

# Prolog TP n° 3

Université Paris I. Maîtrise de Logique. Année 2004-2005.  
Nicolas Stroppa - [stroppa@enst.fr](mailto:stroppa@enst.fr)

## 1 Tris

Dans cette section, il s'agit de manipuler des listes d'entiers dans le but de les trier. Nous allons voir deux sortes de tris, le *tri par insertion* et le *tri rapide*.

### 1.1 Tri par insertion

Définir les prédicats suivants :

1. `max(X, Y, Z)` : Z est le maximum de X et Y.
2. `is_ord(L)` : L est une liste d'entiers ordonnée.
3. `insere(X, Y, Z)` : insère Z dans X (supposée ordonnée) de façon à ce que le résultat Y soit également une liste ordonnée.
4. `tri_insertion(X, Y)` : Y est la liste des éléments de X triée par insertion.

### 1.2 Tri rapide

Le principe du tri rapide (quicksort en anglais) est le suivant. Tout d'abord, on regarde la tête de la liste et on divise le reste de la liste en deux listes qui contiennent respectivement les éléments plus petits et plus grands que la tête. Ensuite, on applique le procédé de façon récursive sur les deux listes construites. On obtient à la fin un arbre binaire ordonné, i.e. tel que les éléments du sous-arbre gauche sont inférieurs à la racine, elle-même inférieure aux éléments du sous-arbre droit (cette propriété est vérifiée pour tous les nœuds de l'arbre). Enfin, pour obtenir la liste triée, il suffit d'« aplatir » l'arbre.

Définir les prédicats suivants :

1. `aplatir(Arbre, L)`.
2. `separe(L, X, L1, L2)` : X est un élément quelconque, L est une liste quelconque, L1 est la liste composée des éléments de L plus petits que X, L2 est la liste composée des éléments de L plus grands que X.

3. `arbre_ordonne(L, Arbre)` : `Arbre` est l'arbre ordonné contenant les éléments de la liste `L`.
4. `tri_rapide(X, Y)` : `Y` est la liste des éléments de `X` triée par tri rapide.

## 2 Fonction de Fibonacci

La fonction de Fibonacci est définie par :

$$fib(0) = fib(1) = 1$$

$$fib(n + 2) = fib(n) + fib(n + 1)$$

1. Définissez un prédicat `fib` permettant de calculer la fonction de Fibonacci (une version avec accumulateur, une version sans, et une version avec `assert` (note: le pendant de `assert` est `retract`)).
2. Essayez le but `fib(20)` dans les différents cas.
3. Commentez.

## 3 Quelques notions sur la programmation en Prolog

### 3.1 Prédicats d'ordre 2

Le langage Prolog propose un certain nombre de prédicats du second ordre pour travailler sur l'ensemble des solutions d'un but.

`findall(X, But, L)` va « mettre » dans `L` tous les solutions de `But` où `X` est la variable recherchée dans `But`. Par exemple, on a :

```
ami(gerard, bertrand).
ami(gerard, sylvie).
ami(robert, bertrand).
```

```
? - findall(X, ami(X, bertrand), L).
L = [gerard, robert] ;
```

1. Essayez les buts `findall(X, ami(X, bertrand), L)`, `findall(X, ami(X, Y), L)`, `bagof(X, ami(X, bertrand), L)`, `bagof(X, ami(X, Y), L)`. `bagof(X, Y^ami(X, Y), L)`.
2. Définissez le prédicat `sous-listes(L, S)` qui stocke dans `S` toutes les sous-listes de `L` (une version avec `bagof`, et une avec `setof`).
3. Servez-vous de `setof` pour définir le prédicat `ensemble_liste` qui est l'ensemble (au sens ensembliste) des éléments d'une liste.
4. Même chose pour le prédicat `union_listes`.

5. Même chose pour le prédicat `intersection_listes`.

**Note:** `setof` est quasiment identique à `bagof`, mais il exclut les doublons et trie la liste.

### 3.2 Enrichir ses programmes

Prolog permet de définir de nouveaux opérateurs, de façon à simplifier l'écriture des termes.

Un opérateur se définit par `:- op(Precedence, Associativite, NomOperateur)`. où `Precedence` est une priorité de 0 à 1200, `Associativite` une constante de la forme `fy` (opérateur unaire) `xfy` (opérateur binaire), etc.

1. Définissez les opérateurs usuels de la logique propositionnelle.

### 3.3 Les cafards

Prolog fournit également des outils pour aider le programmeur dans sa tâche. Un programme n'étant jamais exempt de cafards, il faut savoir se servir des mécanismes permettant de les déceler. Le prédicat `trace` permet de suivre les étapes de résolution du prochain but demandé. Pour activer la trace, il suffit d'invoquer le but `trace, mon_but..`

1. Étudiez le comportement de quelques uns des prédicats que vous avez écrit jusqu'à maintenant à l'aide de la trace (tapez `h` quand la trace est lancée pour voir les options disponibles).

## 4 Éléments de corrigé

```
%%% - Section 1.1 - %%%
max(X, Y, Z) :- X < Y, !, Z = Y.
max(X, _, X).

is_ord([]).
is_ord([_]).
is_ord([X,Y|T]) :-
    X =< Y,
    is_ord([Y|T]).

insere([], [X], X).

insere([H|T], [X,H|T], X) :-
    X < H,
    !.
```

```

insere([H|T], [H|Y], X) :-
    insere(T, Y, X).

tri_insertion(X, Y) :-
    tri_insertion_acc(X, Y, []).

tri_insertion_acc([], L, L).
tri_insertion_acc([H|T], L, Acc) :-
    insere(Acc, Acc2, H),
    tri_insertion_acc(T, L, Acc2).

%%% - Section 1.2 - %%%
aplatir([], []).
aplatir([X], [X]).
aplatir([X, FilsGauche, FilsDroit], L) :-
    aplatir(FilsGauche, L1),
    aplatir(FilsDroit, L2),
    append(L1, [X|L2], L).

separe([], _, [], []).
separe([H|T], X, [H|L1], L2) :-
    H < X,
    !,
    separe(T, X, L1, L2).

separe([H|T], X, L1, [H|L2]) :-
    separe(T, X, L1, L2).

arbre_ordonne([], []).
arbre_ordonne([H|T], [H, Arbre1, Arbre2]) :-
    separe(T, H, L1, L2),
    arbre_ordonne(L1, Arbre1),
    arbre_ordonne(L2, Arbre2).

tri_rapide(L, Res) :-
    arbre_ordonne(L, Arbre),
    aplatir(Arbre, Res).

%%% - Section 2 - %%%
fib1(0, 1).

```

```
fib1(1, 1).

fib1(N, F) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib1(N1, F1),
    fib1(N2, F2),
    F is F1 + F2.

fib2(N, F) :-
    fib2_acc(N, F, _).

fib2_acc(0, 1, 0).
fib2_acc(1, 1, 1).

fib2_acc(N, F, F1) :-
    N > 1,
    N1 is N - 1,
    fib2_acc(N1, F1, F2),
    F is F1 + F2.

:- dynamic(fib3/2).
fib3(0, 1).
fib3(1, 1).

fib3(N, F) :-
    N > 1,
    N2 is N - 2,
    fib3(N2, F2),
    N1 is N - 1,
    fib3(N1, F1),
    F is F1 + F2,
    asserta(fib3(N, F) :- !).
```