Les listes

Représentation des listes:

Les objets élémentaires en Prolog sont les **atomes**, les **variables**, les **nombres**. Les **structures** servent à représenter des objets qui ont plusieurs composantes.
La **liste** est une structure élémentaire de données, employée en programmation non numérique. Une liste est une suite d'un nombre quelconque d'objets.

```
Ex.1.: [a, b, c, d]
Ex.2.: [pomme, banane, fraise, kiwi, orange].
```

- le premier élément est appelé la **tête** de la liste,
- ce qui reste de la liste est appelé la **queue** de la liste.

Les ensembles peuvent être représentés par les listes, mais l'ordre est essentiel dans une liste alors que dans un ensemble l'ordre compte peu. Un même élément peut apparaître plusieurs fois. Les opérations que l'on peut effectuer sur les listes sont similaires aux opérations ensemblistes :

- la vérification de la présence d'un élément dans une liste, ou appartenance ensembliste.
- la concaténation de deux listes, pour en obtenir une troisième, ce qui correspond à l'union ensembliste.
- l'ajout d'un nouvel élément ou la suppression d'un élément dans une liste.

Pour notre exemple `[a, b, c, d]` `a` est la tête de la liste et `[b,c,d]` est la queue de la liste.
idem pour `[pomme, banane, fraise, kiwi, orange], pomme` est la tête de la liste et `[banane, fraise, kiwi, orange]` est la queue de la liste.

En général, la tête peut être n'importe un élément quelconque comme un arbre ou une variable, mais il faut que la queue soit une liste. En combinant la tête et la queue par un foncteur spécial on obtient une **structure**.

`.(Tête, Queue)`

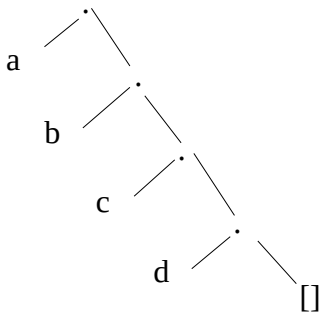

figure 1. représentation arborescente de la liste [a,b,c,d].

- **Liste vide : []**
- **Cas général : [Tete|Queue]**
- **[a,b,c] = [a|[b|[c|[]]]]**

Puisque queue est une liste, elle est soit vide soit composée d'une tête et d'une queue, ce qui permet de représenter les listes d'une longueur quelconque.

.(a, .(b, .(c, .(d, []))))

On peut constater que la liste vide apparaît parmi les termes, car l'avant-dernière queue est la liste constituée d'un seul objet:

[d]

Donc, la queue de cette liste est la liste vide :

[d] = .(d, [])

Et voici un exemple testé avec l'interprète prolog :

```
?- Liste1=[a,b,c].

Liste1 = [a, b, c]

Yes

?- L1 = .(a, .(b, .(c, [] ))).

L1 = [a, b, c]

Yes
```

En Prolog, il n'y a pas la possibilité d'utiliser de tableaux (par exemple pour accéder directement à la 50ème valeur). Par contre, il est très facile de les remplacer par des listes. L'avantage des listes est que leur longueur est dynamique. Leur utilisation est récursive (on ne peut accéder à la 50ème qu'après avoir accédé aux 49 précédentes).

Une liste constante est représentée entre crochets ([ ] signifiant ensemble vide) :

```
etudiants(cours1,[jean,lucie,yves,anne,alain,julie,marc,luc]).
etudiants(cours2,[jo,cathy,ahmed,louis,paul,eve,marie,rose]).
```

Lorsque l'on représente une liste par une variable, deux écritures sont possibles : soit par un nom de variable (commençant par une majuscule), soit par [Tete|Queue], où Tete (du type élément) est le premier de la liste, et Queue (de type liste d'éléments) le reste de la liste.

exemple :

```
afficher([T|Q]) :-
   writeln(T),nl,
   afficher(Q).
afficher([]).
```

essayez le but `afficher([jean, jacques, jules]).`

autre exemple, très utile :

```
appartient_a(X,[X|_]).
appartient_a(X,[_|Queue]) :- appartient_a(X,Queue).
```

Le but `appartient_a/2`, existe en Prolog défini comme `member/2`.

Les listes peuvent servir à représenter des ensembles (les éléments appartiennent ou non à différents ensembles, que l'on peut combiner par l'union ou l'intersection), mais aussi être ordonnée (et entre autres pourront être triées). Je vous propose ici un certain nombre de règles pour tester des listes.

Pour simplifier la gestion des hommes dans une famille par exemple, on peut utiliser une liste :

```
hommes([marc, luc, jean, jules, léon, loic, gerard, hervé, jacques,
paul]).
appartient_a(X,[X|_]).
appartient_a(X,[_|Queue]) :- appartient_a(X,Queue).
est_un_homme(X) :-
      hommes(Liste),
      appartient_a(X,Liste).
```

Observez et vérifiez les exemples qui suivent :

- `[X|L] = [a,b,c]   -> X = a, L= [b,c]`
- `[X|L] = [a]       -> X = a, L = []`
- `[X|L] = []        -> échec`
- `[X,L] = [a,b,c]   -> échec`
- `[X, Y|L] = [a,b,c]-> X = a, Y = b, L = [c]`
- `[X|L1] = [X, Y|L2]-> L1 = [Y|L2]`

Les programmes qui suivent décrivent quelques fonctionnalités de manipulation des listes :

Somme des éléments d'une liste

```
somme_liste([], 0).
somme_liste([X|L], Somme) :-
                somme_liste(L, Somme1)
                Somme is Somme1 + X.
```

La première ligne somme_liste([], 0), donne la somme 0 pour une liste vide. C'est aussi le test d'arrêt pour l'interprète, lorsqu'il atteint le dernier élément d'une liste non vide pour laquelle la deuxième ligne s'applique. On lit la somme d'une liste [X|L], dont le premier élément est X et le reste est L, est égale à Somme si et seulement si on peut avoir la somme du reste de la liste L avec une somme intermédiaire Somme1, et la Somme (finale) est égale à l'addition de la somme intermédiaire Somme1 avec l'élément X. A chaque fois que l'interprète vérifie que la liste n'est pas vide, applique la deuxième règle où X prend la valeur du 1er élément de la nouvelle liste restante.

Et voilà les réponses de l'interprète :

```
?- [somme_liste].
% somme_liste compiled 0.00 sec, 888 bytes

Yes

?- somme_liste([1,2,3,4], S).

S = 10

Yes
```

utilisation de 'listing'

```
?- listing.

somme_liste([], 0).
somme_liste([A|B], C) :-
        somme_liste(B, D),
        C is D+A.

Yes
```

utilisation de liste vide

```
?- somme_liste([], S).

S = 0

Yes
```

utilisation de 'non arithmetic'

```
?- somme_liste([a,b], S).
ERROR: Arithmetic: `b/0' is not a function
^  Exception: (9) _L174 is 0+b ? creep
```

Lorsque l'on demande de faire la somme avec des éléments non arithmetic, nous avons une erreur, car l'addition s'effectue seulement avec des éléments arithmétiques.
Lorsque l'on utilise un prédicat dont on ne connaît pas l'usage on fait appel à l'aide de prolog afin de vérifier la syntaxe et la sémantique de ce prédicat, comme le montre l'exemple qui suit :

Utilisation de 'help'

```
?- help(listing).
listing(+Pred)
    List  specified predicates  (when an  atom is  given all  predicates
    with  this name will  be listed).   The listing  is produced on  the
    basis of the  internal representation, thus losing user's layout and
```

```
   variable name information.  See also portray_clause/1.

listing
   List all predicates of the database using listing/1.


Yes
```

L'interprétation interne change les noms des variables utilisées, par rapport à notre programme (vérifier, exo sur somme_liste.pl).
On peut demander de l'aide pour des prédicats que l'on a vu mais on ne se souvient plus l'usage en tapant help et donnant le nom du prédicat comme argument, par exemple :

vérification d'un prédicat

```
?- help(append).
append(+File)
   Similar  to tell/1,  but positions the  file pointer  at the end  of
   File  rather than truncating an existing  file.  The pipe  construct
   is not accepted by this predicate.

append(?List1, ?List2, ?List3)
   Succeeds  when List3  unifies with  the concatenation  of List1  and
   List2.   The  predicate can be  used with any instantiation  pattern
   (even three variables).


Yes
```

L'exemple nous montre que le prédicat append peut s'utiliser de deux façons différentes, avec les listes prend 3 arguments : deux listes distinctes qui fusionnent et produisent une troisième nouvelle liste fusion de deux listes append/3. Si l'on désire voir le codage de ce prédicat prédéfini il suffit de taper listing et comme argument le nom du prédicat avec son arité :

concaténation de deux listes

L'exemple qui suit donne le codage pour le prédicat de la concaténation *append* avec *3* arguments:

```
?- listing(append/3).


lists:append([], A, A).
lists:append([A|B], C, [A|D]) :-
       append(B, C, D).

Yes

?- append([a,b,c], [d,e,f], L).
```

```
L = [a, b, c, d, e, f]

Yes
```

Le prédicat append/3 est complément symmétrique et peut donc être utilisé pour

- **trouver le dernier élément de la liste :**

```
[debug]  ?- append(_, [X], [a,b,c,d]).

X = d

Yes
?-
```

- **couper une liste en sous-listes :**

```
?- append(L2,L3,[b,c,a,d,e]),append(L1,[a],L2).

L2 = [b, c, a]
L3 = [d, e]
L1 = [b, c]

Yes
?-
```

appartenance à une liste

L'exemple qui suit donne le codage pour le prédicat d'appartenance :

```
?- listing(member/2).


lists:member(A, [A|B]).
lists:member(A, [B|C]) :-
        member(A, C).

Yes
?-
```

longueur d'une liste

L'exemple qui suit donne le codage pour le prédicat de longueur d'une liste :

```
longueur([],0).
longueur([X|Xs], N):- longueur(Xs, N1), N is N1 + 1.
```

la longueur d'une liste vide est égale à 0.
La longueur d'une liste non vide composée de X (le s signale le pluriel) et un nombre inconnu.
Lorsque l'itération commence grâce à la récursivité.

test ou génération

```
[debug]  ?- trace.

Yes
[trace]  ?- somme_liste([1,2,3], S).
    Call: (7) somme_liste([1, 2, 3], _G323) ?
creep
   Call: (8) somme_liste([2, 3], _L212) ? creep
   Call: (9) somme_liste([3], _L232) ? creep
   Call: (10) somme_liste([], _L252) ? creep
   Exit: (10) somme_liste([], 0) ? creep
^  Call: (10) _L232 is 0+3 ? creep
^  Exit: (10) 3 is 0+3 ? creep
   Exit: (9) somme_liste([3], 3) ? creep
^  Call: (9) _L212 is 3+2 ? creep
^  Exit: (9) 5 is 3+2 ? creep
   Exit: (8) somme_liste([2, 3], 5) ? creep
^  Call: (8) _G323 is 5+1 ? creep
^  Exit: (8) 6 is 5+1 ? creep
   Exit: (7) somme_liste([1, 2, 3], 6) ? creep

S = 6

Yes
[debug]  ?-
```

Le prédicat *trace* permet de 'tracer' le programme et visualiser la façon dont l'interprète procède à l'évaluation de l'expression demandée.

Ajout et suppression d'un élément

La méthode la plus simple pour ajouter un élément à une liste est de le placer au début de la liste, de telle sorte à ce qu'il devienne la tête de la liste : **[X | L]**
Il ne faut donc aucune procédure pour cette opération. Mais on peut l'exprimer sous forme d'un fait si nécessaire :

```
ajout(X, L, [X|L]).
```

Pour supprimer un objet d'une liste on peut le programmer avec la relation :

```
effacer(X, L, L1).
```

où L1 est la nouvelle liste sans l'élément X qui a été effacé. Le prédicat effacer a une structure similaire à celle de member/2, car on se trouve face à une alternative :

- si X est la tête de la liste, alors le résultat est la queue de la liste.
- sinon, on cherche à supprimer X de la queue de la liste.

```
effacer(X, [X|Reste] Reste).
effacer(X, [Y|Reste], [Y|NouvReste])::-
      effacer(X, Reste, NouvReste).
```

Comme **member**, **effacer** a par nature un comportement non déterministe. Si plusieurs occurrences

de X se présentent dans la liste alors effacer pourra supprimer n'importe laquelle, grâce au *back-tracking* (retour en arrière). Chacune de séquences exécutées n'effacera qu'une seule occurrence de X sans toucher aux autres. Regardez l'exemple :

```
?- [effacer].
% effacer compiled 0.00 sec, 916 bytes

Yes
?- eff(a, [a,b,a,a], L).

L = [b, a, a] ;

L = [a, b, a] ;

L = [a, b, a] ;

No
?-
```

eff échouera si la liste ne contient pas l'élément à effacer :

```
?- eff(a, [b,c,d], L).

No
```

On peut utiliser eff dans l'autre sens, afin d'insérer un élément n'importe où dans une liste, regardez l'exemple qui suit :

```
?- eff(a, L, [1,2,3]).

L = [a, 1, 2, 3] ;

L = [1, a, 2, 3] ;

L = [1, 2, a, 3] ;

L = [1, 2, 3, a] ;

No
?-
```

Définition d'un prédicat

Lorsque l'on souhaite la définition d'un prédicat nouveau on commence par se poser de bonnes questions :

- **comment ai-je l'intention de l'utiliser**
- **quelles sont les données**
- **quels sont les résultats**
- **faut-il avoir plusieurs solutions ?**

Si l'on veut une seule solution if faut faire des cas exclusifs.

Exercices à faire

1. Créer le prédicat **permuter** à deux arguments : ce sont deux listes telles que l'une est la permutation de l'autre.

2. Définissez deux prédicats :

   longueurpaire(Liste) et longueurImpaire(Liste)

   qui sont vérifiés lorsque le nbr d'éléments de la liste est pair ou impair.

3. Définissez le prédicat :

   retourner(Liste, ListeRetournee)

   qui renverse les listes. Par exemple retourner([a,b,c], [c,b,a]).