PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Visiter l'arbres de dérivation: le prédicat `trace`

On peut visualiser les traces des calculs en utilisant le prédicat prédéfini `trace/0` (`notrace/0` pour quitter le traceur).
Tracer un but donne lieu à une séquence de messages, qui nous informent sur l'arbre de recherche que PROLOG est en train de construire (utile pour debug). Ces messages sont de 4 types:

- `call p(t1,...,tn)` : le but `p(t1,...,tn)` est en tête de la pile des buts. La recherche d'une clause dont la tête s'unifie avec `p(t1,...,tn)` commence.
- `exit p(t1,...,tn)`: le but `p(t1,...,tn)` a été prouvé.
- `redo p(t1,...,tn)`: on revient au but `p(t1,...,tn)` par *backtracking*.
- `fail p(t1,...,tn)`: il n'existe plus aucune possibilité d'unifier le but `p(t1,...,tn)` avec la tête d'une clause.

Considérons par exemple le programme `ex.pl`:

```
q(a).
q(b).
r(b).
r(c).
p(X):-q(X),r(X).
```

et le but:

```
?- P(X).
```

```
 ?- consult(exemple).
yes
   ?- trace.
[ Trace mode on. ]
yes
[trace]
   ?- p(X).
   (1) call:p(_172) ?
   (2) call:q(_172) ?
   (2) exit:q(a) ?
   (3) call:r(a) ?
   (3) fail:r(a) ?
   (2) redo:q(a) ?
   (2) redo:q(_172) ?
   (2) exit:q(b) ?
   (4) call:r(b) ?
   (4) exit:r(b) ?
   (1) exit:p(b) ?
```

```
X = b ? ;
   (1) redo:p(b) ?
   (4) redo:r(b) ?
   (4) fail:r(b) ?
   (2) redo:q(b) ?
   (2) redo:q(_172) ?
   (2) fail:q(_172) ?
   (1) redo:p(_172) ?
   (1) fail:p(_172) ?
no
[trace]
   ?- notrace.
   (1) call:notrace ?
[ Debug mode off. ]
yes
```

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Clauses et buts disjonctifs (1)

La disjonction est une abbreviation en PROLOG. La clause:

`tete :- p_1 ; p_2 ; .... ; p_n .` est équivalente à la séquence de clauses:

`tete :- p_1 . tete :- p_2 . ... tete :- p_n .`

Donc, l'ordre des clauses d'une disjonction est important. Par exemple:

```
boucle:-boucle.
test1:- boucle;true.
test2:- true;boucle.
```

Les prédicats `test1` et `test2` ont la même sémantique déclarative (commutativité de la disjonction) mais le but `test2.` termine avec succes et le but `test1.` provoque une boucle.

Clauses et buts disjonctifs (2)

Analoguement, le but disjonctif

`boucle;true`

termine avec succes, mais

`true;boucle`

diverge.

Une utilisation typique de la disjonction (qui utilise le prédicat `bio` du 1er cours):

```
parent(X,Y):- bio(Y,_,_,_,X,_);bio(Y,_,_,_,_,X).
```

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Empêcher le retour en arrière

- Le retour en arrière est “cablé” en Prolog.
- Il peut causer de l’inefficacité.
- On veut le contrôler.

Exemple: le "if then else"

Le prédicat `max(+Arg1,+Arg2,-Res)`:

```
max(X,Y,X):-X>=Y.
max(X,Y,Y):-X<Y.
```

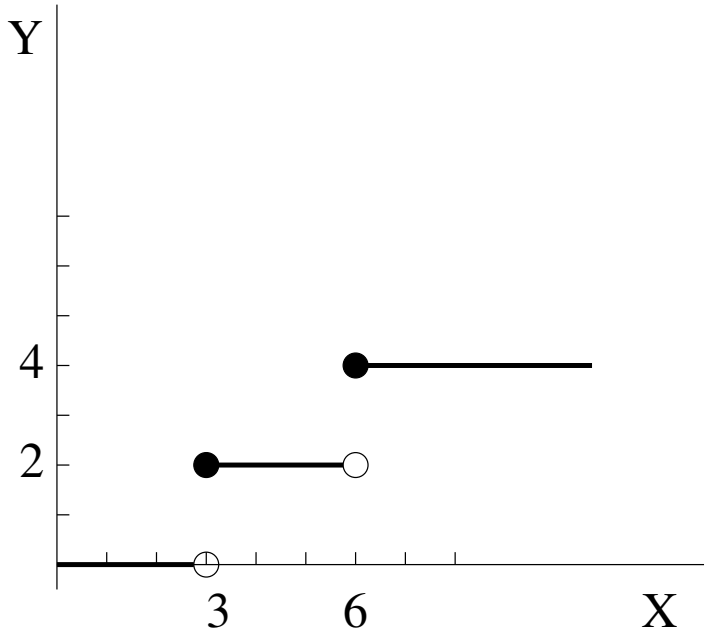
Calculons le but `max(2,1,R)`.

C'est inefficace. Le programme C qui implemente le même algorithme serait:

```
int
max (int n,int m)
{
    if (m>=n) return m;
    if (m<n) return n;
}
```

Comment programmer un "if-then-else" en PROLOG?

Exemple 2: définitions par cas



La relation entre X et Y peut être définie par trois règles:

- si $X < 3$ alors $Y = 0$
- si $X \geq 3$ et $X < 6$ alors $Y = 2$
- si $X \geq 6$ alors $Y = 4$

Exemple 2: définitions par cas

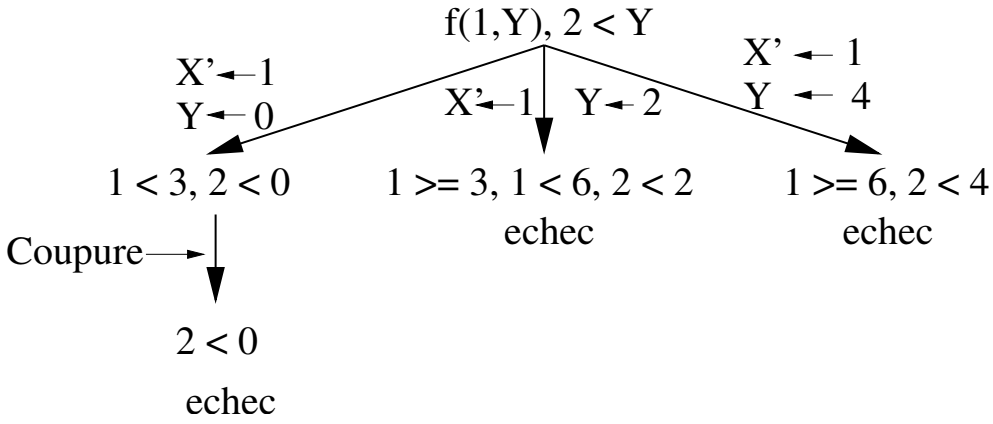
En Prolog on définit le prédicat `f(+Arg, Res)`:

```
f( X, 0) :-  X < 3.          % règle 1

f( X, 2) :-  X >= 3, X < 6.  % règle 2

f( X, 4) :-  X >= 6.         % règle 3
```

Exemple 2: un premier but.

```
?- f( 1, Y), 2 < Y.
```

- Pendant l'exécution du premier but `f( 1, Y)`, `Y` est instanciée par 0. Le deuxième but devient `2 < 0`. C'est un échec.
- Avant de répondre `no`, Prolog fait des retours en arrière successifs.



- Les trois règles pour f sont mutuellement exclusives.
- Il faudrait éviter les retours en arrière.

Le prédicat `!`

- Dans les exemples `max/3` et `f/2`, il faudrait exprimer le fait que, si un choix de valeurs des variables du problème satisfait une certaine condition, alors on veut que ce choix soit définitif, c.à.d. qu'on ne veut pas en essayer d'autres.
- Dans le cas de `max`, la condition est `X>=Y`
- Dans le cas de `f`, la condition est `X<3`, pour la première règle, et `X>=3,X<6` pour la deuxième.
- Cet effet s'obtient en plaçant une *coupure*, c.à.d. une occurrence du prédicat pre-défini `!`, après la condition en question.
- Du point de vue de la sémantique déclarative, `!` équivaut à `true`.

Exemple 2: deuxième version du programme

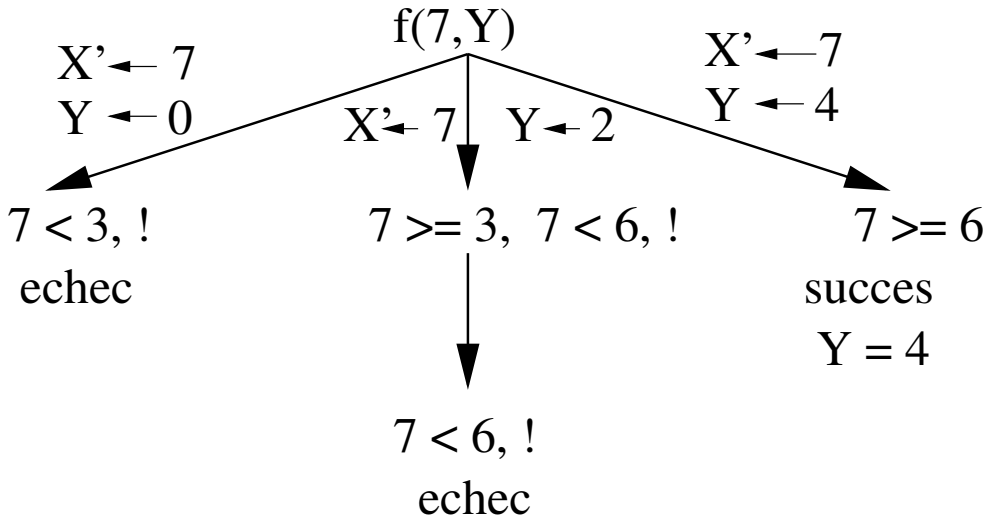
```
f( X, 0) :-  X < 3, !.

f( X, 2) :-  X >= 3, X < 6, !.

f( X, 4) :-  X >= 6.
```

- Prolog répond à la question `?- f( 1, Y), 2 < Y.` plus rapidement.
- Quand on insère des coupures dans un programme, il faut que la sémantique opérationnelle reste inchangée (il s'agit d'une optimisation) dans le cas ci dessus, la sémantique déclarative ne change pas non plus (dans ce cas, les coupures introduites sont des *green cut*).
- La coupure peut aussi changer la sémantique déclarative (*red cut*).

Exemple 2: une deuxième but

```
?- f( 7, Y).
```

```
Y = 4
```



- On teste `X < 3` et après `X >= 3`.
- Même chose pour `X < 6` et `X >= 6`.
- C'est inefficace.

Troisième version du programme

- On peut reformuler les règles:
  - Si $X < 3$ alors $Y = 0$,
  - sinon si $X < 6$ alors $Y = 2$,
  - sinon $Y = 4$.
- En Prolog:

```
f( X, 0) :-  X < 3, !.
f( X, 2) :-  X < 6, !.
f( X, 4).
```

Exemple 2: suite et fin

- Ce programme produit les mêmes résultats que l'autre mais plus efficacement.
- Si on enlève les coupures:

```
f( X, 0) :-  X < 3.
f( X, 2) :-  X < 6.
f( X, 4).
```

  on change les résultats !

- Par exemple,

```
?- f( 1, Y).

Y = 0;

Y = 2;

Y = 4;

no
```

La coupure - Définition

- Nous appelons **but parent** le but qui s'unifie avec la tête de la clause contenant la coupure.
- Si la coupure est atteinte, elle "réussit" (du point de vue logique, `!` est équivalent à `true`).
- Dès qu'un retour en arriere remonte à la coupure, le but parent echoue (au niveau de la trace du calcul, au lieu de faire "redo !" on remonte dans l'arbre et on fait "fail P", où P est le but parent de la coupure).
- Autrement dit, tous les choix entre le "call" du but parent et le "call" de la coupure sont définitifs.
- Tous les eventuels choix alternatifs ne seront plus considérées.

Exemple

```
p(X) :- q(X), !, r(X).
p(X) :- u(X).
p(c).
q(X) :- s(X).
q(d).
r(a).
r(b).
r(d).
s(a).
s(b).
u(d).
```

```
?- p(X).
   X = a ;
No
?- r(X),!,q(Y).
   X = a   Y = a;
   X = a   Y = b;
   X = a   Y = d;
No
```

Exemples

```
% Maximum sans coupure
maximum( X, Y, X) :- X >= Y.
maximum( X, Y ,Y) :- X < Y.

% Maximum avec coupure (verte)
maximum2( X, Y, X) :- X >= Y, !.
maximum2( X, Y ,Y) :- X < Y.

% Autre version avec coupure (rouge)
maximum3( X, Y, X) :- X >= Y, !.
maximum3( X, Y ,Y).
```

Exemples (Listes)

```
% Membre à une solution
membreune( X, [ X | _]) :- !.
membreune( X, [ _ | L]) :- membreune(X, L).

% Ajouter un élément sans duplication
% ajouter(+X,+L,-L1)
ajouter( X, L, L) :- membre( X, L), !.
ajouter( X, L, [X | L]).
```

Schema général d'utilisation de `!`:

`P:-C_1,Q_1.`		`P:-C_1,!,Q_1.`	Si les `C_j` sont
`...`	$\Rightarrow$	`...`	mutuellement exclusives,
`P:-C_n-1,Q_n-1.`		`P:-C_n-1,!,Q_n-1.`	les deux programmes sont
`P:-C_n,Q_n.`		`P:-C_n,Q_n.`	équivalents (green cuts)

`P:-C_1,Q_1.`		`P:-C'_1,!,Q_1.`	`C'_j` est une "simplification
`...`	$\Rightarrow$	`...`	de `C_j`, qui prend en compt
`P:-C_n-1,Q_n-1.`		`P:-C'_n-1,!,Q_n-1.`	l'echec de `C_1,...,C_j-1`.
`P:-C_n,Q_n.`		`P:-Q_n.`	En général, les programmes
			ne sont pas équivalents (red

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Le `if then else`

Avec la coupure, on peut définir le `if then else`. Il s'agit d'une macro prédéfinie en Yap PROLOG. En voici la syntaxe et la définition:

```
(P -> Q; R) :- P, !, Q.
(P -> Q; R) :- R.
```

Exemple d'utilisation:

```
max(X,Y,Z):- ( X>= Y -> Z=X;Z=Y).
```

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

La négation

- Prolog permet la négation avec le prédicat `not/1` (ou `\+`)
- `not(B)` exprime le fait que `B` ne peut pas être prouvé.
- Si un but `B` réussit alors `not(B)` échoue.
- Si un but `B` échoue alors `not(B)` réussi.
- Ce n'est pas la négation logique.
- `not(B)` est défini par:

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

- Attention: nier des prédicats qui contiennent des variables peut donner des résultats inattendus.
  Par exemple: `not(member(X, [a, b]))` echoue.

La négation (suite)

Il faut que l'appel d'un but avec négation soit fait *après* l'instanciation des variables; par exemple:

```
marie(francois).

etudiant(rene).

etudiant_celibataire(X) :-
        not(marié(X)),
        etudiant(X).

?- etudiant_celibataire(X).

no


?- etudiant_celibataire(rene).

yes
```

Version correcte

```
etudiant_celibataire(X) :-
    etudiant(X),
    not(marie(X)).

?- etudiant_celibataire(X).
X = rene ;
No
```

Exemple d'utilisation de la négation

- Un arbre (binaire sans étiquettes) est soit la constante `f` (une feuille), soit un terme de la forme `n(t1,t2)`, où `t1` et `t2` sont deux arbres.
- Un *parcours* est une liste de `g` et `d` (pour "gauche" et "droite"). Définir un prédicat `p(+arbre,-parcours)` qui engendre, pour un arbre donné, tous les parcours qui mènent de la racine aux feuilles (qu'on appelle *parcours complets*). Par exemple:

```
  ?-  parcours(n(n(f,f),f),X).
X = [g,g] ? ;
X = [g,d] ? ;
X = [d] ? ;
no
```

- Un arbre est *équilibré* si la difference entre les longueurs de deux parcours complets quelquonques dans l'arbre est inferieure ou égale à 1. Définir un prédicat `equilibre(+arbre)` qui réussit si et seulement si `arbre` est équilibré.

```
parcours(f,[]).
parcours(n(G,D),[g|X]):- parcours(G,X).
parcours(n(G,D),[d|X]):- parcours(D,X).

longueur([],0).
longueur([_|R],Re):-longueur(R,P), Re is P+1.

desequilibre(T):-parcours(T,X), parcours(T,Y), longueur(X,H1),
    longueur(Y,H2), D is abs(H1-H2) , D>1.

equilibre(T):-not(desequilibre(T)).
```

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Prédicats d'Entrée/Sortie

- Prolog communique avec des flux.
- Il y a un flux d'entrée (clavier) et de sortie standard (écran).

Les prédicats de base:

- `see(fichier)` change le flux d'entrée vers fichier.
- `see(user)` met le flux d'entrée standard.
- `tell(fichier)` change le flux de sortie vers fichier.
- `tell(user)` met le flux de sortie standard.
- `seen` ferme le fichier d'entrée courant.
- `told` ferme le fichier de sortie courant.
- `read(X)` lit le prochain terme (delimité par `.`) du flux d'entré et unifie avec X.
- `write(X,Y).` écrit le terme sur le flux de sortie.

Exemple 1

```
cube :- write('Nouveau chiffre, s.v.p. (ou stop): '),
        read(X),
        process(X).

process(stop).

process(N) :-C is N * N * N, write('Le cube de '),
             write(N),
             write(' est: '),
             write(C), nl, cube.
```

Exemple 2

```
/* etoiles( [3,4]) imprime  */
/*  ***  */
/*  ****  */
etoiles([]).

etoiles([N | L]) :- etoilesligne(N), nl,
                    etoiles( L).

etoilesligne(0).

etoilesligne(N) :-  N > 0,
                     write(*),
                     N1 is N - 1,
                     etoilesligne(N1).
```

Exemple 3

```
/* les fichiers source et destination doivent exister avant */
/* le call de copy. L'effet est de copier les entiers > 3 de */
/* source dans destination                                    */

copy:- see(source), tell(destination),
       boucle,
       seen, told.

boucle:-read(X),
        (X=end_of_file -> true;
        (X>3-> (write(X),boucle);boucle)).
```

PLAN

- Visiter l'arbre de dérivation: le prédicat `trace`.
- Extensions du langage:
  - Disjonction.
  - Coupure.
  - If then else.
  - Négation.
- Prédicats d'entrée/sortie.
- Prédicats du second ordre.

Prédicats du second ordre

- `bagof(X, P, L)` produit la liste `L` de tous les termes `t` tel que le but P`[t/X]` est satisfait.
- `setof(X, P, L)` comme `bagof` sauf que la liste `L` est ordonné et les éléments en double sont éliminés.

```
age(marie,5).
age(anne,5).
age(paul,7).
age(marc,10).
```

```
?- bagof(Enfant,age(Enfant,5),Liste).

Enfant = _G348

Liste = [marie,anne] ;

no
```

```
?- bagof(Enfant,age(Enfant,Age),Liste).
Enfant = _G360
Age = 5
Liste = [marie,anne] ;
Enfant = _G360
Age = 7
Liste = [paul] ;
Enfant = _G360
Age = 10
Liste = [marc] ;
no
```

```
?- setof(Enfant,age(Enfant,Age),Liste).
Enfant = _G360
Age = 5
Liste = [anne, marie] ;
Enfant = _G360
Age = 7
Liste = [paul] ;
Enfant = _G360
Age = 10
Liste = [marc] ;
no
```

L'ordre sur les termes

- The predicateare `@</2` is used to compare and order termsg:
- variables come before numbers, numbers come before atoms which in turn come before compound terms, i.e.: variables @¡ numbers @¡ atoms @¡ compound terms.
- variables are roughly ordered by "age" (the "oldest" variable is put first);
- floating point numbers are sorted in increasing order;
- Integers are sorted in increasing order;
- atoms are sorted in lexicographic order;
- compound terms are ordered first by name, then by arity of the main functor, and finally by their arguments in left-to-right order.