(X,13) %fonctionne uniquement avec la solution bidirectionnelle
X = 6.
```

**Solution 2.1** *Voici le code Prolog pour calculer les nombres de Fibonacci :*

1. *Version récursive*

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,F) :-
  N>1,
  N1 is N-1, fibonacci(N1,F1),
  N2 is N-2, fibonacci(N2,F2),
  F is F1+F2.
```

2. *Version utilisant récursion terminale*

```
fib(0,1).
fib(1,1).
fib(N,F) :-
  N > 1,
  fib-iter(N,1,1,F).

fib-iter(2,F1,F2,F) :-
  F is F1 + F2.
```

```
fib-iter(N,F1,F2,F) :-
  N > 2,
  N1 is N - 1,
  NF1 is F1 + F2,
  fib-iter(N1,F2,NF1,F).
```

3. *Solution bidirectionnelle*

```
:- use_module(library(clpfd)).
fib(0,1).
fib(1,1).
fib(N,F) :-
  N #> 1,
  N1 #= N - 1,
  N2 #= N - 2,
  F #= F1 + F2,
  fib(N1,F1),
  fib(N2,F2).
```

**Exercice 2.2** *Écrivez un programme Prolog pour calculer le plus grand commun diviseur (P.G.C.D.), en utilisant l'algorithme d'Euclide :*

- *pgcd(a,b) = a if a=b*
- *pgcd(a,b) = pgcd(a-b,b) if a>b*
- *pgcd(a,b) = pgcd(a,b-a) if b<a*

*Exemples :*

```
?- gcd(14,21,X).
X = 7

?- gcd(8,4,X).
X = 4
```

**Solution 2.2** *Voici les règles en Prolog pour calculer le pgcd :*

```
gcd(A,A,A). %gcd(A,B,GCD) :- A = B, GCD = A.
gcd(A,B,GCD) :- A > B, Temp is A - B, gcd(Temp,B,GCD).
gcd(A,B,GCD) :- A < B, Temp is B - A, gcd(A,Temp,GCD).
```

*ou alternativement :*

```
gcd(A,A,A).
gcd(A,B,Result) :-
  A>B, Temp is A-B, gcd(Temp,B,Result).
gcd(A,B,Result) :-
  A<B, Temp is B-A, gcd(A,Temp,Result).
```

*ou alternativement (syntaxe) :*

```
gcd(A,A,A).
gcd(A,B,Result) :-
  (A>B -> (Temp is A-B, gcd(Temp,B,Result));
   B>A -> (Temp is B-A, gcd(A,Temp,Result))).
```

*ou plus efficace*

```
gcd3(A,0,A).
gcd3(A,B,GCD) :- R is A mod B, gcd3(B,R,GCD),!.
```

**Exercice 2.3** *Écrivez un programme Prolog pour calculer la fonction d'Ackerman (voir figure 1).*

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

FIGURE 1 – Fonction d'Ackerman

*La valeur, ainsi que le temps de calcul, grandit très vite, même pour des nombres petits. Par exemple, A(4,2) est un entier constitué de 19729 chiffres décimaux.*

**Solution 2.3** *Voici les règles en Prolog pour la fonction d'Ackerman :*

```
ack(0,N,A) :-
  A is N + 1.
ack(M,0,A) :-
  M > 0,
  M1 is M - 1,
  ack(M1,1,A).
ack(M,N,A) :-
  M > 0,
  N > 0,
  M1 is M - 1,
  N1 is N - 1,
  ack(M,N1,A1),
  ack(M1,A1,A).
```

# 3 Les listes

**Exercice 3.1** *Donnez le résultat des requêtes suivantes en Prolog :*

```
?- [1,2,3] = [1|X].
?- [1,2,3] = [1,2|X].
?- [1 | [2,3]] = [1,2,X].
?- [1 | [2,3,4]] = [1,2,X].
?- [1 | [2,3,4]] = [1,2|X].
?- b(o,n,j,o,u,r) =.. L.
?- bon(Y) =.. [X,jour].
?- X(Y) =.. [bon,jour].
?- [1 | [2 | [3]]] = L.
?- [1 | [2 | [3]]] = [X | Y].
```

**Solution 3.1** *Voici la solution :*

```
?- [1,2,3] = [1|X].
X = [2,3]

?- [1,2,3] = [1,2|X].
X = [3]

?- [1 | [2,3]] = [1,2,X].
X = 3

?- [1 | [2,3,4]] = [1,2,X].
No

?- [1 | [2,3,4]] = [1,2|X].
X = [3,4]

?- b(o,n,j,o,u,r) =.. L.
L = [b,o,n,j,o,u,r]

?- bon(Y) =.. [X,jour].
X = bon
Y = jour

?- X(Y) =.. [bon,jour].
ERROR: Syntax error: Operator expected => X

?- [1 | [2 | [3]]] = L.
L = [1, 2, 3].

?- [1 | [2 | [3]]] = [X | Y].
X = 1,
```

```
Y = [2, 3].
```

**Exercice 3.2** *Écrivez un prédicat* `member2(E,L)` *en Prolog pour déterminer si un élément* `E` *est un membre d'une liste* `L`*. Le prédicat doit se comporter comme décrit ci-dessous :*

```
?- member2(c,[a,b,c,d,e]).
Yes

?- member2(f,[a,b,c,d,e]).
No

?- member2(X,[a,b,c]).
X = a;
X = b;
X = c;
No
```

**Solution 3.2** *Le code Prolog dessous :*

```
member2(X, [X|_]).
member2(X,[_|Y]) :-
  member2(X,Y).
```

**Exercice 3.3** *Écrivez un prédicat* `subset2(L1,L2)` *en Prolog pour déterminer si une liste* `L1` *est un sous-ensemble d'une autre liste* `L2`*. Le prédicat doit se comporter comme décrit ci-dessous :*

```
?- subset2([4,3],[2,3,5,4]).
Yes
```

**Solution 3.3** *Le code Prolog dessous :*

```
subset2([X|R],S) :- member(X,S), subset2(R,S).
subset2([],_).
```

**Exercice 3.4** *Écrivez un prédicat* `takeout(E,L1,L2)` *en Prolog pour retirer un éléments d'une liste* `L1`*. Le prédicat devrait être bidirectionnel, permettant également insérer des éléments dans un liste, comme montré ci-dessous :*

```
?- takeout(X,[1,2,3],L).
X = 1
L = [2, 3] ;
X = 2
L = [1, 3] ;
X = 3
L = [1, 2] ;
No
```

```
?- takeout(4,L,[1,2,3]).
L = [4, 1, 2, 3] ;
L = [1, 4, 2, 3] ;
L = [1, 2, 4, 3] ;
L = [1, 2, 3, 4] ;
No

?- takeout(3,[1,2,3],[1,2]).
Yes
```

**Solution 3.4** *Le code Prolog dessous :*

```
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :-
  takeout(X,R,S).
```

**Exercice 3.5** *Écrivez un prédicat* `getEltFromList(L,N,E)` *en Prolog pour obtenir le N-ième élément d'une liste. Il échoue si la liste ne contient pas N éléments.*

```
?- getEltFromList([a,b,c],0,X).
No

?- getEltFromList([a,b,c],2,X).
X = b

?- getEltFromList([a,b,c],4,X).
No
```

**Solution 3.5** *Le code Prolog dessous :*

```
getEltFromList([Head|Tail],0,_) :- fail.
getEltFromList([Head|Tail],1,Head).
getEltFromList([_|Tail],N,Z) :-
  M is N-1,
  getEltFromList(Tail,M,Z).
```

**Exercice 3.6** *Écrivez un programme Prolog pour calculer l'inverse d'une liste. Le programme doit se comporter comme décrit dessous :*

```
?- inverse([a,b,c],[c,b,a]).
Yes

?- inverse(X,[a,b,c]).
X = [c, b, a]
Yes

?- inverse([a,b,c],X).
X = [c, b, a]
Yes
```

*Tentez d'en trouver une version efficace en utilisant la récursion terminale.*

**Solution 3.6** *Première alternative (inefficace, récursive et utilisant append) :*

```
inverse([],[]).

inverse([Head | Tail], List) :-
  var(List),!,
  inverse(Tail, Result),
  append(Result, [Head], List).

inverse([Head | Tail], List) :-
  nonvar(List),
  append(Result, [Head], List),
  inverse(Tail, Result),!.
```

*Deuxième alternative : version efficace (itérative utilisant la récursion terminale) :*

```
inverse2(List,RList) :-
  inverse-iter(List,[],RList).

inverse-iter([],RL,RL) :- !.
  %the cut is used to stop the evaluation after having found one solution
  %in order to avoid an infinite loop
inverse-iter([Element|List],RevPrefix,RL) :-
  inverse-iter(List,[Element|RevPrefix],RL).
```

**Exercice 3.7** *Écrivez un programme Prolog pour déterminer le maximum et le minimum des valeurs dans une liste de nombres. Le programme doit se comporter comme décrit dessous :*

```
?- maxmin([3,1,5,2,7,3],Max,Min).
Max = 7
Min = 1
Yes

?- maxmin([2],Max,Min).
Max = 2
Min = 2
Yes
```

getEltFromList([a,b,c],2,X).

**Solution 3.7** *Le code Prolog dessous :*

```
maxmin([Elt],Elt,Elt).
```

```
maxmin([Elt|Rest],Max,Min) :-
  maxmin(Rest,TempMax,TempMin),
  (TempMax>Elt -> Max is TempMax;
   otherwise -> Max is Elt),
  (TempMin<Elt -> Min is TempMin;
   otherwise -> Min is Elt).
```

**Exercice 3.8** *Écrivez un programme Prolog pour transformer une liste de listes (chaque sous-liste peut contenir des autres listes) en un liste qui contient les élement de chacune des listes dans le même ordre.*

```
flatten([[1,2,3],[4,5,6]], Flatlist).
Flatlist = [1,2,3,4,5,6]
Yes

flatten([[1,[hallo,[[aloha]]],2,[],3],[4,[],5,6]], Flatlist).
Flatlist = [1, hallo, aloha, 2, 3, 4, 5, 6]
Yes
```

**Solution 3.8** *Le code Prolog dessous :*

```
flatten(L,error) :-
  var(L),
  write('flatten called with infinite list '), write(L),nl,!,fail.

flatten([],[]).

flatten(Item,[Item]) :-
  not(isList(Item)).

flatten([First | Rest], Result) :-
  flatten(First,FlatFirst),
  flatten(Rest,FlatRest),
  append(FlatFirst,FlatRest,Result).
```

# 4 Les foncteurs

**Exercice 4.1** *Donnez le résultat des requêtes suivantes en Prolog :*

```
?- op(X) is op(1).
?- op(X) = op(1).
?- op(op(Z),Y) = op(X,op(1)).
?- op(X,Y) = op(op(Y),op(X)).
```

**Solution 4.1** *Voici la solution :*

```
?- op(X) is op(1).
ERROR: Arithmetic: 'op/1' is not a function

?- op(X) = op(1).
X = 1

?- op(op(Z),Y) = op(X,op(1)).
Y = op(1)
Z = _G157
X = op(_G157)

?- op(X,Y) = op(op(Y),op(X)).
X = op(op(op(op(op(op(op(op(op(op(...)))))))))
Y = op(op(op(op(op(op(op(op(op(...))))))))))
```

***Remarque 4.1*** *La façon d'afficher ces imbrications récursives peut différer.*

**Exercice 4.2** *Écrivez un programme en Prolog pour traverser un arbre binaire en "préordre", "postordre" ou "inordre" et fournir la liste des éléments dans l'ordre demandé. L'arbre binaire correspondant à l'expression arithmetique (5+6)*(3-(2/2)) est*

```
tree('*',tree('+',leaf(5),leaf(6)),
         tree('-',leaf(3),tree('/',leaf(2),leaf(2))))
```

*Sa traversée en "préordre" est [*, +, 5, 6, -, 3, /, 2, 2] Sa traversée en "inordre" est [5, +, 6, *, 3, -, 2, /, 2] Sa traversée en"postordre" est [5, 6, +, 3, 2, 2, /, -, *]*

**Solution 4.2** *Voici les règles générales pour traverser un arbre binaire en préordre :*

```
traverse(tree(X,_,_),X).
traverse(leaf(Y),Y).
traverse(tree(_,A,_),Y) :- traverse(A,Y).
traverse(tree(_,_,B),Y) :- traverse(B,Y).
```

*Pour le changer en inorde ou postordre, il faut simplement changer l'ordre des 5 régles. Si on veut obtenir des listes comme résultat, le code est ceci :*

```
preorder(leaf(X),[X]).
preorder(tree(X,Y,Z),[X|List]) :-
   preorder(Y,List1),
   preorder(Z,List2),
   append(List1,List2,List).

inorder(leaf(X),[X]).
inorder(tree(X,Y,Z),List) :-
   inorder(Y,List1),
   inorder(Z,List2),
   append(List1,[X|List2],List).

postorder(leaf(X),[X]).
postorder(tree(X,Y,Z),List) :-
   postorder(Y,List1),
   postorder(Z,List2),
   append(List1,List2,List3),
   append(List3,[X],List).
```

# 5 Le *cut*

**Exercice 5.1** *Écrivez un programme Prolog pour calculer l'union de deux listes. L'ordre n'a pas d'importance, un élément présent dans les deux listes ne doit se trouver qu'une seule fois dans le résultat.*

*Le programme doit se comporter comme décrit dessous :*

```
?- union3([1,2,3],[4,5,6],L).
L = [1, 2, 3, 4, 5, 6]
?- union3([1,2,3,4,7],[2,4,5,6],L).
L = [1, 3, 7, 2, 4, 5, 6].
```

**Solution 5.1** *Le code Prolog dessous :*

```
union3([],Z,Z).
union3([X|Y],Z,W) :-
  member(X,Z), !,
  union3(Y,Z,W).
union3([X|Y],Z,[X|W]) :-
  union3(Y,Z,W).
```

**Exercice 5.2** *Écrivez un programme Prolog pour calculer l'intersection de deux listes. Le programme doit se comporter comme décrit dessous :*

```
?- intersection3([1,2,3,4],[1,a,b,4],L).
L = [1, 4]
Yes
```

**Solution 5.2** *Le code Prolog dessous :*

```
intersection3([],Z,[]).
intersection3([X|Y],Z,[X|W]) :-
  member(X,Z), !,
  intersection3(Y,Z,W).
intersection3([X|Y],Z,W) :-
  intersection3(Y,Z,W).
```

# 6 Les prédicats d'ordre supérieur

**Exercice 6.1** *Implémentez le prédicat d'ordre supérieur* `map(P,L1,L2)` *en Prolog. Ce prédicat prend en entrée un prédicat* `P`*, l'applique sur tous les éléments de la liste* `L1`*, et renvoie une nouvelle liste* `L2` *avec les résultats. Par exemple :*

```
square(X,Y) :- Y is X*X.
?- maplist(square, [1,2,3,4], Result).
[1, 4, 9, 16]

?- maplist(square, [1,2,3,4], [1, 4, 9, 16]).
true

maplist(square, [1,2,3,4], [1, 2, 3, 4]).
false
```

*Consigne : utilisez* `call` *pour implémenter votre solution.*

**Solution 6.1** *Le code Prolog dessous :*

```
maplist(P, [], []).
maplist(P, [X1|X1s], [X2|X2s]) :-
   call(P, X1, X2),
   maplist(P, X1s, X2s).
```

**Exercice 6.2** *Implémentez le prédicat d'ordre supérieur* `filter(F,L1,L2)` *en Prolog. Ce prédicat prend en entrée un prédicat (le filtre)* `F`*, l'applique sur chaque élément de la liste* `L1`*, et renvoie une nouvelle liste* `L2` *contenant tous les éléments de la liste* `L1` *pour lesquels le filtre retourne vrai.*

*Par exemple :*

```
positive(X) :- X>0.

?- filter(positive,[1,2,3,4],X).
X = [1, 2, 3, 4]

?- filter(positive,[1,-2,3,-4],X).
X = [1, 3]
```

```
?- filter(positive,[-1,-2,-3,-4],X).
X = []

?- filter(positive,[],X).
X = []
```

*Consigne : utilisez `call` pour implémenter votre solution.*

**Solution 6.2** *Le code Prolog dessous :*

```
filter(P, [], []).
filter(P, [H|T], [H|T2]) :-
  call(P,H),filter(P,T,T2).
filter(P,[H|T],T2) :-
  not(call(P,H)),filter(P,T,T2).
```

*ou alternativement, une version qui ne nécessite pas l'utilisation de `not` mais qui utilise la notion de* cut *( ! ) au lieu de cela :*

```
filter(P, [], []).
filter(P, [H|T], [H|T2]) :-
  call(P,H),filter(P,T,T2),!.
filter(P,[H|T],T2) :-
  filter(P,T,T2),!.
```

**Exercice 6.3** *Utilisez le prédicat `forall` pour vérifier si deux listes sont disjointes.*

```
?- disjoint([a,b,c],[d,g,f,h]).
Yes

?- disjoint([a,b,c],[f,a]).
No
```

**Solution 6.3** *Le code Prolog dessous :*

```
disjoint(Set1,Set2) :-
  forall(member(El,Set1), not(member(El,Set2))).
```

**Exercice 6.4** *Utilisez le prédicat `forall/2` pour vérifier si une liste est un sous-ensemble d'une autre liste.*

```
?- subset3([a,b,c],[c,d,a,b,f]).
Yes

?- subset3([a,b,q,c],[d,a,c,b,f])
No
```

**Solution 6.4** *Le code Prolog dessous :*

```
subset3(Set1,Set2) :-
  forall(member(El,Set1), member(El,Set2)).
```

**Exercice 6.5** *Utilisez le prédicat `findall/3` pour calculer l'intersection de deux listes.*

**Solution 6.5** *Le code Prolog dessous :*

```
intersection3(Set1,Set2,Intersection) :-
  findall(El,and(member(El,Set1), member(El,Set2)),Intersection).
```

# 7 assert et retract

**Exercice 7.1** *Implémentez de manière efficace un prédicat Prolog `fibo(N,F)` pour calculer les nombres de Fibonacci, sans utiliser la récursion terminale. Basez-vous sur l'idée de la "mémoisation" : chaque fois qu'un nombre de Fibonacci a été calculé, on garde sa valeur en mémoire, afin de récupérer cette valeur quand on en a besoin, évitant ainsi un recalcul, rendant le calcul linéaire en temps au lieu d'exponentiel. Utilisez le méta-prédicat `assert` pour réaliser cette solution*

**Solution 7.1** *Voici la solution :*

```
:- dynamic(fibofact/2).

fibofact(0,1).
fibofact(1,1).

fibo(N,F) :-
  fibofact(N,F),!.
fibo(N,F) :-
  N>1,
  N1 is N-2, fibo(N1,F1),
  N2 is N-1, fibo(N2,F2),
  F is F1+F2,
  assert(fibofact(N,F)).
```