# Projet Prolog 4IF

Hexanôme 4104 - Equipe Dragibus

26 octobre 2013

# Table des matières

**1 Généalogie** **2**

**2 Listes** **5**

**3 Arithmetique** **8**

**4 Ensembles** **8**

# 1 Généalogie

On créera une base de connaissances en spécifiant les relations "parent" directes. parent(X, Y) étant vrai si X est le parent de Y.

```
parent(john, martha).
parent(john, marc).
parent(rodrigo, john).
parent(jean, leo).
parent(louis, jean).
parent(louis, george).
parent(rodrigo, louis).
parent(ivonne, rodrigo).
```

Définition du lien grand-parent par une propagation de la relation parent sur 2 niveaux. grandParent(X, Y) est vrai si X est le grand parent de Y.

```
grandParent(X, Y) :-
  parent(X, Z),
  parent(Z, Y).

% Test de la relation grand-parent
testGrandParent :-
    grandParent(rodrigo, martha),
    grandParent(rodrigo, marc),
    grandParent(rodrigo, jean),
    grandParent(ivonne, john),
    grandParent(rodrigo, marc),
    \+ grandParent(jean, george),
    \+ grandParent(ivonne, martha),
    \+ grandParent(john, martha),
    \+ grandParent(martha, martha),
    grandParent(louis, leo),
    !.
```

Définition du lien ancêtre par une propagation récursive sur plusieurs niveaux, jusqu'à satisfaction. ancetre(X, Y) est vrai si X est un ancetre de Y.

```
ancetre(X, Y) :-
  parent(X, Y).
ancetre(X, Y) :-
  parent(X, Z),
  ancetre(Z, Y).

% Test de la relation ancetre
testAncetre :-
    ancetre(ivonne, martha),
    ancetre(rodrigo, marc),
    ancetre(john, martha),
    \+ ancetre(martha, martha).
```

Définition de la liste des ancêtres à partir d'un élément de la famille. On utilise setof car cette partie n'est pas sur les listes, et son utilisation est pratique pour éliminer les doublons dans les candidats à la relation ancêtre. On définie de la meme façon pour récupérer la liste des descendants.
ancetres(X, L) est vrai si L est l'ensemble des ancetres de X.
descendants(X, L) est vrai si L est l'ensemble des descendants de X.
Dans ces deux prédicats, X ne peut pas être déterminée à partir de L, et donc ne peut pas être une variable.

```
ancetres(X, L) :-
  setof(A, ancetre(A, X), L).

descendants(X, L) :-
  setof(A, ancetre(X, A), L).

ancetres(X, _) :- var(X), fail.
descendants(X, _) :- var(X), fail.
```

Définition de la relation frère-soeur complète en partant du constat que celle-ci correspond au partage des parents. Cette relation n'est pas réflexive, une personne ne pouvant pas être son propre frère-soeur. frereSoeur(X, Y) est vrai si X et Y ont le meme parent (donc sont frère-soeur).

```
frereSoeur(X, Y) :-
  parent(Z, X),
  parent(Z, Y),
  X \= Y.

% Test de la relation frere-soeur
testFrereSoeur :-
    frereSoeur(martha, marc),
    frereSoeur(marc, martha),
    \+ frereSoeur(marc, marc),
    \+ frereSoeur(rodrigo, marc),
    \+ frereSoeur(rodrigo, jean).
```

Définition de la relation oncle-tante par une relation frère-soeur pour un parent. oncleTante(X, Y) est vrai si X est l'oncle (ou la tante) de Y.

```
oncleTante(X, Y) :-
  frereSoeur(X, Z),
  parent(Z, Y).

% Test de la relation oncle-tante
testOncleTante :-
    oncleTante(louis, marc),
    oncleTante(louis, martha),
    \+ oncleTante(rodrigo, martha),
    \+ oncleTante(marc, martha).
```

Définition de la relation cousin par une relation frère-sœur entre deux parents. cousin(X, Y) est vrai si X et Y sont cousin (càd que leur parent sont freres).

```
cousin(X, Y) :-
  parent(A, X),
  parent(B, Y),
  frereSoeur(A, B).

% Test de la relation cousin
testCousin :-
    cousin(jean, martha),
    cousin(marc, jean),
    \+ cousin(marc, martha),
    \+ cousin(rodrigo, martha).
```

## 2 Listes

element(X, L, LprivX) est vrai si L/X est une liste composé des éléments de L privé une fois de X. L'ordre doit etre préservé. Si X n'est pas dans L, alors le prédicat est faux. Les trois paramètres peuvent etre des variables.

```
element(X, [X|Xs], Xs).
element(X, [T|As], [T|Bs]) :-
  element(X, As, Bs),
  X \== T.

testElement :-
    \+ element(_, [], []),
    element(b, [a, b, c], [a, c]),
    element(a, [a, b, c], [b, c]),
    element(c, [a, b, c], [a, b]),
    \+ element(d, [a, b, c], _),
    \+ element(a, [a, b], [a, b]),
    \+ element(a, [c, b], [c, b]),
    \+ element(b, [a, b, c], [a, b]),
    % Il faut rajouter une coupure pour prouver les predicats
    % suivants pour limiter l'espace des solutions :
    % element(X, [X | L], L),
    % element(X, [A, X | L], [A | L]),
    % ...
    true.
```

extract(L1, L2) est vrai si L2 est un sous ensemble de L1. Les deux paramètres peuvent etre des variables.

```
extract(_, []).
extract(L, [X|Xs]) :-
  element(X, L, L2),
  extract(L2, Xs).

testExtract :-
    extract([], []),
    extract([a], []),
    extract([a], [a]),
    extract([a, b], []),
    extract([a, b], [a]),
    extract([a, b], [b]),
    extract([a, b], [b, a]),
    extract([a, b, c], []),
    extract([a, b, c], [b]),
    extract([a, b, c], [c]),
    extract([a, b, c], [a, b]),
    extract([a, b, c], [a, c]),
    extract([a, b, c], [b, a]),
    extract([a, b, c], [a, b, c]),
    extract([a, b, c], [a, c, b]),
    extract([a, b, c], [b, a, c]),
    \+ extract([], [_]),
    \+ extract([a], [b]),
    \+ extract([a, b, c], [a, b, d]),
    % Toutes les permutations de L1 sont des sous ensembles de L2 :
    permutation(L1, L2),
    extract(L1, L2),
    write(L1), write(' '), write(L2), write('\n').
```

concat(L1, L2, L1nL2) est vrai si L1nL2 est la concaténation de L1 et L2.

```
concat([], L2, L2).
concat([X|Xs], L2, [X|L]) :-
  concat(Xs, L2, L).

testConcat :-
    % On utilise append/3 de la lib standard pour tester
    append(L1, L2, L1nL2),
    concat(L1, L2, L1nL2),
    write(L1), write(' + '), write(L2), write(' = '), write(L1nL2), write('\n').
```

inv(L1, L2) est vrai si L1 est l'inverse de L2. On remarque que si L1 n'est pas une variable et que L2 en est une, le prédicat ne se termine pas.

```
inv([], []).
inv([X|Xs], L) :-
  inv(Xs, L2),
  concat(L2, [X], L).

testInv :-
    % On utilise reverse/2 de la lib standard pour tester
    reverse(L1, L2),
    inv(L1, L2),
    write(L1), write(' : '), write(L2), write('\n').
```

subsAll(E, X, L1, L2) est vrai si L2 est la liste L1 avec les éléments E remplacés par des X. Dans le cas ou E est une variable, il prend la première valeur de L1.

```
subsAll(_, _, [], []).
subsAll(E, X, [T|Q], L) :-
  T \= E,
  subsAll(E, X, Q, Ls),
  concat([T], Ls, L).
subsAll(E, X, [E|Q], L) :-
  subsAll(E, X, Q, Ls),
  concat([X], Ls, L).

testSubsAll :-
    % On utilise select/4 de la lib standard.
    select(X, L1, Y, L2),
    subsAll(X, Y, L1, L2),
    write(X), write(' => '), write(Y), write(' : '),
    write(L1), write(' => '), write(L2), write('\n').
```

## 3 Arithmetique

element(Idx, X, L) est vrai si X est à la position Idx dans L (en commençant à compter à partir de 1). On doit passer par une fonction element/4 pour intégrer un accumulateur en paramètre.

```
element(Idx, X, L) :-
  element(Idx, 0, X, L).
element(Idx, _, _, _) :-
      Idx < 1, !, fail.
element(Idx, Count, H, [H|_]) :-
  Idx is Count + 1.
element(Idx, Count, Item, [_|T]) :-
  Count1 is Count + 1,
  element(Idx, Count1, Item, T).

testElement :-
    \+ element(0, _, _),
    element(1, X, [X]),
    element(1, X, [X, _, _, _, _]),
    \+ (X \= Y, element(2, X, [_, Y, _, _, _])),
    element(1, a, [a, b, a]),
    \+ element(2, a, [a, b, a]),
    element(3, a, [a, b, a]),
    % Identique a nth1/3. On l'utilise alors pour prouver le code.
    nth1(Idx, L, E),
    element(Idx, E, L).
```

# 4 Ensembles

list2ens(L, E) est vrai quand E correspond a l'ensemble des elements de L, sans doublon. L'ordre de la liste initiale est conserve. On doit alors définir un predicat list2ens/3 pour utiliser un accumulateur en paramètre. Egalement, on définit simplement un prédicat element/2 qui est vrai lorsqu'un élément X est dans une liste L.

```
element(_, []) :- fail.
element(X, [X|_]).
element(X, [_|Q]) :- element(X, Q).

list2ens([], V, V).
list2ens([T|Q], V, E) :-
  element(T, V),
  list2ens(Q, V, E).
list2ens([T|Q], V, E) :-
  \+element(T, V),
  concat([T], V, V1),
  list2ens(Q, V1, E).

list2ens(L, E) :-
    list2ens(L, [], E1),
    inv(E1, E),
    !.

% Test de la conversion d'une liste en ensemble
testList2Ens :-
    list2ens([], []),
    list2ens([a, a], [a]),
    list2ens([a, b, a], [a, b]),
    \+ list2ens([a, b], [a]),
    \+ list2ens([a, b, c], [a, b]).
```

ensemble(L) est vrai quand L est un ensemble, et donc ne contient pas de doublons. On réutilise le prédicat précédemment fait list2ens.

```
ensemble(L) :-
    list2ens(L, L).

% Test de la verification d'un ensemble
testEnsemble :-
    ensemble([]),
    ensemble([a]),
    ensemble([a, b]),
    ensemble([a, b, c]),
    \+ ensemble([a, a]),
    \+ ensemble([a, b, a]),
    \+ ensemble([a, b, b]),
    \+ ensemble([c, c, b]).
```