La place du "!" dans une rêgle:

```
/* max avec coupure (rouge)*/
max( X, Y, X) :- X >= Y, !.
max( X, Y ,Y).
```

Que se passe-t-il si on avance d'une place la coupure dans le corps de la règle?

```
/* version non correcte */
max( X, Y, X) :- !, X >= Y.
max( X, Y ,Y).
```

Calculons le but `max(1,2,R).`

bon usage de la coupure: Pour exploiter l'exclusion mutuelle et l'exaustivité d'un test, il faut placer la coupure *après* le test en question.

D'autres exemples d'utilisation de !

Le prédicat `inter(liste1,liste2,resultat).`
Version sans cut:

```
inter([],T,[]).
inter([C|G],T,[C|P]):- member(C,T),inter(G,T,P).
inter([C|G],T,R):- not(member(C,T)),inter(G,T,R).

 ?- inter([2,1,4],[1,3,4],X).
X = [1,4] ? ;
no
```

Version avec cut:

```
inter([],T,[]).
inter([C|G],T,[C|P]):- member(C,T), ! ,inter(G,T,P).
inter([_|G],T,R):- inter(G,T,R).
```

Calculons le but `inter([1,2],[2,3],R)`

Que se passe-t-il pour le même but, si on remplace

```
inter([C|G],T,[C|P]):-member(C,T), !, inter(G,T,P).
```

par

```
inter([C|G],T,[C|P]):- !, member(C,T), inter(G,T,P).
```

?

Encore un exemple

Le prédicat `ects_sem(sem,projet,ects)` du TD3:

version sans cut:

```
ects_sem(X,[],0).
ects_sem(X,[Y|G],N):-ue(Y,M,X),ects_sem(X,G,K),N is M+K.
ects_sem(X,[Y|G],N):-ue(Y,M,Z),X=\=Z,ects_sem(X,G,N).
```

```
 ?- ects_sem(1,[an1,ia,ig], X).
X = 9 ? ;
no
```

(rappel: `ue(an1,3,1).  ue(ia,6,1).  ue(ig,6,2).`)

Version avec cut:

```
ects_sem(X,[],0).
ects_sem(X,[Y|G],N):-ue(Y,M,X), !, ects_sem(X,G,K), N is M+K.
ects_sem(X,[Y|G],N):-ue(Y,M,Z), ects_sem(X,G,N).
```

bon usage de la coupure: Pour exploiter l'exclusion mutuelle et l'exaustivité d'un test, il faut placer la coupure *après* le test en question.

Ici le test est implicite:

```
ects_sem(X,[Y|G],N):-ue(Y,M,X), !, ects_sem(X,G,K), N is M+K.
```

abrège

```
ects_sem(X,[Y|G],N):-ue(Y,M,Z), Z=X, !, ects_sem(X,G,K),
                                  N is M+K.
```

Le cut peut apparaître en première position dans le corps d'une clause:

```
f(0,1).
f(N,R):- N>0, P is N-1, f(P,Q), R is N*Q.
```

version avec cut:

```
f(0,1):-!.
f(N,R):- P is N-1, f(P,Q), R is N*Q.
```

Dans cet exemple, le test mutuellement exclusif a lieu directement dans la tête de la rêgle. Version équivalente:

```
f(N,1):- N=0,!.
f(N,R):- P is N-1, f(P,Q), R is N*Q.
```

L'algorithme d'unification

```
Entrée:  deux termes t1 et t2.
Sortie:  la substitution σ, mgu de t1 et t2, si elle existe,
sinon echec.
Empiler t1,t2.
Tant que pile non vide
   dépiler u,v
   Selon le cas:
   1) Si u est une variable sans occurrence dans v:
  On substitue v à u dans la pile et dans σ
  2) Si v est une variable sans occurrence dans u:
  On substitue u à v dans la pile et dans σ
  3) Si u et v sont des constantes ou variables identiques:
  On continue.
  4) Si u = f(u_1,...,u_n) v = f(v_1,...,v_n).
  On empile u_i = v_i pour i de n à 1.
  Sinon échec.
```

Exemple

```
f(g(X,A,h(X,b)),Z) = f(g(A,b,Z),Y)
```

au tableau

Spécifier un prédicat: arguments d'entrée et de sortie

Un argument est *d'entrée* pour un prédicat si, au moment du `call` du prédicat, l'argument en question doit être clos, sans variables.
Un argument est *de sortie* pour un prédicat si, au moment du `exit` du prédicat, l'argument en question doit contenir le résultat de l'appel.
Par exemple dans la définition:

```
fact(0,1).
fact(N,R):- N>0, M is N-1, fact(M,P), R is P*N.
```

le premier argument et d'entrée et le deuxiéme de sortie.
Une specification correcte de `fact` est

```
fact(+Arg, -Res)
```

Spécifier un prédicat: arguments d'entrée et de sortie (2)

Les erreurs comme

```
[INSTANTIATION ERROR- in arithmetic:  expected bound value ]
```

viennent de l'utilisation de termes ouverts pour des argument d'entrée.

Par exemple:

```
?  - X is Y + 4
```

ou

```
?  - Z < 7
```

Spécifier un prédicat: arguments d'entrée et de sortie (3)

Dans certains cas il n'y a pas d'arguments d'entrée et de sortie. Un exemple:

```
concat([],L,L).
concat([X|L],G,[X|H]):-concat(L,G,H).
```

Plusieurs possibilités d'utilisation:

```
[eclipse 1]: concat([1,2],[3],L).
L = [1, 2, 3]
Yes (0.00s cpu)
```

....mais aussi:

```
[eclipse 2]:  concat([1,2],L,[1,2,3]).
L = [3]
Yes (0.00s cpu)
```

et:

```
[eclipse 3]:  concat(L,[3],[1,2,3]).
L = [1, 2]
Yes
```

ou encore:

```
[eclipse 5]:  concat(L,G,[1,2,3]).
L = []
G = [1, 2, 3]
Yes (0.00s cpu, solution 1, maybe more) ?  ;
L = [1]
G = [2, 3]
Yes (0.00s cpu, solution 2, maybe more) ?  ;
L = [1, 2]
G = [3]
Yes (0.00s cpu, solution 3, maybe more) ?  ;
L = [1, 2, 3]
G = []
Yes (0.00s cpu, solution 4)
```

Une spécification correcte dans ce cas est `concat(?Liste1,?Liste2,?Liste3)`

Algorithmes "générer et tester"

Le "backtracking" (retour en arrière) de Prolog est très bien adapté aux problèmes qui nécessitent l'exploration de plusieurs configurations possibles à la recherche d'une (ou de plusieurs) solutions (coloriage d'un plan, *n* reines, carrés magiques, sudoku....)

Deux exemples dans ce cours (d'autres en TP):

Le problème des *n* reines.

La reconnaissance d'un mot par un automate fini non déterministe.

Les n reines

Problème: placer n reines sur un echiquier $n \times n$, sans qu'elles se menacent mutuellement (chaque ligne, colonne et diagonale doit contenire au plus une reine).
Réprésentation des configurations: une liste de n entiers, inclus entre 1 et n.
Le i-ème entier donne la position de la reine sur la i-ème ligne.
Par exemple, pour $n = 4$, la configuration:

		R	
			R
R			
	R		

est réprésentée par
`[3,4,1,2]`

Les n reines

Autant de configurations que de permutations des entiers de 1 à n.
Remarque: une représentation du problème bien choisie est essentielle pour éviter un trop grand nombre de configurations.

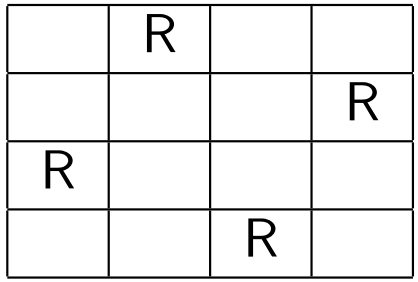
Si une configuration était réprésentée, par exemple, par le choix de n case de l'echiquier, on aurait $C_{n^2}^n$ configuration (au lieu de $n!$).

Pour $n = 8$ cela donne 4426165368 configurations au lieu de 40320 (on gagne un facteur 10^5 par un choix judicieux des configurations).

Les *n* reines

Par construction, chaque ligne et chaque colonne de l'échiquier contient exactement une reine.
Les solutions sont les configuration telles que chaque diagonale et chaque anti-diagonale contient au plus une reine.

Par exemple:



réprésentée par `[2,4,1,3]`

Les n reines

Deux reines en positions (i, j) et (i', j') se trouvent sur la même diagonale ssi $i - j = i' - j'$.

0	-1	-2	-3
1	0	-1	-2
2	1	0	-1
3	2	1	0

Les n reines

Deux reines en positions (i, j) et (i', j') se trouvent sur la même anti-diagonale ssi $i + j = i' + j'$.

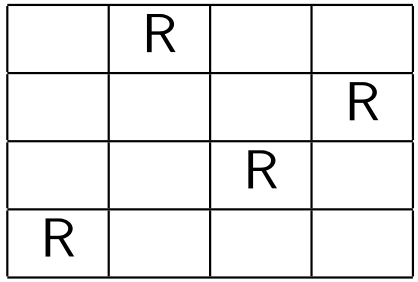
2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

Les n reines

Pour resoudre n reines on peut donc:

- générer une permutation P de $1, ..., n$.
- calculer les listes D de différences et la liste S de sommes associés à P. Le i-ème élément de D est $i - P(i)$, le i-ème élément de S est $i + P(i)$.
- vérifier que tous les éléments de D sont différents entre eux. Même chose pour S. Si ces tests réussissent, on a une solution.
- Passer à la permutation suivante et recommencer.

Exemple: on applique l'algorithme du transparent précédent à la permutation `p= [2,4,3,1]`, qui représente la configuration (qui n'est pas un solution):



La liste de différences de `p` est `[1-2,2-4,3-3,4-1]=[-1,-2,0,3]` Donc le test sur les diagonales passe (les reines se trouvent sur des diagonales toutes différentes les unes des autres).
La liste des sommes de `p` est `[1+2,2+4,3+3,4+1]=[3,6,6,5]`. La deuxième et troisième reine se trouvent sur la même anti-diagonale. On élimine cette configuration et on passe à la suivante.

Les *n* reines

Voici un prédicat pour générer les permutations, défini par récurrence sur n:

```
perm([],[]).

perm([X|L],Z):-perm(L,W),insertion(X,W,Z).

insertion(X,L,[X|L]).

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

Exemple d'utilisation de `perm` avec $n = 3$:

```
   ?- perm([1,2,3],X).
X = [1,2,3] ? ;
X = [2,1,3] ? ;
X = [2,3,1] ? ;
X = [1,3,2] ? ;
X = [3,1,2] ? ;
X = [3,2,1] ? ;
no
```

Les *n* reines

Voici un prédicat pour générer les listes des sommes et des différences: (le 1er argument contient les indices de ligne, le 2ème ceux de colonne, le 3ème les sommes et le 4ème les differences).

```
combiner([],[],[],[]).

combiner([X1|X],[Y1|Y],[S1|S],[D1|D]) :-
     S1 is X1 +Y1,

     D1 is X1 - Y1,

     combiner(X,Y,S,D).
```

Exemple d'utilisation de `combiner` pour l'exemple précedent:

```
 ?- combiner([1,2,3,4],[2,4,3,1],S,D).
D = [-1,-2,0,3],
S = [3,6,6,5] ? ;
no
```

Les *n* reines

Voici un prédicat pour vérifier que tous les éléments d'une liste sont différents:

```
tous_diff([X]).

tous_diff([X|Y]) :-  not(member(X,Y)), tous_diff(Y).
```

Les n reines

Voici un prédicat qui resout n reines, dans le cas $n = 4$:

```
resout(P) :-
        perm([1,2,3,4],P),
        combiner([1,2,3,4],P,S,D),
        tous_diff(S),
        tous_diff(D).
```

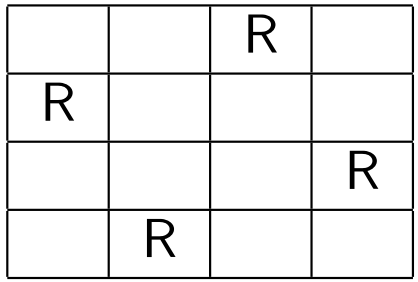
Les *n* reines

On calcule les solutions:

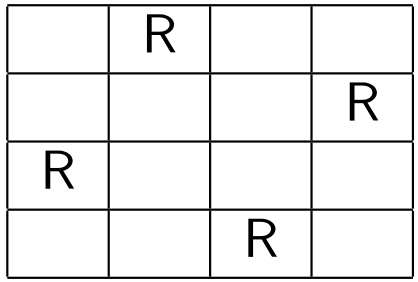
```
?- resout(X).

X = [3,1,4,2] ? ;

X = [2,4,1,3] ? ;

no
```

Les n reines

Les deux solutions pour $n = 4$ sont: `[3,1,4,2]`



et `[2,4,1,3]`

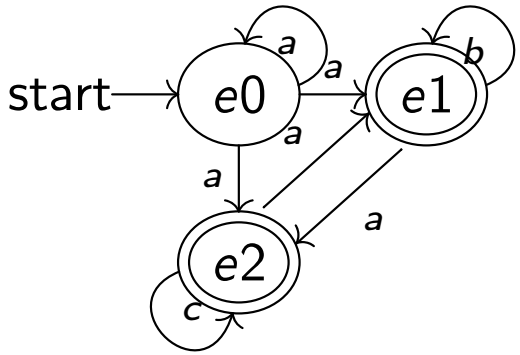


Les *n* reines

Pour $n = 8$, (après modification de `resout`):

```
 ?- resout(X).

X = [5,2,6,1,7,4,8,3] ? ;

X = [6,3,5,7,1,4,2,8] ? ;

X = [6,4,7,1,3,5,2,8] ? ;

X = [3,6,2,7,5,1,8,4] ?

...

?- setof(X,resout(X),L),length(L,N).

L = [[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],
[1,7,4,6,8,2,5,3],....]

N = 92 ?
```

AFND



est réprésenté par:

```
init(e0).

final(e1). final(e2).

delta(e0,a,e0). delta(e0,a,e1). delta(e0,a,e2).

delta(e1,b,e1). delta(e1,a,e2).

delta(e2,a,e1). delta(e2,c,e2).
```

AFND

Pour vérifier si un mot est accepté:

- générer un chemin de lecture du mot à partir de l'état initial.
- vérifier que le chemin temine sur un état final.
- Sinon, recommencer.

AFND

```
parse(L) :- init(S),        /* L liste de caracteres:
            accepte(S,L). /* le mot a lire

accepte(X,[]) :-    /* fin de la lecture
      final(X).    /* mot accepté si l'etat X est finale

accepte(X,[A|B]) :-
      delta(X,A,Y),   /* lecture de la lettre suivante
      accepte(Y,B).
```

Dans l'exemple, la lecture de [a] termine avec succes à la 2ème tentative.
(au tableau)

Prédicats de test

- `integer/1` teste si l'argument est un entier.

```
?- integer(3).
Yes.
?- integer(X).
No.
```

- `number/1` teste si l'argument est un nombre.
- `atomic/1` teste si l'argument est une constante.
- `var/1` teste si l'argument est une variable non instanciée.
- `nonvar/1` teste si l'argument n'est pas une variable non instanciée.
- `compound/1` teste si l'argument est un terme composé.
- `ground/1` teste si l'argument est un terme sans variables.

Exemple d'utilisation

```
/*Addition*/

plus(X, Y, Z) :-
    ground(X),
    ground(Y),
    Z is X + Y.

plus(X, Y, Z) :-
    ground(Y),
    ground(Z),
    X is Z - Y.

plus(X, Y, Z) :-
    ground(X),
    ground(Z),
    Y is Z - X.
```

Prédicats de manipulation de termes

- `functor(Terme, Foncteur, Arite).`
  Mode d'utilisation (+, -, -) ou (-, +, +) seulement.

```
?- functor(pere(jean, isa), F, A).
F = pere, A = 2.

?- functor(T, pere, 2).
T = pere(X, Y).
```

- `arg(N, Terme, X)` unifie `X` et le `N`-ième argument de Terme.

```
?- arg(1, pere(jean, isa), X).
X = jean.
```

Prédicats de manipulation de termes

`Terme =.. Liste` transforme un terme en une liste.

```
?- pere(jean, isa) =.. L.
L = [pere, jean, isa].

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

Exemple d'utilisation

```
/*Map*/
map_carre([],[]).
map_carre([X|L],[Y|T]):-Y is X*X, map_carre(L,T).

?-  map_carre([2,4,3],L).
L = [4,16,9] ?

map([],_,[]).
map([X|L],T,[Y|R]):-Q=..[T,X,Y],Q,map(L,T,R).

carre(X,Y):- Y is X*X.

?-  map([2,4,3],carre,L).
L = [4,16,9] ?

double(X,Y):-Y is X+X.

?-  map([2,4,3],double,L).
L = [4,8,6] ?
```

Exemple d'utilisation

```
/*Filter*/

filtre_negatifs([],[]).
filtre_negatifs([N|L],[N|P]):-N>=0,filtre_negatifs(L,P).
filtre_negatifs([N|L],P):-N<0,filtre_negatifs(L,P).

?- filtre_negatifs([1,-3,7,-2],L).
L = [1,7] ?

filtre([],_,[]).
filtre([X|L],G,[X|R]):- T=..[G,X], T, filtre(L,G,R).
filtre([X|L],G,R):- T=..[G,X], not(T), filtre(L,G,R).

positif(X):-X>0.

?-  filtre([1,-3,7,-2],positif,L).
L = [1,7] ?

pair(X):- (X mod 2) =:= 0.
```

L'unification re-programmée en Prolog

```
/* Unification*/

/* Unifier deux Variables*/
unify(X, Y) :-
     var(X),var(Y),X = Y.


/* Premier argument variable*/
unify(X, Y) :-
     var(X),nonvar(Y),X = Y.


/* Deuxieme argument variable*/
unify(X, Y) :-
     nonvar(X),var(Y),Y = X.
```

```
/* Les arguments sont des constantes atomiques*/
unify(X, Y) :-
     nonvar(X), nonvar(Y),
     atomic(X), atomic(Y),
     X = Y.


/* Les arguments sont des termes  composes.*/
unify(X, Y) :-
     nonvar(X), nonvar(Y),
     compound(X), compound(Y),
     termUnify(X, Y).
```

```
/* Unifier deux termes composes*/
termUnify(X, Y) :-
     functor(X, F, N),
     functor(Y, F, N), /* meme symb. fonctionnelle*/
     argUnify(N, X, Y).


/* Unifier les arguments */
argUnify(N, X, Y) :-
     N>0,
     argUnify1(N, X, Y),
     Ns is N - 1,
     argUnify(Ns, X, Y).


argUnify(0, X, Y).
```

```
/* Unifier les arguments N */
argUnify1(N, X, Y) :-
     arg(N, X, ArgX),
     arg(N, Y, ArgY),
     unify(ArgX, ArgY).
```