

TD de *Prolog et programmation par contraintes* n° 2  
(Correction)

## Récursion, I/O de base et listes.

Pour lancer l'application (sur nivose): yap.

Pour charger le programme toto.pl: consult(toto). (ou, après modifications, reconsult(toto).)

Pour quitter l'application: halt.

**Exercice 1** On considère la fonction  $fibonacci : N \rightarrow N$  définie par:

$$fibonacci(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & \text{si } n \geq 2 \end{cases}$$

Définir un prédicat Prolog `fibonacci(Arg, Res)` pour le calcul de *fibonacci*.

Vérifier que la requête `fibonacci(n, R)` a un temps de calcul qui augmente très rapidement pour  $n > 30$  (pour stopper un calcul, `control c`, puis `a`).

Définir une deuxième version optimisée de *fibonacci*, en utilisant un prédicat auxiliaire à 3 places, tel que `aux(n, p, q)` réussit si `p` est *fibonacci*( $n$ ) et `q` est *fibonacci*( $n-1$ ).

Expliquer pourquoi cette version est plus efficace.

**Correction :**

```
fibonacci(0,1). fibonacci(1,1). fibonacci(N,R):- N > 1,P is N-1,PP is P-1,fibonacci(P,R1),fibonacci(PP,R2),R is R1+R2.
```

```
f1(N,R):-g(N,R,_).
```

```
g(0,1,0). g(1,1,1).
```

```
g(N,R,S):- N > 1,P is N-1, g(P,S,T), R is S+T.
```

**Exercice 2** On peut représenter les entiers naturels en *notation unaire* par les termes suivants:

0, `s(0)`, `s(s(0))`, `s(s(s(0)))`, etc.

– Écrire un prédicat `transform(X, Y)` qui étant donné un entier en notation unaire `X` donne dans `Y` sa valeur. Par exemple

```
?- transform(s(s(s(0))), Y)
```

donne

```
Y=3.
```

Est-ce qu'on peut utiliser ce prédicat pour une requête de la forme `?- transform(X, 3)` (qui devrait donner `X = s(s(s(0)))`)? Sinon définissez un prédicat pour cela.

– Écrivez un prédicat `somme(X, Y, Z)` qui prend deux entiers unaires et qui calcule leur somme. Par exemple

```
?- somme(s(s(0)), s(0), Z)
```

donne

```
Z = s(s(s(0)))
```

Est-ce que ça marche aussi pour `?- somme(X, Y, s(s(s(0))))`? (ne pas utiliser le prédicat `transform`)

- Même chose pour le produit de deux entiers en utilisant `somme/3`.
- Même chose pour le calcul de  $n^m$  en utilisant `produit/3`.

**Correction :**

```
transform(0,0).
transform(s(X),Y) :- transform(X,Z), Y is Z + 1.
```

```
somme(0,X,X).
somme(s(Y),X,s(Z)) :- somme(Y,X,Z).
```

```
produit(s(0),X,X).
produit(s(X),Y,Z) :- produit(X,Y,Z1), somme(Y,Z1,Z).
```

```
puissance(X,0,s(0)).
puissance(X,s(Y),R):-puissance(X,Y,T), produit(X,T,R).
```

**Exercice 3** Définir un prédicat `afficher/1` tel que, pour  $n$  entier naturel, `afficher(n)` affiche les entiers de 0 à  $n$  dans l'ordre (pour l'affichage, utiliser le prédicat prédéfini `write/1`).

**Correction :**

```
afficher(0) :- write(0).
afficher(N) :- N>0, M is N-1, afficher(M),nl, write(N).
```

**Exercice 4** Définir un prédicat `boucle/0` qui affiche à l'écran "choisir un entier", lit un entier, si l'entier donné est 0 termine, sinon repose la question est ainsi de suite. (Pour lire un entier, utiliser le prédicat prédéfini `read/1`).

**Correction :**

```
boucle:- write('choisir un entier'),
        nl,
        read(X),
        test(X).
```

```
test(0).
```

```
test(_):-boucle.
```

**Exercice 5** Si  $x_1, \dots, x_n$  sont des termes,  $[x_1, \dots, x_n]$  représente la liste de ces termes, et si  $x$  est un terme et  $l$  une liste,  $[x|l]$  représente la liste qui a comme  $x$  comme premier élément ("head") et  $l$  comme reste ("tail") (par exemple  $[1|[2|[ ]]] = [1,2], [ ]$  étant la liste vide)

Donner le résultat de chacune des requêtes suivantes (travailler sur papier, puis lancer la requête pour vérifier).

1. ?-  $[a, [a]] = [H|T]$  .
2. ?-  $[[a, b], c] = [[H|T1]|T2]$  .
3. ?-  $[a, b, [c]] = [H1|[H2|[H3|T]]]$  .

- Définir le predicat `concat(X, Y, Z)`, vrai si  $Z$  est la concatenation de  $X$  et  $Y$ .

- Écrire une requête en utilisant `concat`, qui enlève les trois dernières éléments d'une liste `L` donnée et produit une liste `L1`. (Indication: `L` est la concaténation de `L1` avec une liste à trois éléments)
- Écrire une requête en pour enlever les trois premiers et les trois dernières éléments d'une liste `L` et produire une liste `L1`.
- Définir le prédicat `dernier(X, L)` tel que `X` est le dernier élément de `L`. Donner deux versions de ce prédicat, avec et sans utilisation de `concat`.
- Définir deux prédicat `longueurpaire(L)` est `longueurimpaire(L)` qui sont vraies si leur argument est une liste avec un nombre pair (impair) d'éléments (n'utilisez pas d'opérations arithmétiques).
- Définir le prédicat `renverse(L, L1)` qui renverse des listes. Par exemple `renverse([a, b, c], L)` donne `L = [c, b, a]`. Indication: Pour ajouter un élément à la fin d'une liste un peut utiliser `concat`. Est-ce que votre programme marche aussi avec `renverse(L, [a, b, c])` ?

**Correction :**

```
concat([], L, L).
```

```
concat([X|L], L1, [X|L2]) :-
    concat(L, L1, L2).
```

```
concat(L1, [_ , _ , _], L).
```

```
concat([_ , _ , _ | L1], [_ , _ , _], L).
```

```
dernier1(X, L) :- concat(_, [X], L).
```

```
dernier2(X, [X]).
```

```
dernier2(X, [_ | L]) :- dernier2(X, L).
```

```
longueurpaire([]).
```

```
longueurpaire([_ | L]) :- longueurimpaire(L).
```

```
longueurimpaire([_ | L]) :- longueurpaire(L).
```

```
renverse1([], []).
```

```
renverse1([X|L], L1) :-
    renverse1(L, L2),
    concat(L2, [X], L1).
```