

Prolog TP n° 2

Université Paris I. Maîtrise de Logique. Année 2004-2005.
Nicolas Stroppa - stroppa@enst.fr

1 Types

Dans le premier TP, nous avons vu la structure de liste (un cas particulier de termes), et comment cette structure permettait de représenter des données structurées. Les constantes présentes dans ces termes peuvent être de certains types. Les prédicats associés sont les suivants :

- `atom(A)` : A est un atome.
- `integer(I)` : I est un entier.
- `real(R)` : R est un flottant.
- `number(N)` : N est un entier ou un flottant.
- `atomic(A)` : A est un atome ou un nombre.

Certains prédicats peuvent fournir des informations sur la nature des objets manipulés :

- `var(V)` : V est une variable.
- `nonvar(NV)` : NV n'est pas une variable.
- `functor(T,F,A)` : T est un terme dont le foncteur est F, et l'arité A.
- `T =..L` : T est un terme dont la décomposition sous forme de liste est L (ex.: `f(a,b) =.. [f, a, b]`).
- `clause(H,T)` : H :- T est une règle du programme.

1.1 Utilisation de prédicats sur les types

Écrire les prédicats suivants :

1. `est_liste_atomes(X)` : X est une liste dont les éléments sont des atomes.
2. `atomise(X, Y)` : Y est la liste des atomes de X.
3. `somme(X, N)` : N est la somme des éléments entiers de X (0 s'il n'y a pas d'entiers dans X).

2 Prolog : un langage « logique » ?

Même si les fondations de Prolog sont construites à partir de la logique, l'identification *Prolog = Logique* présente un certain nombre de limites.

2.1 L'équivalence logique

1. Commentez le programme suivant d'un point de vue logique :

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

2. Que se passe-t-il quand on demande le but `p(X)` ou le but `q(X)` ?
3. Qu'en concluez vous ?

2.2 L'ordre des littéraux

1. Établir des faits sur les liens de parenté entre des personnes (prédicat `parent(X, Y)`).
2. Définir le prédicat `ancêtre` à partir du prédicat `parent`.
3. Concluez sur l'importance de l'ordre des littéraux dans les clauses Prolog.

2.3 La négation

La négation logique n'est pas présente en tant que telle dans Prolog. Prolog fait l'hypothèse du « monde clôt », i.e. il considère faux ce qu'il n'arrive pas à démontrer. Pour simuler le faux (prédicat Prolog `not`), il faut se servir du prédicat `fail` qui fait toujours échec et de la coupure (le « cut », notée !), que l'on va étudier dans la suite.

3 Nécessité de la coupure

1. Écrire `ajouter` qui ajoute un élément dans une liste, en s'assurant que cet élément n'est pas dupliqué.
2. Xavier apprécie tous les restaurants, sauf les restaurants mexicains. Écrire le prédicat `apprécie` en utilisant `fail`, puis donner une deuxième solution en utilisant `not`.
3. Considérer le programme suivant :

```
chic(lasserre).  
chic(maxims).  
cher(lasserre).
```

```
raisonnable(Restaurant) :- not (cher(Restaurant)).
```

4. Rechercher maintenant les restaurants chics et raisonnables, puis les restaurants raisonnables et chics.
5. Concluez.

4 Éléments de corrigé

```
%%% - Section 1.1 - %%%
est_liste_atomes([]).
est_liste_atomes([X|T]) :-
    atom(X),
    est_liste_atomes(T).

%% sans cut %%
atomise([], []).
atomise([X|L1], [X|L2]) :-
    atom(X),
    atomise(L1, L2).

atomise([X|L1], L2) :-
    not(atom(X)),
    atomise(L1, L2).

%% avec cut %%
atomise2([], []).
atomise2([X|L1], [X|L2]) :-
    atom(X),
    !,
    atomise2(L1, L2).

atomise2([X|L1], L2) :-
    atomise2(L1, L2).

%% sans cut %%
somme([], 0).
somme([H|T], N) :-
    integer(H),
    somme(T, N1),
    N is N1 + H.

somme([H|T], N1) :-
    not(integer(H)),
    somme(T, N1).

%% avec cut %%
somme2([], 0).
somme2([H|T], N) :-
    integer(H),
```

```

!,
somme2(T, N1),
N is N1 + H.

somme2([H|T], N1) :-
    somme2(T, N1).

%%% - Section 2.2 - %%%
parent(elizabeth, charles).
parent(charles, william).
parent(charles, henry).
parent(elizabeth, anne).
parent(anne, peter).
parent(anne, zara).
parent(elizabeth, andrew).
parent(andrew, beatrice)
parent(andrew, eugenie).
parent(elizabeth, edward).
parent(edward, louise).

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

ancetre(X, Y) :-
    parent(X, Z).
    ancetre(Z, Y). %% notez l'ordre !

%%% - Section 2.3 - %%%
non(X) :-
    X,
    !,
    fail.
non(X).

%%% - Section 3 - %%%
ajouter(X, L, X) :-
    extraire(X, L, _),
    !.

ajouter(X, L, [X|L]).

```

```
apprecie1(xavier, Y) :-  
    mexicain(Y),  
    !,  
    fail.  
  
apprecie1(xavier, _).  
  
apprecie2(xavier, Y) :-  
    not (mexicain(Y)).
```