Les listes en Prolog

par Pierre Caboche

Date de publication : 18 Septembre 2006

Dernière mise à jour :

Cet article présente les listes en Prolog: principes de base et prédicats prédéfinis permettant la manipulation de listes.

Introduction

Le but de cet article est de fournir une ressource concernant l'utilisation des listes en Prolog. Cette ressource vient compléter les documentations existantes sur le sujet.

Nous commencerons par présenter les notions de base concernant les listes en Prolog. Ensuite, nous présenterons les prédicats existants concernant le traitement des listes en Prolog. Enfin, nous conclurons sur une discussion plus générale sur l'utilisation des listes en Prolog.

Une grande importance a été accordée aux prédicats servant à la manipulation des listes en Prolog, et ce pour plusieurs raisons:

1. il est inutile de réinventer la roue. De nombreux prédicats ont été implémentés en standard afin de faciliter le travail du programmeur
2. les prédicats implémentés en standard sont sûrs, très bien testés, bien optimisés, et offrent souvent plus de fonctionnalités que les prédicats implémentés par le programmeur
3. les prédicats sur les listes sont parfois implémentés à bas niveau, directement en C, ce qui leur confère d'excellentes performances. C'est le cas par exemple des prédicats permettant le tri

Il existe de nombreuses implémentations de Prolog différentes. C'est swi-prolog qui a été choisie pour écrire cet article. Pourquoi? Parce que swi-prolog est entièrement gratuite et propose de nombreuses fonctionnalités (en témoignent les nombreux prédicats servant au traitement des listes). Les prédicats décrits dans cet article sont ceux qu'on trouve dans swi-prolog.

Si vous utilisez une autre implémentation de Prolog, il est fort probable qu'elle dispose de sa propre implémentation des prédicats décrits dans cet article (si ce n'est pas le cas, permettez-moi d'émettre de sérieux doutes quant à la qualité de cette implémentation étant donné qu'un produit gratuit comme swi-prolog les fournit en standard). Sinon, swi-prolog étant diffusé sous licence LGPL, vous pouvez récupérer et utiliser les fichiers implémentant les différents prédicats décrits dans cet article (fichier "lists.pl" du répertoire "library").

Comme il a été dit plus haut, les différents prédicats portant sur les listes sont implémentés en standard afin de faciliter le travail du programmeur. Ces prédicats étant nombreux, la difficulté est alors de trouver celui dont on a besoin, ou plus simplement de savoir s'il existe avant de chercher à l'implémenter soi-même. Pour faciliter cette tâche, j'ai effectué plusieurs classements :

- par ordre alphabétique
- par catégorie
- sous forme de "How-to": on accède au prédicat qui convient à partir d'une liste de questions fréquemment posées, du genre: *"Comment trier une liste en fonction d'un ou plusieurs critères ?"*

Voici donc le plan pour cet article :

- Section 1: *Généralités sur les listes en Prolog*

  Présente les concepts de base sur les listes

- Section 2, 3, 4: *Prédicats sur les listes en swi-prolog*

  Les prédicats servant au traitement de listes en Prolog

- Section 5: *Exemple de prédicat sur les listes*

  Un exemple servant à montrer comment utiliser les prédicats sur les listes

- Section 6: *Listes et programmation en Prolog*

  Discussion sur l'usage des listes en Prolog

1 - Généralités

En Prolog, on ne connait que deux choses d'une liste:

- son premier élément (la tête, ou *head*)
- le reste de la liste (la queue, ou *tail*)

Le parcours d'une liste se fait de manière récursive sur la queue.

La liste vide se note: **[]**.

Une liste constituée d'au moins un élément se note **[T|Q]**.

Ici, **T** représente la tête et **Q** représente la queue. C'est aussi grâce à **[T|Q]** que l'on ajoute un élément en début de liste (par unification).

Les différents éléments d'une liste sont séparés par des virgules. Exemple: **[1, 2, 3, 4, 5, 6, 7]**.

Ainsi: **[1, 2, 3 | Q]** représente une liste dont les premiers éléments sont 1, 2, 3 et le reste de la liste est unifié avec la variable **Q**.

Voilà, c'est à peu près tout ce qu'il est nécessaire de savoir pour parcourir des listes en Prolog. Contrairement à d'autres langages, on n'a pas besoin d'avoir recours à des itérateurs pour parcourir une liste (comme c'est le cas en C++ ou en Java, et je ne parle même pas du C !). En Prolog, tout est déjà prévu pour cela dans le langage.

Ceux qui connaissent Caml ou tout autre langage fonctionnel seront avantagés car le parcours d'une liste est très similaire en Prolog. Pour les autres, ils trouveront un exemple de prédicat utilisant les listes un peu plus loin dans ce tutoriel.

Les prochaines sections de ce tutoriels portent sur les prédicats servant à la manipulation de listes en Prolog (et plus particulièrement en swi-prolog).

3 - How-to

Comment:

Accéder au i-ème élément d'une liste ?

Accéder au dernier élément d'une liste ?

Applanir une liste de listes ?

Appliquer un prédicat à tous les membres d'une liste ?

Calculer la somme de tous les éléments d'une liste ?

Créer une liste contenant tous les entiers compris dans l'intervalle [Min-Max] ?

Concaténer deux listes ?

Connaître la longueur d'une liste ?

Enumérer toutes les éléments d'une liste ?

Enumérer toutes les permutations possibles ?

Extraire un élément d'une liste et retourner la liste privée de cet élément ?

Extraire tous les éléments d'une liste vérifiant certaines propriétés ?

Faire la liste de toutes les solutions d'un prédicat ?

Fusionner deux listes triées ?

Inverser une liste ?

Insérer un élément en i-ème position d'une liste ?

Supprimer tous les doublons d'une liste ?

Supprimer d'une liste tous les éléments contenus dans une autre liste ?

Trier une liste ?

Trier une liste en fonction d'un ou plusieurs critères ?

Vérifier qu'un élément se trouve dans une liste ?

Vérifier qu'une variable est bien une liste ?

Vérifier qu'une liste ne comporte pas de doublons ?

Vérifier que tous les éléments d'une liste vérifient certaines propriétés ?

4 - Prédicats sur les listes (par catégorie)

4.1 - Propriétés de liste

is_list/1 *

```
is_list(+Terme).
```

Réussit si *Terme* est une liste.

is_set/1

```
is_set(+Terme).
```

Réussit si *Terme* est un set, c'est-à-dire une liste ne comportant pas de doublons.

length/2 *

```
length(+Liste, +Int).
```

Unifie *Int* avec la longueur de la liste.

Exemple

```
| ?- length([1,2,3,4,5], Long).

Long = 5

Yes
```

Peut également servir à créer une liste de longueur *Int* ne contenant que des variables libres.

Exemple

```
| ?- length(L, 3).

L = [_G316, _G319, _G322]

Yes
```

4.2 - Accès aux éléments

nth0/3

```
nth0(?Ind, ?List, ?Elem).
```

Réussit si l'élément d'indice *Ind* de la liste *List* s'unifie avec *Elem*. Les indices commencent à 0.

nth0/3 sert généralement à récupérer l'élément d'indice *Ind* dans la liste *List*.

nth0/3 peut servir à:

- Récupérer l'élément d'indice *Ind* :

Exemple
```
| ?-  nth0(2, [3,1,8,6,9], E).

E = 8
```

- Vérifier si l'élément d'indice *Ind* est *Elem* :

Exemple
```
| ?-  nth0(2, [3,1,8,6,9], 8).

Yes
```

- Enumérer les éléments d'une liste avec leurs indices :

Exemple
```
| ?- nth0(I, [3,15,8], E).

I = 0
E = 3 ;

I = 1
E = 15 ;

I = 2
E = 8 ;

No
```

- Insérer l'élément *Elem* à l'indice *Ind* dans la liste *List* si l'élément à l'indice *Ind* est une variable libre :

Exemple
```
| ?- length(L,5), nth0(1, L, 8888).

L = [_G444, 8888, _G450, _G453, _G456]

Yes
```

nth1/3

```
nth1(?Ind, ?List, ?Elem).
```

Pareil que *nth0*, mais avec les indices commençant à 1.

last/2

```
last(?List, ?Elem).
```

Unifie *Elem* avec le dernier élément de la liste *List*.

4.3 - Manipulations sur les listes

numlist/3

```
numlist(+Min, +Max, -List).
```

Unifie *List* avec la liste des entiers compris.

- *Min* <= *Max* (sinon échec)
- *Min* et *Max* sont des entiers (sinon échec)

Ce prédicat est très utile pour créer des intervalles de valeurs entières, pour symboliser le domaine d'une variable, par exemple.

Exemple
```
| ?- numlist(2,8, Domaine).

Domaine = [2, 3, 4, 5, 6, 7, 8]
```

Si l'usage d'une liste n'est pas justifié, préférer l'usage de *between/3*.

reverse/2

```
reverse(+List1, -List2).
```

Inverse l'ordre des éléments de *List1* et unifie le résultat dans *List2*.

Exemple

```
| ?- reverse([1,2,3,4,5], L2).

L2 = [5, 4, 3, 2, 1]
```

append/3

```
append(?List1, ?List2, ?List3).
```

Réussit si *List3* est la concaténation de *List1* et *List2*.

append/3 sert principalement pour ajouter le contenu de *List2* à la suite de *List1* et unifier le résultat dans *List3*.

Exemple 1

```
| ?- append([1,2,3], [4,5,6], L3).

L3 = [1, 2, 3, 4, 5, 6]
```

Cependant, append/3 ne se limite pas à cet usage et peut unifier indifféremment les variables *List1*, *List2* et *List3* :

Exemple 2

```
| ?- append(L1, L2, [1,2,3]).

L1 = []
L2 = [1, 2, 3] ;

L1 = [1]
L2 = [2, 3] ;

L1 = [1, 2]
L2 = [3] ;

L1 = [1, 2, 3]
L2 = [] ;
```

Exemple 2
```
No
```

flatten/2

```
flatten(+List1, -List2).
```

Applanit *List1* et unifie le résultat dans *List2*. *List1* peut contenir de nombreuses listes imbriquées récursivement. *flatten/2* extrait tous les éléments contenus dans *List1* et stoque le résultat dans une liste "plane" (à une seule dimension).

Exemple
```
| ?- flatten([[1,[2],3], [[4,5],[6,7]]], Flat).

Flat = [1, 2, 3, 4, 5, 6, 7]
```

permutation/2

```
permutation(?List1, ?List2).
```

Permet de vérifier si *List1* est une permutation de *List2* ou inversement.

Exemple
```
| ?- permutation([1,2,3,4,5], [5,3,1,4,2]).

Yes
```

Permet également d'énumérer toutes les permutations possibles de *List1* ou *List2*.

Exemple
```
| ?- permutation([1,2,3], Perm).

Perm = [1, 2, 3] ;

Perm = [2, 1, 3] ;

Perm = [2, 3, 1] ;

Perm = [1, 3, 2] ;

Perm = [3, 1, 2] ;

Perm = [3, 2, 1] ;

No
```

4.4 - Test d'appartenance

memberchk/2 *

```
memberchk(?Elem, ?List)
```

Permet de vérifier si *Elem* appartient à *List*. memberchk/2 parcours *List* et s'arrête au premier élément qui s'unifie avec *Elem*.

memberchk/2 se comporte comme member/2 mais sans point de choix :

Exemple
```
memberchk(Elem, List) :-
  member(Elem, List), !.
```

memberchk/2 retournera au maximum 1 solution (la première), alors que memberchk/2 retournera autant de solutions qu'il y a d'éléments qui s'unifient avec *Elem*.

nextto/3

```
nextto(?X, ?Y, ?List)
```

Réussit si *X* et *Y* se suivent dans *List*.

Permet :

- de vérifier si *X* et *Y* sont cote-à-cote dans *List*
- de déterminer quel élément suit *X* (ou précède *Y*)
- d'énumérer les éléments *X* et *Y* qui se suivent dans *List*

Exemple
```
| ?- nextto(3, 4, [1,2,3,4,5]).

Yes

| ?- nextto(X, Y, [1,2,3]).

X = 1
Y = 2 ;

X = 2
Y = 3 ;
```

Exemple

```
No
| ?- nextto(a, Y, [1,a,2,3,a,5,a,9,10]).
 Y = 2 ;
 Y = 5 ;
 Y = 9 ;
No
```

4.5 - Enumération d'éléments

member/2

```
member(?Elem, ?List).
```

Réussit si *Elem* avec un élément de la liste *List*.

Utilisé principalement pour énumérer les membres de la liste.

Exemple

```
| ?- member(X, [1,2,3,5,8]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 5 ;
X = 8 ;
No
```

member/2 peut être utilisé avec forall/2 pour appliquer des prédicats sur chacun des éléments de la liste (exemple 1) ou pour vérifier que tous les éléments d'une liste vérifient certaines propriétés (exemple 2) :

Exemple 1

```
...
forall(
  member(Var, Liste),
  (
    write(Var), nl
  )
),
...
```

Exemple 2

```
...
forall(
  member(Var, Liste),
  (
    5 < Var , Var < 10
  )
),
...
```

A ce sujet, voir aussi le prédicat maplist/2.

Les prédicats nth0/3 et nth1/3 permettent en plus de connaître l'indice où se trouve l'élément de la liste.

Le prédicat select/3, quant à lui, retourne la liste privée de l'élément.

select/3

```
select(?Elem, ?List, ?Rest).
```

Se comporte comme member/2, sauf qu'en plus il unifie *Rest* avec la liste *List* privée de l'élément *Elem*.

Exemple

```
?- select(E, [1,2,3,4], R).

E = 1
R = [2, 3, 4] ;

E = 2
R = [1, 3, 4] ;

E = 3
R = [1, 2, 4] ;

E = 4
R = [1, 2, 3] ;

No
```

Il permet en outre d'énumérer les insertions possibles dans une liste :

Exemple

```
| ?-  select(99, L, [1,2,3,4]).

L = [99, 1, 2, 3, 4] ;

L = [1, 99, 2, 3, 4] ;

L = [1, 2, 99, 3, 4] ;
```

Exemple

```
L = [1, 2, 3, 99, 4] ;
L = [1, 2, 3, 4, 99] ;
No
```

4.6 - Tri

msort/2 *

```
msort(+List, -Triee).
```

Trie la liste *List* et unifie le résultat dans *Triee*.

Le tri est très rapide car implémenté en interne en C. L'algorithme utilisé est le tri fusion naturel (rapide et stable). Pour ordonner les éléments, msort/2 se base sur l'ordre standard des termes défini dans Prolog.

sort/2 *

```
sort(+List, -Triee).
```

Se comporte comme msort/2, mais **supprime les doublons**.

keysort/2 *

```
keysort(+List, -Triee).
```

Le prédicat keysort/2 permet de trier sur un ou plusieurs critères. Les éléments de *List* sont de la forme *Clef-Valeur*. Le tri s'effectue sur les clefs et non sur les valeurs.

Il est possible d'utiliser plusieurs critères pour le tri, pour ce faire il suffit d'ajouter des clefs

(ex: *Clef1-Clef2-Clef3-...-ClefN-Valeur*).

Caractéristiques de keysort/2:

- keysort/2 conserve l'ordre des éléments de même clef (on dit que le tri est "stable")
- keysort/2 ne supprime pas les doublons
- keysort/2 est rapide car implémenté en interne

En effet, keysort/2 est basé sur un tri fusion naturel (rapide et stable) implémenté en C.

Exemple
```
| ?- keysort( [5-3-d, 3-4-c, 5-3-b, 5-1-a], L).

L = [3-4-c, 5-1-a, 5-3-d, 5-3-b]
```

predsort/3 *

```
predsort(+Pred, +List, -Triee).
```

Se comporte comme sort/2, mais utilise le prédicat *Pred* pour déterminer l'ordre.

Le prédicat Pred est de la forme *Pred(-Delta, +E1, +E2)*, où *E1* et *E2* sont les élements à comparer et *Delta* peut être l'un des trois symboles suivants: <, > ou =.

Si utilisé avec le prédicat *compare/3*, predsort/2 se comporte comme sort/2.

predsort/3 supprime les doublons.

4.7 - Maplist

Les prédicats maplist appliquent un prédicat sur tous les membres d'une liste ou jusqu'à ce que le prédicat échoue.

Pour ce faire, des arguments sont ajoutés au prédicat *Pred* grâce au prédicat call/[2..]. Ainsi, pour les prédicat d'arité supérieure à l'arité requise par *maplist*, les premiers arguments devront être unifiés.

Exemple
```
| ?- maplist(plus(8), [0, 1, 2], L).

X = [8, 9, 10]
```

maplist/2

```
maplist(+Pred, +List).
```

Applique le prédicat *Pred* à tous les membres d'une liste ou jusqu'à ce que *Pred* échoue, auquel cas le prédicat *maplist/2* échoue.

Ce prédicat permet de vérifier que tous les éléments d'une liste vérifient certaines propriétés (grâce au prédicat *Pred*).

Voici par exemple comment vérifier qu'une liste est composée uniquement d'entiers (grâce au prédicat integer/1) :

Exemple
```
| ?- maplist( integer, [15,8,5,6,9,4] ).

Yes
```

Et voici comment vérifier qu'une liste est composée uniquement de nombres suppérieurs à 3 (grâce au prédicat </2) :

Exemple
```
| ?- maplist( <(3), [15,8,5,6,9,4] ).

Yes
```

maplist/3

```
maplist(+Pred, ?List1, ?List2).
```

Appelle le prédicat *Pred* (d'arité 2 au minimum) avec comme arguments les membres de *List1* et *List2*.

Ce prédicat sert à appliquer le prédicat *Pred* sur les membres d'une liste et à mettre le résultat dans l'autre.

Exemple :

Exemple
```
| ?- maplist(plus(8), [0, 1, 2], L).

X = [8, 9, 10]
```

Ce prédicat est comparable à la fonction *map* du Caml.

maplist/4

```
maplist(+Pred, ?List1, ?List2, ?List3).
```

Appelle le prédicat *Pred* (d'arité 3 au minimum) avec comme arguments les membres de *List1*, *List2* et *List2*.

Ce prédicat sert à appliquer le prédicat *Pred* sur les membres de 2 listes et à mettre le résultat dans une troisième.

Exemple :

Exemple
```
| ?- maplist(plus, [0, 1, 2], [4, 8, 16], L).

X = [4, 9, 18]
```

Ce prédicat est comparable à la fonction *map2* du Caml.

4.8 - Extraction de données

sublist/3

```
sublist(+Pred, +List1, ?List2).
```

Extrait tous les éléments de *List1* pour lesquels *Pred* réussit, unifie le résultat dans *List2*.

Exemple: voici comment extraire d'une liste tous les éléments supérieurs à 5 :

Exemple
```
| ?- sublist( <(5), [15,2,8,5,6,9,4], L ).

L = [15, 8, 6, 9]
```

Explications sur le fonctionnement :

Dans l'exemple précédent, on utilise le prédicat </2. Exemple:

Exemple
```
| ?- <(1,5).

Yes
```

<(X,Y) est équivalent à X<Y.

On fixe le premier argument de </2 à 5 et c'est sublist qui ajoute le deuxième argument grâce au prédicat call/[2..]. Donc en fait, ce qui est appellé, c'est <(5,Y) avec Y appartenant à *List1* (donc *List2* est bien unifiée avec les nombres supérieurs à 5 contenus dans *List1*).

4.9 - Liste les solutions d'un prédicat

findall/3

Les prédicats findall/3, bagof/3 et setof/3 permettent de faire la liste des solutions d'un prédicat.

Ces prédicats peuvent être utiles pour dresser une liste de solutions possibles afin de trier ces solutions en fonction de certains critères (grâce au prédicat keysort/2, par exemple).

Ces prédicats peuvent également être utiles pour effectuer des tests unitaires en Prolog (on dresse la liste des solutions d'un prédicat grâce au prédicat findall/2, puis on compare cette liste avec la liste des solutions attendues).

Les prédicats findall/3, bagof/3 et setof/3 ont fait l'objet d'un autre article qui met en avant les différences entre ces prédicats et explique leur fonctionnement : findall, bagof et setof

bagof/3

Voir : findall, bagof et setof

setof/3

Voir : findall, bagof et setof

4.10 - Opérations sur les sets

Un set est une liste ne contenant aucun doublon.

is_set/1

list_to_set/2

```
list_to_set(+List, -Set).
```

Transforme une liste en set. En d'autres termes, supprime tous les doublons de *List* et unifie le résultat dans *Set*.

list_to_set/2 conserve l'ordre des éléments (si la liste contient des doublons, seul le premier est gardé)

subset/2

```
subset(+Subset, +Set).
```

Réussit si tous les éléments de *Subset* appartiennent à *Set*.

Note: subset/2 ne vérifie pas que *Subset* et *Set* ne contiennent pas de doublons, et ce pour de nombreuses raisons (y compris de performance).

union/3

```
union(+Set1, +Set2, -Set3).
```

Unifie *Set3* avec l'union de *Set1* et *Set2*. Les listes n'ont pas besoin d'être ordonnées.

Note: pour des raisons de performance, aucune vérification n'est faite pour garantir que *Set1* et *Set2* ne contiennent pas de doublons. Si *Set1* et *Set2* ne contiennent pas de doublons, *Set3* ne contiendra pas de doublon. Si *Set1* et *Set2* contiennent des doublons, *Set3* est susceptible de contenir des doublons. Utilisez list_to_set/2 pour supprimer les doublons.

intersection/3

```
intersection(+Set1, +Set2, -Set3).
```

Unifie *Set3* avec l'intersection de *Set1* et *Set2*. Les listes n'ont pas besoin d'être ordonnées.

Note: pour des raisons de performance, aucune vérification n'est faite pour garantir que *Set1* et *Set2* ne contiennent pas de doublons. Si *Set1* et *Set2* ne contiennent pas de doublons, *Set3* ne contiendra pas de doublon. Si *Set1* et *Set2* contiennent des doublons, *Set3* est susceptible de contenir des doublons. Utilisez list_to_set/2 pour supprimer les doublons.

subtract/3

```
subtract(+Set, +Delete, -Rest).
```

Supprime de *Set* tous les éléments contenus dans *Delete* et unifie le résultat dans *Rest*.

Note: pour des raisons de performance, aucune vérification n'est faite pour garantir que *Set* et *Set2* ne contient pas de doublons.

4.11 - Divers

sumlist/2

```
sumlist(+List, -Sum).
```

Unifie *Sum* avec la somme de tous les éléments de List.

- *List* est une liste de nombres (sinon, échoue)

Exemple

```
| ?- sumlist([2,4,6,8,10], Sum).

Sum = 30
```

merge/3 *

```
merge(+List1, +List2, -List3).
```

List1 et *List2* sont des listes triées. merge/3 fusionne *List1* et *List2* pour en faire en une liste triée et unifie le résultat dans *List3*. merge/3 ne supprime pas les doublons.

- *List1* et *List2* sont triées (sinon, comportement aléatoire)

Le prédicat merge/3 est utilisé par le prédicat msort/2.

merge_set/3 *

```
merge_set(+Set1, +Set2, -Set3).
```

Set1 et *Set2* sont des listes triées, supposées sans doublons. merge_set/3 fusionne *Set1* et *Set2* pour en faire en une liste triée sans doublons et unifie le résultat dans *Set3*.

- *Set1* et *Set2* sont triés (sinon, comportement aléatoire)
- *Set1* et *Set2* ne contiennent pas de doublons

- Si *Set1* et *Set2* ne contiennent pas de doublons, alors *Set3* ne contiendra pas de doublons
- Si *Set1* ou *Set2* contiennent des doublons, alors *Set3* contiendra des doublons

Le prédicat merge_set/3 est utilisé par le prédicat sort/2.

5 - Exemple

Parmi les prédicats portant sur les listes il en manquait un permettant de mélanger une liste. Qu'à cela ne tienne ! Cela nous permettra de voir comment écrire un prédicat sur les listes en Prolog.

shuffle/2

Voici donc le prototype du prédicat shuffle/2 (que nous allons implémenter), et sa description :

```
shuffle(+List, -Shuffled).
```

Change l'ordre des éléments de *List* et unifie le résultat dans *Shuffled* (le mélange se fait aléatoirement).

Voici maintenant le code du prédicat shuffle/2 :

Code du prédicat shuffle/2

```
shuffle(List, Shuffled) :-
  length(List, Len),
  shuffle(Len, List, Shuffled).


shuffle(0, [], []) :- !.

shuffle(Len, List, [Elem|Tail]) :-
  RandInd is random(Len),
  nth0(RandInd, List, Elem),
  select(Elem, List, Rest),
  !,
  NewLen is Len - 1,
  shuffle(NewLen, Rest, Tail).
```

Fonctionnement:

Dans shuffle/2, on invoque length/2 et on appelle shuffle/3.

Dans shuffle/3, on génère un indice *RandInd* entre 0 et la taille de la liste grâce à random/1. Avec nth0/3, on récupère la valeur située à l'indice *RandInd*, puis grâce à select/3 on extrait cette valeur ainsi que la liste privée de cette valeur (*Rest*).

On invoque ensuite shuffle/3 sur *Rest* (de longueur *Len*-1) jusqu'à obtenir une liste vide (dans ce cas, on s'arrête).

C'est en extrayant les éléments de cette façon que l'on obtient la liste mélangée.

Le prédicat shuffle/3 a été introduit dans un but d'effectuer une optimisation. En effet, on ne calcule la longueur de la liste qu'une seule fois (au début, dans shuffle/2), ce qui évite de la parcourir à chaque itération.

Maintenant, la partie ci-dessous pourrait être optimisée afin de ne parcourir la liste qu'une seule fois et pas deux :

```
...
nth0(RandInd, List, Elem),
select(Elem, List, Rest),
...
```

Cependant, cela nous obligerait à écrire un nouveau prédicat et nous prive de l'opportunité de vous montrer l'utilisation des prédicats nth0/3 et select/3 dans un exemple concrêt. Aussi, je vous laisse cette optimisation à titre d'exercice.

6 - Les listes et la programmation en Prolog (Conclusion)

Comme je l'ai dit dans les principes de base sur les listes (section 1), en Prolog, on ne connait que les premiers éléments d'une liste (la tête) et c'est de manière récursive qu'on accède au reste de la liste (la queue). On n'a pas d'accès direct aux éléments d'une liste et pour accéder au ième élément, on utilise les prédicats nth0/3 ou nth1/3 (programmés récursivement).

Tout cela est extrèmement long et couteux.

Aussi, autant que faire se peut, il faut garder à l'esprit qu'une liste est censée être parcourue séquentiellement et optimiser ses programmes en conséquence, afin d'éviter les aller-retours inutiles.

Maintenant, je souhaiterais attirer votre attention sur un deuxième point: la manipulation de listes requièrt beaucoup de mémoire. En effet, en Prolog les manipulations de listes nécessitent souvent la création de nouvelles listes, ce qui peut rapidement engendrer un coût énorme en terme de mémoire, qui peut se traduire par l'arrêt du programme par manque de mémoire.

Dans certains cas, l'utilisation de listes est extrêmement utile, par exemple pour effectuer des tris. Cependant, dans d'autres cas, il est tout à fait possible d'exploiter certaines spécificités du langage Prolog pour se passer de listes. Par exemple, nous avons vu qu'il était possible d'effectuer des opérations sur les sets (union/3, intersection/3, subtract/3, etc.) or ces opérations sont également réalisables au moyens de prédicats, comme indiqué dans l'article Prolog et algèbre relationnelle.

Une question fondammentale en Prolog est: "Comment représenter le problème à résoudre?". De la réponse à cette question dépendra l'implémentation et donc les performances du programme.

Les programmeurs habitués à des langages comme C, Java ou autres ont tendance à se réfugier derrière les listes car elles leur rappellent certaines structures de données existantes dans d'autres langages (les tableaux). Cependant, il existe d'énormes différences entre manipuler des tableaux en C et manipuler des listes en Prolog (pas d'accès direct, grande consommation de mémoire...). A l'inverse, Prolog offre des mécanismes qui n'ont aucun équivalent dans d'autres langages (bases de connaissance et de règles, retour sur trace, etc.) et qu'il faut apprendre à exploiter. Bref, pour faire du Prolog, il faut mettre de coté certaines habitudes issues d'autres langages.

En résumé, il est nécessaire de se rappeller que :

1. les listes ne sont pas le seul moyen pour stocker de l'information en Prolog
2. quand on a besoin d'utiliser des listes, il faut penser à exploiter les spécificités des listes (accès aux éléments par la tête de liste) ainsi que les prédicats mis à notre disposition pour nous faciliter le travail
3. la quantité de mémoire nécessaire pour manipuler les listes ne doit pas être négligée

Remerciements

Je tiens à remercier Trap D et Katyucha pour la relecture de cet article.