

# PROLOG 2

## Prolog - Avancé

- <http://www.labri.fr/perso/moot/SE/>
- <http://www.labri.fr/perso/retore/SE/>

# Prolog

- ① Prolog est un langage de Programmation en Logique
- ① On donne les définitions des relations (prédicats) qui nous intéressent et grâce à ces définitions Prolog répond à nos questions.
- ① A cause de l'algorithme que Prolog utilise, des façons logiquement équivalents pour définir une relation ne donnent pas toujours le même programme

# Rappel: Listes

④ Liste

④ []

④ [1 | []]

④ [1 | [2 | []]]

④ [1 | [2 | [3 | []]]]

④ [1 | [2 | [3 | [4 | []]]]]

④ Version simple

④ []

④ [1]

④ [1,2]

④ [1,2,3]

④ [1,2,3,4]

# Rappel: Listes

```
% = append(Liste1, Liste2, Liste3)
```

```
%
```

```
% vrai si Liste3 contient les éléments de Liste1
```

```
% suivi par les éléments de Liste2, c'est-à-dire
```

```
% Liste3 est la concatenation de Liste1 et %
```

```
Liste2
```

```
append([], Ys, Ys).
```

```
append([X|Xs], Ys, [X|Zs]) :-
```

```
    append(Xs, Ys, Zs).
```

# Rappel: Listes

```
% = reverse(Liste1, Liste2)
```

```
%
```

```
% vrai si Liste1 et Liste2 contiennent les mêmes  
% éléments mais dans l'ordre inverse.
```

```
reverse([], []).
```

```
reverse([X|Xs], Rs) :-
```

```
    reverse(Xs, Rs0),
```

```
    append(Rs0, [X], Rs).
```

# Rappel: Listes

`reverse([], []).`

`reverse([X|Xs], Rs) :-  
 reverse(Xs, [X], Rs).`

`reverse([], Rs, Rs).`

`reverse([X|Xs], Ys, Rs) :-  
 reverse(Xs, [X|Ys], Rs).`

# Rappel: Listes

?- reverse([a,b,c,d], Reverse).

reverse([], []).

→ reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

X = a

Xs = [b,c,d]

Reverse = Rs

# Rappel: Listes

?- reverse([a,b,c,d], Reverse).

reverse([], []).

→ reverse([a|[b,c,d]], Reverse) :-  
reverse([b,c,d], [a], Reverse).

X = a  
Xs = [b,c,d]  
Reverse = Rs

reverse([], Rs, Rs).

reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

# Rappel: Listes

?- reverse([b,c,d], [a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

# Rappel: Listes

?- reverse([b,c,d], [a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

X = b

Xs = [c,d]

Ys = [a]

Rs = Reverse

# Rappel: Listes

?- reverse([b,c,d], [a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-

reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

→ reverse([b|[c,d]], [a], Reverse) :-  
reverse([c,d], [b|[a]], Reverse).

X = b

Xs = [c,d]

Ys = [a]

Rs = Reverse

# Rappel: Listes

?- reverse([b,c,d], [a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-

reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([b,c,d], [a], Reverse) :-  
reverse([c,d], [b,a], Reverse).

X = b

Xs = [c,d]

Ys = [a]

Rs = Reverse

# Rappel: Listes

?- reverse([c,d], [b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

# Rappel: Listes

?- reverse([c,d], [b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

X = c

Xs = [d]

Ys = [b,a]

Rs = Reverse

# Rappel: Listes

?- reverse([c,d], [b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-

reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([c,d], [b,a], Reverse) :-  
reverse([d], [c,b,a], Reverse).

X = c

Xs = [d]

Ys = [b,a]

Rs = Reverse

# Rappel: Listes

?- reverse([d], [c,b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

reverse([], Rs, Rs).

➔ reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

X = d

Xs = []

Ys = [c,b,a]

Rs = Reverse

# Rappel: Listes

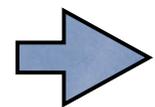
?- reverse([d], [c,b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-

reverse(Xs, [X], Rs).

reverse([], Rs, Rs).



reverse([d], [c,b,a], Reverse) :-

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

X = d

Xs = []

Ys = [c,b,a]

Rs = Reverse

# Rappel: Listes

?- reverse([], [d,c,b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

→ reverse([], Rs, Rs).  
reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

Rs = [d,c,b,a]  
Rs = Reverse

# Rappel: Listes

?- reverse([], [d,c,b,a], Reverse).

reverse([], []).

reverse([X|Xs], Rs) :-  
reverse(Xs, [X], Rs).

➔ reverse([], Rs, Rs).  
reverse([X|Xs], Ys, Rs) :-  
reverse(Xs, [X|Ys], Rs).

Reverse =  
[d,c,b,a]

# Rappel: Listes

% = sous\_liste(Partie, Totale)

%

% vrai si Totale contient Partie

C'est à dire :

sous\_liste([a,b,c,d], [a,b,c,d,e,f,g]).

sous\_liste([b,c,d], [a,b,c,d,e,f,g]).

sous\_liste([f], [a,b,c,d,e,f,g]).

sous\_liste([], [a,b,c,d,e,f,g]).

sont tous vrais.

# Rappel: Listes

% = sous\_liste(Partie, Totale)

%

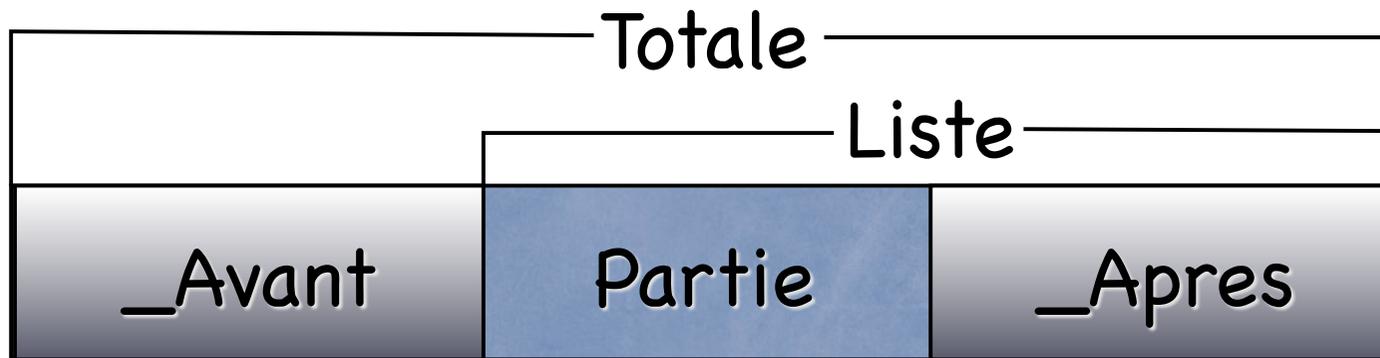
% vrai si Totale contient Partie

sous\_liste(Partie, Totale) :-

append(Partie, \_Après, Liste),

append(\_Avant, Liste, Totale).

# Rappel: Listes



```
% = sous_liste(Partie, Totale)
```

```
%
```

```
% vrai si Totale contient Partie
```

```
sous_liste(Partie, Totale) :-
```

```
    append(Partie, _Après, Liste),
```

```
    append(_Avant, Liste, Totale).
```

# Prolog

- ④ On a déjà vu des concessions à l'efficacité, par exemple le calcul des expressions arithmétique avec "is" qui ne sont pas purement logique.
- ④ Aujourd'hui on va voir des autres constructions qui ont qu'un sens procédural : si on ne fait pas attention, elles peuvent détruire le sens logique des prédicats dont elles sont utilisés.

# Prolog

## Constructions non-logiques

④ Les constructions non-logiques qu'on a déjà vu sont:

— X is Expression

— == et \==

# Prolog

## Constructions non-logiques

④ Les constructions non-logiques qu'on a déjà vu sont:

— `X is Expression`

— `==` et `\==`

`X is X0 + 1`

“is” a un sens pour Prolog que quand on connaît la valeur de `X0`

# Prolog

## Constructions non-logiques

④ Les constructions non-logiques qu'on a déjà vu sont:

— `X is Expression`

— `==` et `\==`

Contrairement à l'unification `X == Y` échoue quand `X` et `Y` sont deux variables qui peuvent dénoter des objets différents.

# Prolog

## Constructions non-logiques

- ④ Les constructions non-logiques se divisent en plusieurs catégories:
  - entrée/sortie (interaction avec fichiers et l'utilisateur)
  - contrôle (de la recherche des preuves)
  - introspection (vérification de types)
  - meta-programmation (changements de base de données, etc.)

# Prolog

## Entrée/Sortie

- ④ Les entrées et sorties se font grâce à quelque prédicats (comme “read” et “write”) qui sont faits pour.
- ④ Les sens logique des ses prédicats est “true” (vrai) mais on s’intéresse plutôt aux effets de borne (side effects) qui sont généralement l’affichage ou la lecture des termes.

# Prolog

## Entrée/Sortie

- ④ `read(Terme)` - Terme est une variable qui sera unifié avec un Terme lu du clavier; pour entrer ce Terme, l'utilisateur doit terminer par "." et Terme doit respecter la syntaxe des termes en Prolog
- ④ `write(Terme)` - écrit Terme vers le terminal.
- ④ `writeln(Terme)` - comme `write(Terme)`, mais finit sur une nouvelle ligne.

# Prolog

## Entrée/Sortie

- ④ `nl`, saut de ligne
- ④ `writeln(Terme)`, équivalent à “`write(Terme),nl`”
- ④ `tab(Int)`, imprime `Int` espaces.

# Prolog

## Entrée/Sortie

% = write\_liste(Liste)  
% écrit Liste vers le terminal, avec  
% chaque element sur une ligne.

```
write_liste([]).  
write_liste([X|Xs]) :-  
    writeln(X),  
    write_liste(Xs).
```

# Prolog

## Entrée/Sortie

- ④ l'outil le plus puissant pour imprimer des termes est `format(Atome, Liste)` qui joue un rôle similaire à `printf` en C.
- ④ `Atome` est un atome en Prolog, qui peut contenir plusieurs expressions du forme  $\sim x$  qui on une interprétation qui dépend de  $x$ .
- ④ `Liste` contient un nombre de termes qui dépend du nombre d'expressions  $\sim x$

# Prolog

## Entrée/Sortie

Exemple:

```
format('~w', [Terme])           % ~w = write
format('~w~n', [Terme])        % ~n = nl
format(' - ~w~n' [Terme])
```

```
    = write(' - '),
      write(Terme),
      nl
```

# Prolog

## Entrée/Sortie

```
% = write_liste(Liste)  
% écrit Liste vers le terminal, avec  
% chaque element sur une ligne.
```

```
write_liste([]).  
write_liste([X|Xs]) :-  
    format(' - ~w~n', [X]),  
    write_liste(Xs).
```

# Prolog

## Entrée/Sortie

- ④ pour l'interaction avec l'utilisateur vous avez le droit d'utiliser un petit bibliothèque qui nous transformera les symboles entrées par l'utilisateur en liste de termes.
- ④ ce bibliothèque traite les espaces et les symboles ponctuation comme séparateurs de mot

# Prolog

## Entrée/Sortie

Exemple

```
read_line(Liste)
```

|: Jean aime Marie, mais ce n'est pas depuis tres  
longtemps.

```
Liste = [jean, aime, marie, mais, ce, n, est, pas,  
depuis, tres, longtemps]
```

# Prolog

## Entrée/Sortie

🕒 ceci nous permet de trouver des motifs dans le symboles entrées par l'utilisateur

🕒 par exemple:

- membre(bonjour, Liste)
- sous\_liste([mal,de,tete], Liste)

# Prolog

## Entrée/Sortie

Alors, les interactions peuvent être des choses comme :

start :-

```
writeln('Bienvenue !'),  
writeln('Parlez-moi de votre problème').  
read_line(Mots),  
boucle(Mots).
```

# Prolog

## Entrée/Sortie

```
boucle(Mots) :-  
    reponse(Mots),  
    read_line(PlusDeMots),  
    boucle(PlusDeMots).
```

```
reponse(Mots) :-  
    membre(bonjour, Mots),  
    writeln('Bonjour. Content de vous revoir').
```

# Prolog

## Entrée/Sortie

```
reponse(Mots) :-  
    membre(bonjour, Mots),  
    writeln('Bonjour. Bien content de vous revoir').  
reponse(Mots) :-  
    sous_liste([mal,de,tete], Mots),  
    writeln('Prenez un aspirine').  
reponse(Mots) :-  
    sous_liste([mon,X], Mots),  
    format('Parlez-moi plus de votre ~w~n', [X]).
```

# Prolog

## Entrée/Sortie

- ④ Ce stratégie de réponse à base de la recherche de motifs ("pattern matching") a été utilisé par Eliza, un genre de psychiatre virtuel.
- ④ Prolog permet une analyse syntaxique plus profonde, mais pour notre but (et pour votre projet) les règles vues suffisent.

# Prolog

## Contrôle

- ④ Prolog permet plusieurs constructions qui ont pour but de changer la façon dont Prolog cherche des preuves.
- ④ Le plus important est la coupure “!” (en Anglais cut)
- ④ Beaucoup d'autres constructions (comme le “if...then...else”, la négation et “once”) se définissent grâce à la coupure.

# Prolog

## Contrôle

- ⦿ Attention : essayer d'éviter la coupure le plus possible !
- ⦿ Un prédicat qui contient une coupure n'a pas d'interprétation logique (au moins pas nécessairement, contrairement aux clauses en Prolog "pure") : c'est-à-dire que le sens de la coupure est procédural

# Prolog

## Contrôle

- 🕒 Il y a plusieurs raisons pour utiliser une coupure. Le plus important est l'efficacité : la coupure peut aider Prolog à éviter de faire des calculs inutiles.
- 🕒 Mais, cherchez toujours d'abord une solution sans coupure.
- 🕒 Et écrivez clairement dans le commentaire ce que la coupure est censée de faire.

# Prolog

## Contrôle

### 🌀 Le sens de la coupure “!”

- réussit toujours (comme “true”)
- toutes les solutions alternatives pour le prédicat (les clauses du même prédicat après celle-ci) sont éliminées.
- toutes les solutions alternatives pour des prédicats qui apparaissent dans la même clause avant la coupure sont éliminés.

# Prolog

## Entrée/Sortie

```
boucle([au,revoir]) :-  
    !,  
    fin.
```

```
boucle(Mots) :-  
    reponse(Mots),  
    !,  
    read_line(PlusDeMots),  
    boucle(PlusDeMots).
```

# Prolog

## Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-
    auteur(X, _).
```

# Prolog

## Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-
    auteur(X, _),
    !.
```

# Prolog

## Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
→ auteur(X) :-
    auteur(X, _),
    !.
```

# Prolog

## Contrôle

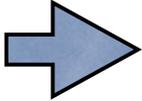
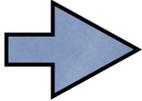
```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-
    → auteur(X, _),
    !.
```

# Prolog

## Contrôle

 % = auteur(Auteur, Livre)  
 auteur(leo\_tolstoy, anna\_karenina).  
 auteur(leo\_tolstoy, war\_and\_peace).

% = auteur(Auteur)

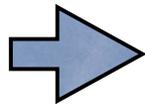
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
    !.
```

# Prolog

## Contrôle

% = auteur(Auteur, Livre)

auteur(leo\_tolstoy, anna\_karenina). 

 auteur(leo\_tolstoy, war\_and\_peace).

% = auteur(Auteur)

% vrai si Auteur est l'écrivain d'au moins un livre

auteur(X) :-

    auteur(X, \_),

    !.

# Prolog

## Contrôle

% = auteur(Auteur, Livre)

auteur(leo\_tolstoy, anna\_karenina).

➔ auteur(leo\_tolstoy, war\_and\_peace).

% = auteur(Auteur)

% vrai si Auteur est l'écrivain d'au moins un livre

auteur(X) :-

  auteur(X, \_), ➔

!.

# Prolog

## Contrôle

% = auteur(Auteur, Livre)

auteur(leo\_tolstoy, anna\_karenina).

➔ auteur(leo\_tolstoy, war\_and\_peace).

% = auteur(Auteur)

% vrai si Auteur est l'écrivain d'au moins un livre

auteur(X) :-

    auteur(X, \_),

➔ !.

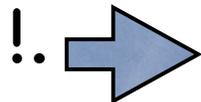
# Prolog

## Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-
    auteur(X, _),
```



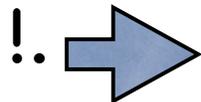
# Prolog

## Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-
    auteur(X, _),
```



# Prolog

## Contrôle

?- max(34, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

X >= Y.

max(X,Y,Y) :-

Y > X.

# Prolog

## Contrôle

?- max(34, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

    !.

max(X,Y,Y) :-

    Y > X.

# Prolog

## Contrôle

?- max(34, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

➔ max(X,Y,X) :-  
    X >= Y,  
    !.

➔ max(X,Y,Y) :-  
    Y > X.

X = 34

Y = 24

Max = X = 34

# Prolog

## Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

➔ max(34,24,34) :-  
    34 >= 24,  
    !.

➔ max(X,Y,Y) :-  
    Y > X.

X = 34  
Y = 24  
Max = X = 34

# Prolog

## Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(34,24,34) :-

→ 34 >= 24,

!.

→ max(X,Y,Y) :-

Y > X.

X = 34

Y = 24

Max = X = 34

# Prolog

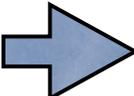
## Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(34,24,34) :-

34 >= 24, 

!.

X = 34

Y = 24

Max = X = 34

 max(X,Y,Y) :-

Y > X.

# Prolog

## Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(34,24,34) :-

34 >= 24,

→ !.

→ max(X,Y,Y) :-

Y > X.

X = 34

Y = 24

Max = X = 34

# Prolog

## Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(34,24,34) :-

34 >= 24,

!.

max(X,Y,Y) :-

Y > X.

X = 34

Y = 24

Max = X = 34

# Prolog

## Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

➔ max(X,Y,X) :-  
    X >= Y,  
    !.

max(X,Y,Y) :-  
    Y > X.

X = 0  
Y = 24  
Max = X = 0

# Prolog

## Contrôle

?- max(0, 24, 0).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

➔ max(0,24,0) :-  
    0 >= 24,  
    !.

➔ max(X,Y,Y) :-  
    Y > X.

X = 0  
Y = 24  
Max = X = 0

# Prolog

## Contrôle

?- max(0, 24, 0).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(0,24,0) :-

→ 0 >= 24,

!.

→ max(X,Y,Y) :-

Y > X.

X = 0

Y = 24

Max = X = 0

# Prolog

## Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

    !.

X = 0

Y = 24

Max = Y = 24

➔ max(X,Y,Y) :-

    Y > X.

# Prolog

## Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

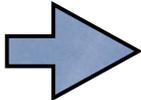
X >= Y,

!.

X = 0

Y = 24

Max = Y = 24

 max(0,24,24) :-  
24 > 0.

# Prolog

## Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

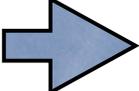
    !.

X = 0

Y = 24

Max = Y = 24

max(0,24,24) :-

 24 > 0.

# Prolog

## Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

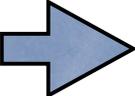
    !.

X = 0

Y = 24

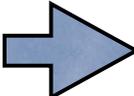
Max = Y = 24

max(0,24,24) :-

    24 > 0. 

# Prolog

## Contrôle

?- max(0, 24, 24). 

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-  
    X >= Y,  
    !.

max(X,Y,Y) :-  
    Y > X.

X = 0  
Y = 24  
Max = Y = 24

# Prolog

## Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

    !.

max(X,Y,Y) :-

    Y > X.

Si le test dans le premier clause échoue on peut être sûr que le test dans le deuxième clause réussit.

# Prolog

## Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

    !.

max(X,Y,Y).

Alors, on peut être tenté de supprimer le deuxième test.

Quel est le problème avec ce programme ?

# Prolog

## Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

➔ max(X,Y,X) :-  
    X >= Y,  
    !.

➔ max(X,Y,Y).

Echec d'unification, car on ne peut pas unifier X à la fois avec 34 et avec 24.

# Prolog

## Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,X) :-

    X >= Y,

    !.

➔ max(X,Y,Y).

Réussit, car il n'y a plus de vérification que  $Y > X$ .

# Prolog

## Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,Z) :-

    X >= Y,

    !,

    Z = X.

max(X,Y,Y).

Version correcte :

unification explicite après la  
coupure.

# Prolog

## Contrôle

% = membre(Element, Liste)

%

% vrai si Liste contient Element.

% s'utilise pour chercher un Element, mais aussi

% pour enumerer les differents Elements

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

```
membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

# Prolog

## Contrôle

```
% = verifie_member(Element, Liste)
%
% vrai si Element est strictement egal a un
% des membres de Liste.
```

```
verifie_membre(X, [Y|_]) :-
    X == Y,
    !.
verifie_membre(X, [_|Ys]) :-
    verifie_membre(X, Ys).
```

# Prolog

## Contrôle

- Alors, quelle est la différence entre
- `membre(a, [a,b,c])`. (“member” est prédéfini et marche pareil)
  - `verifie_membre(a, [a,b,c])`. (“memberchk” est prédéfini et marche pareil)

# Prolog

## Contrôle

?- membre(a, [a,b,c]).

➔ membre(X, [X|\_]).

X = a  
\_ = [b,c]

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

# Prolog

## Contrôle

?- membre(a, [a,b,c]).

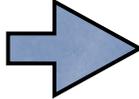
membre(X, [X|\_]). →

X = a  
\_ = [b,c]

→ membre(X, [\_|Ys]) :-  
membre(X, Ys).

Réussite directe,  
mais ...

# Prolog Contrôle

?- membre(a, [a,b,c]). 

membre(X, [X|\_]).

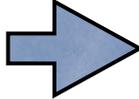
X = a  
\_ = [b,c]

 membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog a encore le  
choix !

# Prolog

## Contrôle

?- membre(a, [a,b,c]). 

membre(X, [X|\_]).

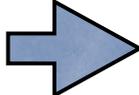
X = a  
\_ = [b,c]

 membre(X, [\_|Ys]) :-  
membre(X, Ys).

C'est-à-dire, si on  
demande une autre  
solution ...

# Prolog

## Contrôle

?- membre(a, [a,b,c]). 

membre(X, [X|\_]).

X = a  
\_ = [b,c]

 membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [a,b,c]).

membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a

\_ = a

Ys = [b,c]

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [a,b,c]).

membre(X, [X|\_]).

➔ membre(a, [\_|[b,c]]) :-  
membre(a, [b,c]).

X = a

\_ = a

Ys = [b,c]

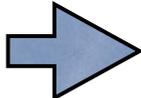
Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [a,b,c]).

membre(X, [X|\_]).

membre(a, [\_|[b,c]]) :-  
 membre(a, [b,c]).

X = a

\_ = a

Ys = [b,c]

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [b,c]).

➔ membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a

X = b

échec!

Prolog va essayer  
d'en trouver.

# Prolog Contrôle

?- membre(a, [b,c]).

membre(X, [X|\_]).

X = a  
Ys = [c]

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog va essayer  
d'en trouver.

# Prolog Contrôle

?- membre(a, [b,c]).

membre(X, [X|\_]).

X = a  
Ys = [c]

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [c]).

➔ membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a

X = c

échec!

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, [c]).

membre(X, [X|\_]).

X = a  
Ys = []

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, []).

membre(X, [X|\_]).

X = a  
Ys = []

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, []).

➔ membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a  
[] \= [X|\_]  
échec!

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, []).

membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a  
[\_|Ys] \= []  
échec

Prolog va essayer  
d'en trouver.

# Prolog

## Contrôle

?- membre(a, []).

membre(X, [X|\_]).

➔ membre(X, [\_|Ys]) :-  
membre(X, Ys).

X = a  
[\_|Ys] \= []  
échec

Prolog va essayer  
d'en trouver.

Et ça demande  
des efforts!

# Prolog

## Contrôle

?- membre(a, [a,a,a]).

membre(X, [X|\_]).

membre(X, [\_|Ys]) :-  
    membre(X, Ys).

Deuxième problème  
potentiel: qu'est-ce  
qui ce passe avec  
la question en  
haut ?

# Prolog

## Contrôle

?- membre(a, [a,a,a]).

membre(X, [X|\_]).

membre(X, [\_|Ys]) :-  
    membre(X, Ys).

Deuxième problème  
potentiel: qu'est-ce  
qui se passe avec  
la question en  
haut ?

Prolog dit "oui",  
comme il faut, mais il  
le fait trois fois

# Prolog

## Contrôle

?- verifie\_membre(a, [a,b,c]).

➔ verifie\_membre(X, [Y|\_]) :-  
    X == Y,  
    !.

X = a  
Y = a  
\_ = [b,c]

➔ verifie\_membre(X, [\_|Ys]) :-  
    verifie\_membre(X, Ys).

# Prolog

## Contrôle

?- verifie\_membre(a, [a,b,c]).

verifie\_membre(X, [Y|\_]) :-  
    → X == Y,  
      !.

X = a  
Y = a  
\_ = [b,c]

→ verifie\_membre(X, [\_|Ys]) :-  
    verifie\_membre(X, Ys).

# Prolog

## Contrôle

?- verifie\_membre(a, [a,b,c]).

verifie\_membre(X, [Y|\_]) :-

X == Y,

➔ !.

X = a  
Y = a  
\_ = [b,c]

➔ verifie\_membre(X, [\_|Ys]) :-  
verifie\_membre(X, Ys).

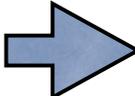
# Prolog

## Contrôle

?- verifie\_membre(a, [a,b,c]).

verifie\_membre(X, [Y|\_]) :-

X == Y,

!. 

X = a

Y = a

\_ = [b,c]

verifie\_membre(X, [\_|Ys]) :-

verifie\_membre(X, Ys).

# Prolog

## Contrôle

- ④ une coupure vert est une coupure qui élimine des démonstrations, mais n'élimine pas de solutions : c'est à dire qu'il aide Prolog à éviter des démonstrations qui ne peuvent pas réussir.
- ④ une coupure rouge est un coupure qui élimine des démonstrations mais aussi des solutions.

# Prolog

## Contrôle - négation

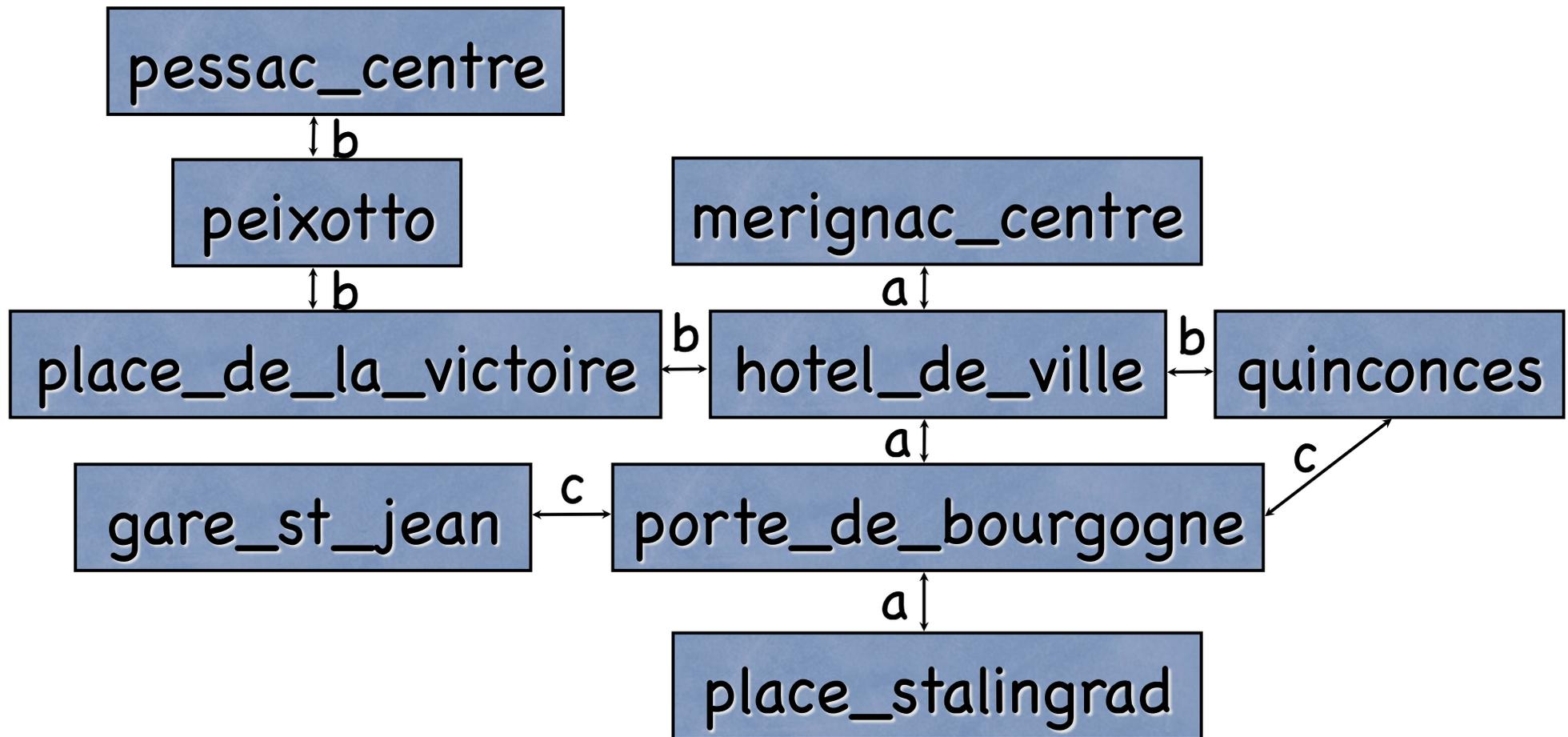
- ④ On a vu que les clauses de Horn sont une restriction sur des formules en forme normale disjonctive tel que il y a exactement un littéral positif par clause.
- ④ La négation en Prolog est un remède partiel qui permet d'avoir plusieurs littéraux positifs dans un clause.

# Prolog

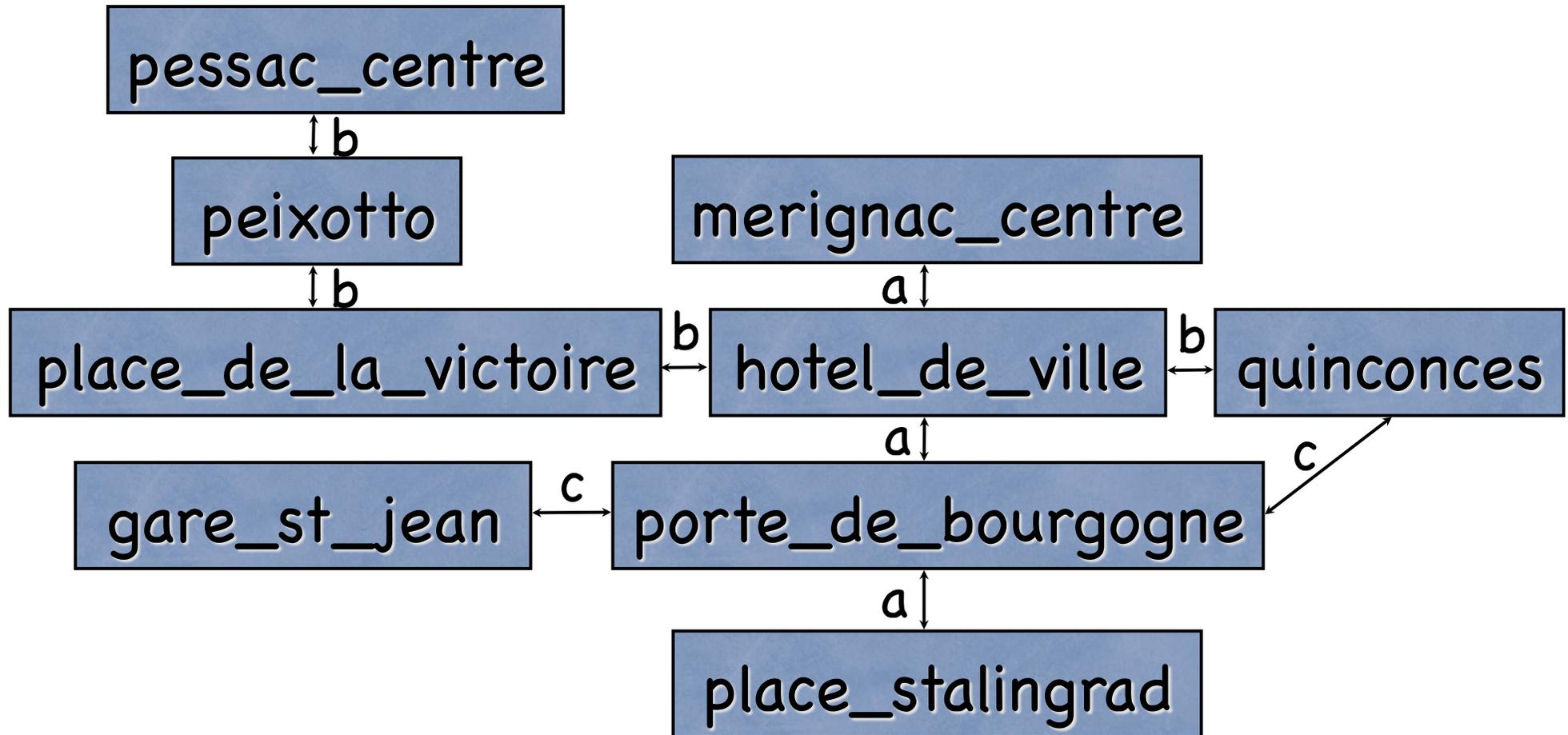
## Contrôle - négation

- ④ La négation présente dans Prolog s'appelle "négation as failure" : un littéral est faux quand il n'y a pas de preuve démontrant qu'il est vrai.
- ④ C'est pour ça que l'inverse de "true" (vrai) ne s'appelle pas "false" (faux) mais "fail" (échec).

# Tramway



# Tramway



connexion(place\_stalingrad, porte\_de\_bourgogne, a).

# Tramway

On peut prendre chaque connexion dans les deux sens

connexion\_sym(Source, Destination, Ligne) :-  
    connexion(Source, Destination, Ligne).

connexion\_sym(Source, Destination, Ligne) :-  
    connexion(Destination, Source, Ligne).

connexion(place\_stalingrad, porte\_de\_bourgogne, a).

connexion(porte\_de\_bourgogne, gare\_st\_jean, c).

...

# Tramway

```
chemin(Source, Destination) :-  
    connexion(Source, Destination, _).  
chemin(Source, Destination) :-  
    connexion(Source, Arret, _),  
    chemin(Arret, Destination).
```

# Tramway

chemin(Source, Destination) :-  
    connexion(Source, Destination, \_).

chemin(Source, Destination) :-  
    connexion(Source, Arret, \_),  
    chemin(Arret, Destination).

Quel est le problème  
avec ce programme ?

# Tramway

?- chemin(place\_stalingrad, peixotto, Chemin).

chemin(Source, Destination, Chemin) :-  
chemin(Source, Destination, [], Chemin).

chemin(Source, Destination, Chemin0, Chemin) :-  
reverse(Chemin0, Chemin).

chemin(Source, Destination, Chemin0, Chemin) :-  
connexion\_sym(Source, Arret, Ligne),  
\+ member(c(Arret,\_,\_), Chemin0),  
chemin(Arret, Destination,  
[c(Source,Arret,Ligne)|Chemin0], Chemin).

# Prolog

## Contrôle - négation

- ④ On peut définir la négation avec la coupure et “fail” (l’inverse de “true”, l’atome dont la preuve échoue directement).

# Prolog

## Contrôle - négation

```
non_membre(Element, Liste) :-  
    membre(Element, Liste),  
    !,  
    fail.  
non_membre(_Element, _Liste).
```

# Prolog

## Contrôle - négation

- ⊗ Faites attention à la négation s'il y a un terme avec des variables libres. Le but  $\neg \text{auteur}(\text{leo\_tolstoy}, \text{Livre})$  ne veut pas dire "quels sont les livres qui ne sont pas écrit par leo\_tolstoy" mais "il n'y a pas de livre dont leo\_tolstoy est l'écrivain."

# Prolog

## Contrôle - négation

```
non_auteur(Ecrivain, Livre) :-  
    auteur(Ecrivain, Livre),  
    !,  
    fail.  
non_auteur(_Ecrivain, _Livre).
```

# Prolog

## Contrôle - if then else

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

max(X,Y,Z) :-

    X >= Y,

    !,

    Z = X.

max(X,Y,Y).

# Prolog

## Contrôle - if then else

```
% = max(X, Y, Z)
```

```
max(X,Y,Z) :-
```

```
(
```

```
    X >= Y
```

```
->
```

```
    Z = X
```

```
;
```

```
    Z = Y
```

```
).
```

# Prolog

## Introspection

- Il y a plusieurs prédicats en Prolog qui permettent de déterminer le type d'un terme.
  - `var(X)`, vrai si  $X$  est une variable libre
  - `atomic(X)`, vrai si  $X$  est une terme atomaire (constante, nombre entier ou réel)
  - `compound(X)`, vrai si  $X$  est une terme complexe (du forme  $f(A,B)$ )

# Prolog

## Introspection

- ④ Un liste  $L$  est dit un liste propre si et seulement si  $L$  ne contient pas de sous-listes qui sont des variables libres.
- ④ Alors  $[], [X,a]$  et  $[X,Y,Z]$  sont des listes propres.
- ④ Mais  $X, [a | Y]$  et  $[a,b,c | Z]$  ne sont pas propres.

# Prolog

## Introspection

```
% = est_liste_propre(Terme)  
% vrai si Terme est un liste propre
```

```
est_liste_propre(Var) :-  
    var(Var),  
    !,  
    fail.
```

```
est_liste_propre([]).  
est_liste_propre([_|L]) :-  
    est_liste_propre(L).
```

# Prolog

## Introspection

- ① On a déjà vu que la syntaxe pour les termes (nos structures de données) et les prédicats (nos éléments de programmes) sont similaires.
- ① Le prédicat `call(Terme)` prend son argument Terme et l'appelle comme un prédicat.
- ① `call(Terme)` se généralise à `call(Terme, Arg)`, `call(Terme, Arg1, Arg2)` etc.

# Prolog

## Introspection

🌀 Qu'est-ce que ça veut dire ?

- `call(append([a,b],[c,d],Ys))` est équivalent à `append([a,b],[c,d],Ys)`
- `call(append, [a,b], [c,d], Ys)` est aussi équivalent à `append([a,b],[c,d],Ys)`
- `call(parent, X, Y)` est équivalent à `parent(X,Y)`

# Prolog

## Introspection

```
% = map(Liste1, Predicat, Liste2)
```

```
%
```

```
% applique Predicate(Element1, Element2)
```

```
% a chaque element de Liste1 pour obtenir un
```

```
% element pour Liste2.
```

```
map([], _, []).
```

```
map([X|Xs], P, [Y|Ys]) :-
```

```
    call(P, X, Y),
```

```
    map(Xs, P, Ys).
```

# Prolog

## Meta-programmes

- ④ `call(Terme)` nous permet de traiter un terme comme un prédicat (partie du programme) et d'essayer de faire la preuve qui en correspond.
- ④ les prédicats `assert(Terme)` et `retract(Terme)` vont plus loin en nous permettent de changer le programme.

# Prolog

## Meta-programmes

alors le prédicat `assert(parent(jean,marie))` nous ajoute le prédicat `parent(jean,marie)` à notre programme. Si `parent(jean,marie)` était faux (ou ne pas démontrable) avant, il sera vrai après l'assert.

# Prolog

## Meta-programmes

- ④ si l'ordre des clauses est important, on peut utiliser deux variants d'assert :
  - `asserta(Clause)` ajoute Clause au début des clauses pour le prédicat,
  - `assertz(Clause)` ajoute Clause à la fin.

# Prolog

## Meta-programmes

- ④ inversement, le prédicat `retractall`  
`(parent(_, _))` supprime tout les  
clauses du forme `parent(X,Y)`.
- ④ après il n'y aura aucun fait de forme `parent`  
`(X,Y)` dans la base de données.

# Prolog

## Meta-programmes

- ④ assert et retract sont utiles pour enregistrer des données de façon permanente.
- ④ alors ils peuvent servir pour avoir une base de données dynamique (où les données peuvent changer)
- ④ mais aussi pour faire la tabulation : pour enregistrer les résultats de calcul intermédiaire et ainsi d'éviter de recalculer de résultats

# Prolog

## Meta-programmes

```
toutes_solutions(Pred, Solutions) :-  
    call(Pred),  
    assertz(solution_file(Pred)),  
    fail.
```

```
toutes_solutions(_Pred, Solutions) :-  
    assertz(solution_file(fin)),  
    recuperer_solutions(Solutions).
```

# Prolog

## Meta-programmes

```
recuperer_solutions(Solutions) :-  
    retract(solution_file(Element)),  
    (  
        Element = fin  
    ->  
        Solutions = []  
    ;  
        Solutions = [Element|Sols0],  
        recuperer_solutions(Sols0)  
    ).
```

# Prolog

## Meta-programmes

- ④ les prédicats `findall`, `bagof` et `setof` (partie de SWI Prolog et beaucoup d'autres) généralisent le prédicat `toutes_solutions`

# Prolog

## Un mini Système Expert

- 🌐 tout ceci nous donne les bons outils pour programmer un système expert.
- 🌐 je vous donnerai un petit exemple très simple, mais qui sera facile à étendre.

# Prolog

## Un mini Système Expert

- ④ l'architecture est la suivante :
  - le système expert est dans un certain état,
  - suivant la réponse de l'utilisateur le système expert change d'état et pose une nouvelle question ou donne une réponse.
  - chaque état à une petite explication qui explique le pourquoi de la question ou de la réponse.