Listes

- Les listes sont des structures de données de base en Prolog

- Syntaxe :
  - [ ] est la liste vide
  - [Tête | Queue] est la liste où le premier élément est Tête et le reste de la liste est Queue
  - [a,b,c,3,'Coucou'] est une liste de constantes

- *Exemples* :
  - **`[X|L] = [a,b,c]`** *donne* **`YES {X=a,L=[b,c]}`**
  - **`[X|L] = [a]`** *donne* **`YES {X=a,L=[]}`**
  - **`[X|L] = []`** *donne* **`NO`**
  - **`[X,L] = [a,b,c]`** *donne* **`NO`**
  - **`[X,Y|L] = [a,b,c]`** *donne* **`YES {X=a,Y=b,L=[c]}`**
  - **`[X|L] = [A,B,C]`** *donne* **`YES {X=A, L=[B,C]}`**

Listes imbriquées

- Il est possible d'imbriquer les listes :
  - **`[X|L] = [a,[]].`** *donne* **`YES {X=a, L=[[]]}`**
  - **`[X|L] = [a,[b,c],d].`** *donne* **`YES {X=a, L=[[b,c],d]}`**
  - **`[X] = [a|[]].`** *donne* **`YES {X=a}`**
  - **`[X,Y] = [a|[b|[]]].`** *donne* **`YES {X=a, Y=b}`**
  - **`[X,Y] = [a|[b]].`** *donne* **`YES {X=a, Y=b}`**
  - **`[X,a] = [b,Y].`** *donne* **`YES {X=b, Y=a}`**

- *Calcul de la longueur d'une liste* : **`length(?list,?integer)`** réussit si le deuxième argument est la longueur de la liste passée en premier argument (**`length`** est prédéfini)

```
mylength([],0).

mylength([_|Y],L) :- mylength(Y,L2),L is L2 + 1.
```

Exemple_Prolog

Appartenance à une liste

- *Appartenance à une liste* : **`member(?term,?list)`** réussit si le premier argument est membre de la liste passée en deuxième argument (**`member`** est prédéfini)

```
mymember(X,[X|_]).

mymember(Y,[X|L]) :- X \== Y, mymember(Y,L).
```

- *Exemples* :
  - **`mymember(e,[a,b,e]).`** *donne* **YES**
  - **`mymember(R,[a,b,[c]]).`** *donne* **YES {R=a}** *ou* **{R=b}** *ou* **{R=[c]}**
  - **`mymember(e,L).`** *donne* **YES {L=[e|_]}** *ou* **{L=[_,e|_]}** ...


Exemple_Prolog

- Prédicat qui prend en compte toutes les occurences :

```
mymember2(X,[X|_]).

mymember2(Y,[_|L]) :- mymember2(Y,L).
```


Concaténation de listes

- *Concaténation de listes* : **`append(?list,?list,?list)`** réussit si la troisième liste est la concaténation de la première et de la deuxième (**append** est prédéfini)

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

myappend([X|L1],L2,[X|L3]) :- myappend(L1,L2,L3).
```

- *Exemples* :
  - **`myappend([a,b,[g]],[i,j],L).`** *donne* **YES {L=[a,b,[g],i,j]}**
  - **`myappend(L,[a,b],[t,u,a,b]).`** *donne* **YES {L=[t,u]}**


Exemple_Prolog

- Nouvelle version du prédicat d'appartenance :

```
mymember3(X,L) :- myappend(_,[X|_],L).
```

Inversion d'une liste

- *Inversion d'une liste* : **`reverse(?list,?list)`** réussit si la deuxième liste est l'inverse de la première (**`reverse`** est prédéfini)

```
myreverse([],[]).

myreverse([X|L1],L2) :- myreverse(L1,L3),append(L3,[X],L2).
```

- *Exemples* :
  - **`myreverse([a,b,[c]],L).`** *donne* **`YES {L=[[c],b,a]}`**
  - **`myreverse([a,b,c],[X|Y]).`** *donne* **`YES {X=c, Y=[b,a]}`**
  - **`myreverse([a,b,c,d],[d,c|L]).`** *donne* **`YES {L=[b,a]}`**
  - **`myreverse(L,[a,b,c]).`** *donne* **`YES {L=[c,b,a]}`** *ou* **`stackoverflow`**


Exemple_Prolog

Dernier élément d'une liste

- La syntaxe [X|Y] permet de récupérer le premier élément d'une liste

- *Dernier élément d'une liste* : l**`ast(?list,?term)`** réussit si le deuxième argument est le dernier élément de la liste (**`last`** est prédéfini)

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

mylast([Y|L],X) :- Y \== X, mylast(L,X).
```


Exemple_Prolog

- *Exemples* :
  - **`mylast([a,b,c],[c]).`** *donne* **`NO`**
  - **`mylast([],X).`** *donne* **`NO`**
  - **`mylast(L,a).`** *donne* **`YES {L=[a]}`** ***ou*** **`{L=[_,a]} ...`**

Insertion dans une liste

- *Insertion dans une liste* : **`insert(?term,?list,?list)`** réussit si la deuxième liste est le résultat de l'insertion du terme dans la première liste

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

insert(X,[Y|L1],[Y|L2]) :- insert(X,L1,L2).
```

- *Exemples* :
  - **`insert(a,[b,c],L)`** *donne* **YES `{L=[a,b,c]}`** *ou* **`{L=[b,a,c]}`** *ou* **`{L=[b,c,a]}`**
  - **`insert(X,[a,b,c],[a,b,d,c])`** *donne* **YES `{X=d}`**
  - **`insert(a,L,[b,c,a])`** *donne* **YES `{L=[b,c]}`**


Exemple_Prolog

Permutation d'une liste

- *Permutation d'une liste* : **`permutation(?list,?list)`** réussit si la deuxième liste est une permutation de la première (**`permutation`** est prédéfini)

```
mypermutation([],[]).

mypermutation([X|L1],L2) :- mypermutation(L1,L3),insert(X,L3,L2).
```

- *Exemples* :
  - **`mypermutation([a,b,c],L)`** *donne* **YES `{L=[a,b,c]}` ou `{L=[b,a,c]}` ...**
  - **`mypermutation([a,b,c,d],[c,d,b,a])`** *donne* **YES**


Exemple_Prolog

Tri d'une liste

- *Exemple du tri par insertion* : on enlève le premier élément, on tri le reste de la liste, et on insère l'élément à sa place dans la liste triée

- **`insertion(+atom,?list,?list)`** réussit si la deuxième liste est le résultat de l'insertion de l'atome, à sa place, dans la première liste

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

insertion(X,[Y|L],[X,Y|L]) :- X =<Y.

insertion(X,[Y|L],[Y|L1]) :- X > Y, insertion(X,L,L1).
```

- **`tri_insertion(?list,?list)`** réussit si la deuxième liste est le résultat du tri par insertion de la première liste

```
tri_insertion([],[]).

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


Différence de liste (1/2)

- Concaténation de deux listes

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

myappend([X|L1],L2,[X|L3]) :- myappend(L1,L2,L3).
```

- **Notoirement inefficace !**
  - "parcours" de la liste élément par élément

- *Le problème*
  - pas de « pointeur » sur le dernier élément
  - pas de possibilité de *modification* d'une variable logique uniquement **instanciation d'une inconnue**

Différence de liste (2/2)

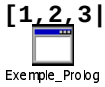
- *Une solution : une nouvelle structure de liste*
  - Idée : conserver une partie ***inconnue*** *pour instanciation au dernier moment*
  - ***Liste de différence***
    - on **nomme** la fin de la liste
    - `[a,b,c]` est représentée par `[a,b,c|Xs]-Xs`
    - la liste vide est représentée par `Xs-Xs`
- La concaténation peut être écrite en une seule ligne

```
appendDiff(Ls-Xs,Xs-Rs,Ls-Rs).
```

- *Exemple :*

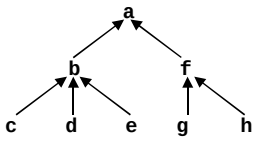
```
?- append([a,b,c|Xs]-Xs, [1,2,3|Ys]-Ys, R).
```

donne `YES {R = [a,b,c,1,2,3|Ys]-Ys,Xs = [1,2,3|Ys]}`


Exemple_Prolog

Arbres en Prolog

- *Représentation des arbres en Prolog* :
  - représentation par une liste de listes (plus simple)
  - représentation par un prédicat (plus explicite?)
    `arbre(noeud,liste_des_fils)`
  - arbre binaire avec prédicat
    `arbre_binaire(noeud,fils_gauche,fils_droit)`
- *Exemples :*
  - [a,[b,[c,d,e]],[f,[g,h]]]
  - arbre(a,[arbre(b,[c,d,e]),arbre(f,[g,h])])



- Le Prolog est bien adapté pour le parcours et la manipulation d'arbre!

Termes structurés

- *Notion de foncteur* :
  - pour représenter des termes structurés (objets avec attributs, arbres, ...), on peut utiliser des **listes** (de listes)
  - mais on préfère **structurer** l'information dans des prédicats pour rendre plus explicite la nature des données

- *Exemple : une famille à trois enfants peut être représentée par*
  - *une liste* `[Mere,Pere,Enf1,Enf2,Enf3]`
  - *un prédicat* `famille(mere(X),pere(Y),enfant(A),enfant(B),enfant(C))`

Manipulation de foncteur

- `functor(+nonvar, ?atomic, ?integer)` ou `functor(-nonvar, +atomic, +integer)` réussit si le premier terme a pour nom le deuxième et pour arité le troisième

- *Exemples* :
  - `functor(pere(tom,pouce), A,B).` *donne* `YES {A=pere, B=2}`
  - `functor(P,pere,2).` *donne* `YES {P=pere(_,_)}`

- `arg(+integer, +compound_term, ?term)` réussit si l'argument d'indice le premier paramètre du deuxième terme est le troisième terme

- *Exemple* :
  - `arg(2,pere(tom,pouce),T).` *donne* `YES {T=pouce}`

Construction/Déconstruction

- *Construction/déconstruction de foncteurs* : **`=..(+var,?list)`** ou **`=..(-var,+list)`** réussit si la liste a pour tête l'atome correspondant au principal foncteur du premier argument et pour queue la liste des arguments de ce terme

- *Exemples* :
  - **`=..(X,[date, 20,mars,2006]).`** *donne* **`YES {X = date(20,mars,2006)}`**
  - **`date(20,mars,2006) =.. L.`** *donne* **`YES {L=[date, 20,mars,2006]}`**

Prédicat du second ordre (1/2)

- Les prédicats prédéfinis **`findall`**, **`bagof`** et **`setof`** permettent de récupérer toutes les unifications solutions d'un but

- **`findall(?term,+callable_term,?list)`** réussit si la liste contient toutes les valeurs du terme obtenues par application du but sur le terme

- *Exemple* :
  - **`findall(X,permutation([a,b,c],X),L).`** *donne* **`YES {L=[[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]}`**
  - **`findall(X,append(X,Y,[a,b,c]),L).`** *donne* **`YES {L=[[],[a],[a,b],[a,b,c]]}`**

Prédicat du second ordre (2/2)

- **`bagof(?term,+callable_term,?list)`** réussit si la liste contient, pour une instanciation des variables libres du but, les valeurs du terme possiblement obtenues par application du but sur le terme

- *Exemples* :
  - `bagof(X,permutation([a,b,c],X),L).` *donne* **YES**
    `{L=[[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]}`
  - `bagof(X,append(X,Y,[a,b,c]),L).` *donne* **YES**
    `{L=[[a,b,c]], Y=[]}` *ou*
    `{L=[[]], Y=[a,b,c]}` *ou*
    `{L=[[a]], Y=[b,c]}` *ou*
    `{L=[[a,b]], Y=[c]}`

- **`setof(?term,+callable_term,?list)`** est équivalent à **`bagof`** mais la liste de solutions est triée (les éléments en double sont éliminés)

Prédicats de contrôle

- Prolog offre des prédicats pour contrôler le backtracking :
  - **`true`** : réussit toujours
  - **`fail`** : ne réussit jamais (donc provoque le backtracking)
  - **`repeat`** : réussit toujours mais crée toujours un point de choix
  - **`!`** : la "coupure", réussit toujours et supprime tous les points de choix créés depuis le début de l'exploitation du paquet des clauses du prédicat où le coupe-choix figure.

- Ces prédicats servent à
  - contrôler le mécanisme de résolution
  - optimiser la recherche de solution
  - créer des structures de contrôle

Principe de la coupure

- *Coupure* : Le coupe-choix permet de signifier à Prolog qu'on ne désire pas conserver les points de choix en attente
  - en élaguant les branches de l'arbre de recherche
  - rend les programmes plus simples et efficaces

- *Exemple* :

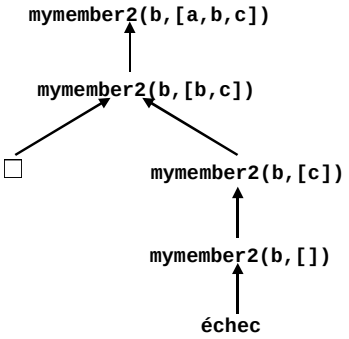
```
tete(...) :- pr1(...), pr2(...), !, pr3(...), pr4(...).
```

  - *si* **pr1** *ne réussit pas, on peut tenter de démontrer* **tete** *avec une autre clause*
  - *Si* **pr2** *ne réussit pas, on peut annuler l'unification de* **pr1** *et tenter de démontrer* **pr1** *ou* **tete** *avec d'autres clauses*
  - *si* **pr1** *et* **pr2** *réussissent, on franchit la coupure (qui réussit toujours) et les unifications de* **pr1** *et* **pr2** *ne peuvent être modifiées (par contre si* **pr4** *ne réussit pas, on peut revenir sur* **pr3***)*

Exemple sans coupure

```
mymember2(X,[X|_]).

mymember2(Y,[_|L]) :- mymember2(Y,L).
```



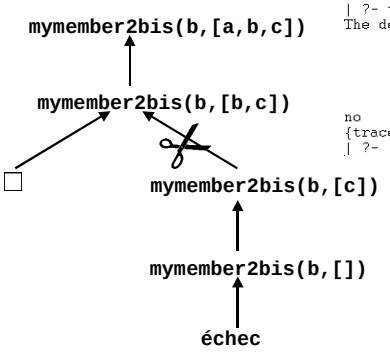

```
| ?- mymember2(b,[a,b,c]),fail.
      1    1  Call: mymember2(b,[a,b,c]) ?
      2    2  Call: mymember2(b,[b,c]) ?
      2    2  Exit: mymember2(b,[b,c]) ?
      1    1  Exit: mymember2(b,[a,b,c]) ?
      1    1  Redo: mymember2(b,[a,b,c]) ?
      2    2  Redo: mymember2(b,[b,c]) ?
      3    3  Call: mymember2(b,[c]) ?
      4    4  Call: mymember2(b,[]) ?
      4    4  Fail: mymember2(b,[]) ?
      3    3  Fail: mymember2(b,[c]) ?
      2    2  Fail: mymember2(b,[b,c]) ?
      1    1  Fail: mymember2(b,[a,b,c]) ?

no
{trace}
| ?-
```

Exemple avec coupure

```
mymember2bis(X,[X|_]) :- !.

mymember2bis(Y,[_|L]) :- mymember2bis(Y,L).
```



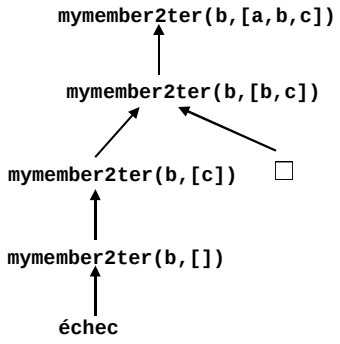

```
| ?- trace,mymember2bis(b,[a,b,c]),fail.
The debugger will first creep -- showing everything
      1    1  Call: mymember2bis(b,[a,b,c]) ?
      2    2  Call: mymember2bis(b,[b,c]) ?
      2    2  Exit: mymember2bis(b,[b,c]) ?
      1    1  Exit: mymember2bis(b,[a,b,c]) ?

no
{trace}
| ?-
```


Autre exemple avec coupure

```
mymember2ter(Y,[_|L]) :- mymember2ter(Y,L).

mymember2ter(X,[X|_]) :- !.
```




```
| ?- mymember2ter(b,[a,b,c]).
      1    1  Call: mymember2ter(b,[a,b,c]) ?
      2    2  Call: mymember2ter(b,[b,c]) ?
      3    3  Call: mymember2ter(b,[c]) ?
      4    4  Call: mymember2ter(b,[]) ?
      4    4  Fail: mymember2ter(b,[]) ?
      3    3  Fail: mymember2ter(b,[c]) ?
      2    2  Exit: mymember2ter(b,[b,c]) ?
      1    1  Exit: mymember2ter(b,[a,b,c]) ?

true ?

yes
{trace}
| ?-
```

Utilité de la coupure (1/2)

- Utile pour arrêter la recherche à la première solution :
  - *Exemple : recherche d'un élément dans une liste : on s'arrête dès qu'on a trouvé une occurence*
- Utile pour éviter des tests d'exclusion mutuelle :
  - *Exemple* :

```
humain(X) :- homme(X), !.

humain(X) :- femme(X), !.
```

- Utile pour assurer la terminaison de certains programmes :
  - *Exemple : trouver la racine entière d'un nombre à l'aide d'un générateur de nombres entiers*

```
entier(0).

entier(N) :- entier(N1), N is N1 +1.

racine(N,R) :- entier(K), K*K > N, R is K-1, !.
```


Utilité de la coupure (2/2)

- **Green cut** : la sémantique déclarative du programme **n'est pas** modifiée
  - on peut enlever le cut le programme fonctionnera toujours
  - il s'agit simplement d'optimiser la résolution
  - *Exemple* :

```
humain(X) :- homme(X), !.

humain(X) :- femme(X), !.
```

- **Red cut** : la sémantique déclarative du programme **est** modifiée
  - Le retrait du cut conduit à un programme au fonctionnement erroné
  - Souvent délicat à manipuler
  - *Exemple* :

```
min(X,Y,Z) :- X < Y, !, Z =X.

min(_,Y,Y).
```

Si alors sinon

- La coupure peut servir à simuler un **si alors sinon**

- *Exemples* :
  - *Déclaration des valeurs d'une fonction selon des intervalles*

```
f(X,0) :- X < 3,!.

f(X,2) :- X < 6,!.

f(_,4).
```

  - *Un "if then else" générique*

```
ite(P,Q,_) :- P,!,Q.

ite(_,_,R) :- R.
```

`ite(true,write('1'),write('2')).` *donne* `1 YES`
`ite(fail,write('1'),write('2')).` *donne* `2 YES`

La négation (1/3)

- On peut redéfinir la négation des opérateurs à l'aide de la coupure

- Négation de l'unification : `\=`

```
nonunif(X,Y) :- X = Y,!,fail.

nonunif(_,_).
```

- Négation de l'égalité : `\==`

```
nonegal(X,Y) :- X == Y,!,fail.

nonegal(_,_).
```

La négation (2/3)

- Négation générale d'un prédicat : **`not(+callable_term)`**

```
not(P) :- P,!,fail.

not(_).
```

- NB : en GNU Prolog, le **`not`** s'écrit **`\+`**
- Il s'agit d'une négation par l'échec : **`not(P)`** réussit si **`P`** échoue
- **`not(X)`** ne veut pas dire ***"X est toujours faux"*** mais veut simplement dire ***"X n'est pas démontrable avec les informations données"***
- **Prolog** considère ce qui n'est pas prouvé vrai comme faux et vice-versa => théorie du **monde clos**

La négation (3/3)

- *Exemple :*
  - *soit le programma* **`p(a).`**
  - *on demande* **`?- X=b, \+(p(X)).`**
    - *réponse* ***`YES {X=b}`***
  - *on demande* **`?- \+(p(X)), X=b.`**
    - *réponse* **`NO`**

```
\+(P) :- P,!,fail.

\+(_).
```

- Cette incohérence peut être évitée si l'on n'utilise la négation que sur des prédicats dont les arguments sont **déterminés**.
- De toutes manières, l'utilisation de la négation ne détermine jamais la valeur d'une variable!

Repeat

- **`Repeat`** sert à boucler sur des prédicats, généralement des prédicats de lecture sur un flux

```
lecture :- repeat,
           read(X),
           write('lu : '),
           write(X),
           nl,
           X = fin,
           !.
```