# Cours3DéclarativitéEtContrôle

January 22, 2024

## 1 Cours 3 : Déclarativité et Contrôle

Séance 4 et 5

Fin du prolog pur, début de la convergence entre différents paradigmes de programmation (CP, RO, CAS), avec la logique comme pilier central général.

1) Réversibilité (quand on peut, si on peut : ce n'est pas un objectif en soi ; mais parfois, c'est une partie de la solution, cela ouvre d'autres perspectives de résolution, à explorer !)
   - Prédicat naturellement symétrique
   - Cas de la concaténation (8 cas à observer)
   - Réversibilité de prédicats courants ?
2) Contraintes (réversibilité arithmétique ?)
   - cas du dif, cas du is
   - cas des contraintes numériques (linéaires et/ou pas, entière vs réelle vs complexe)
   - cas des contraintes structurelles (=>graphes !)
   - cas des contraintes ensemblistes
3) Non-déterminisme
   - Solutions redondantes
   - Solutions non-redondantes
   - Prédicats d'énumération classiques (entiers naturels, entiers relatifs, couple, puissance, produit cartésien, liste, partie, permutation)
   - Enumération + Contraintes => Recherche combinatoire
4) Contrôle de l'exécution : Coupure, ordre supérieur, gel, assert
   - Coupure (mécanisme de base, not, coupure verte vs coupure rouge)

## 2 Réversibilité

Un programme sera dit réversible si l'exécution prévue (Données ==» Résultats) peut être inversé (Résultats «== Données), c'est à dire si le profil informel du prédicat est stricte (certaines variables ont un statut de données en entrée strict, ou pas)

La réversibilité est établi en assurant la terminaison des exécutions pour des profils d'appels différents. En première approche, la correction et la complétude ne sont pas touchées.

La réversibilité n'est pas un objectif en soi, mais permet des usages plus diversifiés, réduit le nombre de programmes nécessaires (un seul programme pour la concaténation et la séparation, par ex.), ouvre des perspectives de recherche d'algorithmes nouveaux (en chaînage arrière, plutôt qu'en chaînage avant seul ; ou avec des chaînages mixtes), permet des utilisations avec des données partiellement connues (données imparfaites ou données avec parties manquantes)

## 2.1 Prédicats symétriques

Certains prédicats semblent symétriques, p(I,O) = p(O,I), pour autant les programmes sont-ils reversibles, est-il possible de les rendre réversibles ? * symétrique (ok)

[1]:
```
%%writefile prog.pl
/* symétrique(A,B) vrai ssi A et B sont 2 arbres binaires symétriques l'un de␣
 ↪l'autre */
symetrique([],[]).
symetrique([Rac,FilsGaucheA,FilsDroitA],[Rac,FilsGaucheB,FilsDroitB]):-
  symetrique(FilsGaucheA,FilsDroitB),
  symetrique(FilsDroitA,FilsGaucheB).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[2]:
```
!echo "symetrique([1,[2,[],[]],[]],X)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [symetrique([1,[2,[],[]],[]],[1,[],[2,[],[]]])]
```

[3]:
```
!echo "symetrique(X,[1,[2,[],[]],[]])." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [symetrique([1,[],[2,[],[]]],[1,[2,[],[]],[]])]
```

- inverser premier et dernier élément d'une liste (?, cela dépend de l'écriture)

[5]:
```
%%writefile prog.pl
/* inversePermierDernier(A,B) vrai ssi A et B sont 2 avec les mêmes éléments␣
 ↪sauf les extrèmes qui sont inversés */
inversePermierDernier([],[]).
inversePermierDernier([A],[A]).
inversePermierDernier([A,B],[B,A]).
inversePermierDernier(A,B) :-
  concatene([Premier],FinA,A),
  concatene([Dernier],FinB,B),
  concatene(Centre,[Dernier],FinA),
  concatene(Centre,[Premier],FinB).

concatene([],F,F).
concatene([E|L],F,[E|R]):-concatene(L,F,R).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
```

```
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[6]:
```
!echo "inversePermierDernier([1,2,3,4,5],X)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [inversePermierDernier([1,2,3,4,5],[5,2,3,4,1])]
```

[7]:
```
!echo "inversePermierDernier(X,[1,2,3,4,5])." | swipl -g halt -s prog.pl
```

```
Votre requête ?

ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 4Kb, global: 0.9Gb, trail: 0Kb
ERROR:   Stack depth: 19,173,120, last-call: 100%, Choice points: 13
ERROR:   In:
ERROR:     [19,173,120] user:concatene('<garbage_collected>', [length:1], _416)
ERROR:     [38] user:inversePermierDernier('<garbage_collected>',
'<garbage_collected>')
ERROR:     [37] '$bags':findall_loop(<compound inversePermierDernier/2>,
'<garbage_collected>', _464, [])
ERROR:     [36] system:setup_call_catcher_cleanup(<compound (:)/2>, <compound
(:)/2>, _494, <compound (:)/2>)
ERROR:     [32] user:main
ERROR:
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to double the limit.
Warning: Goal (directive) failed: user:main
```

Pour comprendre pourquoi l'un fonctionne et pas l'autre, il faut analyser la réversibilité de la concaténation (puis la réversibilité du prédicat d'inversion premier-dernier, ensuite).

## 2.2 Concaténation

Cas de la concaténation :

[ ]:
```
%%writefile prog.pl
/* concatene(D,F,L) est vrai ssi la concaténation des listes D et F corespond à␣
 ↪la liste L. */
concatene([],F,F).
concatene([E|L],F,[E|R]):-concatene(L,F,R).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

[16]:
```
!echo "concatene([1,2],[3,4,5],L),write(\"L=\"),writeln(L)." | swipl -g halt -s␣
↪prog.pl
```

```
Votre requête ?
L=[1,2,3,4,5]
Liste des résultats :
[(concatene([1,2],[3,4,5],[1,2,3,4,5]),write(L=),writeln([1,2,3,4,5]))]
```

[17]:
```
!echo "concatene(D,[3,4,5],[1,2,3,4,5]),write(\"D=\"),writeln(D)." | swipl -g␣
↪halt -s prog.pl
```

```
Votre requête ?
D=[1,2]
Liste des résultats :
[(concatene([1,2],[3,4,5],[1,2,3,4,5]),write(D=),writeln([1,2]))]
```

[18]:
```
!echo "concatene([1,2],F,[1,2,3,4,5]),write(\"F=\"),writeln(F)." | swipl -g␣
↪halt -s prog.pl
```

```
Votre requête ?
F=[3,4,5]
Liste des résultats :
[(concatene([1,2],[3,4,5],[1,2,3,4,5]),write(F=),writeln([3,4,5]))]
```

[19]:
```
!echo␣
↪"concatene(D,F,[1,2,3,4,5]),write(\"D=\"),writeln(D),write(\"F=\"),writeln(F).
↪" | swipl -g halt -s prog.pl
```

```
Votre requête ?
D=[]
F=[1,2,3,4,5]
D=[1]
F=[2,3,4,5]
D=[1,2]
F=[3,4,5]
D=[1,2,3]
F=[4,5]
D=[1,2,3,4]
F=[5]
D=[1,2,3,4,5]
F=[]
Liste des résultats : [(concatene([],[1,2,3,4,5],[1,2,3,4,5]),write(D=),writeln(
[]),write(F=),writeln([1,2,3,4,5])),(concatene([1],[2,3,4,5],[1,2,3,4,5]),write(
D=),writeln([1]),write(F=),writeln([2,3,4,5])),(concatene([1,2],[3,4,5],[1,2,3,4
,5]),write(D=),writeln([1,2]),write(F=),writeln([3,4,5])),(concatene([1,2,3],[4,
5],[1,2,3,4,5]),write(D=),writeln([1,2,3]),write(F=),writeln([4,5])),(concatene(
[1,2,3,4],[5],[1,2,3,4,5]),write(D=),writeln([1,2,3,4]),write(F=),writeln([5])),
(concatene([1,2,3,4,5],[],[1,2,3,4,5]),write(D=),writeln([1,2,3,4,5]),write(F=),
```

```
writeln([]))]
```

[ ]:
```
!echo "concatene(D,[3,4,5],L),write(\"D=\"),writeln(D),write(\"L=\"),writeln(L).
↪" | swipl -g halt -s prog.pl
```

[22]:
```
!echo "concatene([1,2],F,L),write(\"F=\"),writeln(F),write(\"L=\"),writeln(L)."␣
↪| swipl -g halt -s prog.pl
```

```
Votre requête ?
F=_2486
L=[1,2|_2486]
Liste des résultats : [(concatene([1,2],_2588,[1,2|_2588]),write(F=),writeln(_25
88),write(L=),writeln([1,2|_2588]))]
```

### 2.3 Prédicats courants

- sur le premier élément d'une liste : premier, fin, ajoutEnPremier, supprimePremier, sépareePremierFin
- sur le dernier élément d'une liste : dernier, debut, ajoutEnFin, sépareDebutDernier
- prédicat linéaire sur une liste donnant un nombre : longueur, somme, indice, nbOccurence
- prédicat linéaire sur une liste donnant une liste : remplace, filtre, concatene, supprDoublonConsécutif
- prédicat moins élémentaire sur une liste : inversePremierDernier, inverse, supprDoublonQlcq, tri

On peut étudier maintenant et comprendre la reversibilité de inversePremierDernier.

Parmi les prédicats de la premier liste, ajoutEnPremier et supprimePremier sont inverse l'un de l'autre et peuvent être fait à partir de sépareePremierFin s'il est réversible (ce qui semble raisonnable, le prédicat ne nécessite probablement pas de récursivité, la terminaison ne doit pas être un problème).

[29]:
```
%%writefile prog.pl
/* separePremierFin(E,F,L) est vrai ssi E est le premier et F la fin de la␣
↪liste L, i.e. : [E|F] = L . */
separePremierFin(E,F,[E|F]).

/* ajoutEnPremier(E,L,R) est vrai ssi en ajoutant E en premier à L , on a la␣
↪même liste que R */
ajoutEnPremier(E,L,Res) :-
  separePremierFin(E,L,Res).

/* supprimePremier(L,R) est vrai ssi en supprimant le premier élément de L , on␣
↪a la même liste que R */
supprimePremier(L,Res) :-
  separePremierFin(_E,Res,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
```

```
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[26]:
```
!echo "ajoutEnPremier(0,[1,2,3],L),write(\"L=\"),writeln(L)." | swipl -g halt␣
 ↪-s prog.pl
```

```
Votre requête ?
L=[0,1,2,3]
Liste des résultats :
[(ajoutEnPremier(0,[1,2,3],[0,1,2,3]),write(L=),writeln([0,1,2,3]))]
```

[27]:
```
!echo "supprimePremier([1,2,3],L),write(\"L=\"),writeln(L)." | swipl -g halt -s␣
 ↪prog.pl
```

```
Votre requête ?
L=[2,3]
Liste des résultats :
[(supprimePremier([1,2,3],[2,3]),write(L=),writeln([2,3]))]
```

Pour les prédicats sur le dernier élément, est-il raisonnable de vouloir le même comportement ? Pour le premier élément, il n'y avait pas de difficulté (car pas de récursivité), mais pour le dernier élément, à priori cela sera plus difficile à cause de la récursivité et du risque de non-terminaison. Il faut donc vérifier la terminaison avec attention

[30]:
```
%%writefile prog.pl
/* separeDebutDernier(Debut,Dernier,L) est vrai ssi Debut est le début et␣
 ↪Dernier le dernier élément de la liste L */
separeDebutDernier([],Dernier,[Dernier]).
separeDebutDernier([E|Debut],Dernier,[E|Res]):-
  separeDebutDernier(Debut,Dernier,Res).

/* ajoutEnDernier(E,L,R) est vrai ssi en ajoutant E en dernier à L , on a la␣
 ↪même liste que R */
ajoutEnDernier(E,L,Res) :-
  separeDebutDernier(L,E,Res).

/* supprimeDernier(L,R) est vrai ssi en supprimant le dernier élément de L , on␣
 ↪a la même liste que R */
supprimeDernier(L,Res) :-
  separeDebutDernier(Res,_E,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[31]:
```
!echo "ajoutEnDernier(4,[1,2,3],L),write(\"L=\"),writeln(L)." | swipl -g halt␣
↪-s prog.pl
```

```
Votre requête ?
L=[1,2,3,4]
Liste des résultats :
[(ajoutEnDernier(4,[1,2,3],[1,2,3,4]),write(L=),writeln([1,2,3,4]))]
```

[32]:
```
!echo "supprimeDernier([1,2,3],L),write(\"L=\"),writeln(L)." | swipl -g halt -s␣
↪prog.pl
```

```
Votre requête ?
L=[1,2]
Liste des résultats :
[(supprimeDernier([1,2,3],[1,2]),write(L=),writeln([1,2]))]
```

Pour les autres prédicats, idem, il faut regarder finement les terminaisons et aussi, parfois, se demander ce que pourrait être l'inverse d'un prédicat. Ex. :

- l'inverse de la longueur, la génération d'une liste quelconque de longueur donnée ? (ok)
- l'inverse de la somme, en limitant aux entiers positifs non nul, l'ensemble des découpages possibles d'un entier en somme d'entiers positifs non nuls ; en ne supposant pas de limitation, … ?
- l'inverse du tri, la permutation ? (ok, mais il y a peut-être plus simple) Visiblement, certains prédicats risquent de ne pas être facilement inversibles.

La réversibilité, introduit deux questions naturelles : comment assurer l'inversion numérique (de A+B => = C, peut-on déduire à partir de A et de C la valeur de B) et comment caractériser/prendre en compte la multiplicité des solutions (ex. les permutations).

## 2.4 Contraintes

Rappel :

- en pur Prolog pas d'arithmétique (N+1 est un terme/arbre +(N,1))
- pour ajouter de l'arithmétique, pendant 20 ans, dif et is
  - dif : pour établir la différence entre termes
  - is : pour calculer des expressions

[33]:
```
%%writefile prog.pl
main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[34]:
```
!echo "dif(2,3)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [dif(2,3)]
```

[35]:
```
!echo "dif(2,2)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : []
```

[36]:
```
!echo "dif(2,X)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [dif(2,_4610)]
```

[37]:
```
!echo "dif(X,Y)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [dif(_4610,_4672)]
```

[38]:
```
!echo "dif(X,X)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : []
```

- dif est reversible, il n'y a pas de statut spécifique pour les paramètres, toutes les exécutions fonctionnent. Avec dif, on peut faire un prédicat supprimeDoublon reversible, mais attention à ne pas en vouloir trop (ajouter des doublons, cela peut se faire un nombre infini de fois, il risque d'y avoir une nombre de solution infini ; pour limiter les réponses, donner une liste réponse de taille fixe, ex. : supprimeDoublon([A,B,C,D],[1,2]). )

[39]:
```
!echo "E is 2+3." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [5 is 2+3]
```

[40]:
```
!echo "5 is E+3." | swipl -g halt -s prog.pl
```

```
Votre requête ?

ERROR: is/2: Arguments are not sufficiently instantiated
Warning: Goal (directive) failed: user:main
```

- is n'est pas réversible, son utilisation dans les prédicats introduit des comportements non réversibles.

Rem. (il n'y a pas que is en cause, c'est tout l'arithmétique et les tests, l'utilisation prévue est orientée et non-réversible) :

[41]:
```
!echo "2 > X." | swipl -g halt -s prog.pl
```

```
Votre requête ?

ERROR: >/2: Arguments are not sufficiently instantiated
Warning: Goal (directive) failed: user:main
```

Pour pallier à ce défaut de réversibilité pour l'arithmétique, de nombreuses solutions ont été explorées par la communauté prolog :

- contraintes pseudo-linéaires (simplexe)
- contraintes entières ou réelles, ou complexes
- contraintes sur ensemble fini, ou sur intervalle
- propagation de contraintes

Pour ce cours, le moteur de résolution adopté sera clpq, dit de résolution de contraintes rationnelles (pseudo-linéaire). Cela limitera, certes, mais en pratique cela sera suffisant (en particulier en Miage). La syntaxe est simple, il suffit d'ajouter des accolades, ex. : {A+B=C}

Le moteur de résolution de contrainte s'intègre au moteur d'unification, dans les arbres d'exécution (à la main), on pourra ajouter des ensembles de contraintes à respecter, et les résoudre "à la main", si nécessaire.

[42]:
```
%%writefile prog.pl
main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- use_module(library(clpq)), main.
```

```
Overwriting prog.pl
```

[43]:
```
!echo "{5=E+3}." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [{5=2+3}]
```

D'autres domaines scientifique utilisant des contraintes (appartenant à la Recherche Operationnelle ou à la Programmation par Contraintes) ont aussi été mis à contribution pour donner ainsi naissance à la Programmation Logique avec Contraintes (PLC ou CLP) :

- contraintes sur les booléens ou sur des ensembles finis (pour des représentations et des énumérations "efficaces", ou pour des représentation de problème en 0-1, avec ou sans (ensembliste))
- contraintes structurelles (sur les structures arborescentes => graphes)
  - exemple de listes circulaires

[44]:
```
!echo "L=[1,2,3|L]." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : @([S_1=S_1],[S_1=[1,2,3|S_1]])
```

- exemple de graphe

[47]:
```
!echo "L=[1,[2,L,[]],[3,[],L]]." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : @([S_1=S_1],[S_1=[1,[2,S_1,[]],[3,[],S_1]]])
```

Certains prédicats continuent de marcher sur ces objets (d'autres non, ou pas tel que) :

- premier, ajoutEnPremier, supprimePremier : ok
- dernier, etc. : ko

[49]:
```
%%writefile prog.pl
/* separePremierFin(E,F,L) est vrai ssi E est le premier et F la fin de la␣
 ↪liste L, i.e. : [E|F] = L . */
separePremierFin(E,F,[E|F]).

/* ajoutEnPremier(E,L,R) est vrai ssi en ajoutant E en premier à L , on a la␣
 ↪même liste que R */
ajoutEnPremier(E,L,Res) :-
  separePremierFin(E,L,Res).

/* supprimePremier(L,R) est vrai ssi en supprimant le premier élément de L , on␣
 ↪a la même liste que R */
supprimePremier(L,Res) :-
  separePremierFin(_E,Res,L).

/* separeDebutDernier(Debut,Dernier,L) est vrai ssi Debut est le début et␣
 ↪Dernier le dernier élément de la liste L */
separeDebutDernier([],Dernier,[Dernier]).
separeDebutDernier([E|Debut],Dernier,[E|Res]):-
  separeDebutDernier(Debut,Dernier,Res).

/* ajoutEnDernier(E,L,R) est vrai ssi en ajoutant E en dernier à L , on a la␣
 ↪même liste que R */
ajoutEnDernier(E,L,Res) :-
  separeDebutDernier(L,E,Res).

/* supprimeDernier(L,R) est vrai ssi en supprimant le dernier élément de L , on␣
 ↪a la même liste que R */
supprimeDernier(L,Res) :-
  separeDebutDernier(Res,_E,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[51]:
```
!echo "L=[1,2|L], supprimePremier(L,R)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
```

```
Liste des résultats :
@([([1|S_1]=[1|S_1],supprimePremier([1|S_1],S_1))],[S_1=[2,1|S_1]])
```

[52]:
```
!echo "L=[1,2|L], ajoutEnDernier(0,L,R)." | swipl -g halt -s prog.pl
```

```
Votre requête ?

ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 3Kb, global: 0.9Gb, trail: 0Kb
ERROR:   Stack depth: 38,346,207, last-call: 100%, Choice points: 13
ERROR:   In:
ERROR:     [38,346,207] user:separeDebutDernier('<garbage_collected>', 0, _416)
ERROR:     [37] '$bags':findall_loop(<compound (',')/2>, '<garbage_collected>',
_438, [])
ERROR:     [36] system:setup_call_catcher_cleanup(<compound (:)/2>, <compound
(:)/2>, _468, <compound (:)/2>)
ERROR:     [32] user:main
ERROR:     [31] system:catch(<compound (:)/2>, <compound error/2>, <compound
(:)/2>)
ERROR:
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to double the limit.
Warning: Goal (directive) failed: user:main
```

L'algorithmique de ces objets doit prendre en compte les boucles possibles dans les structures de données pour éviter les boucles infinies introduites par les données (c'est possible, en marquant ces objets ou en gardant la liste des objets déjà visités, cf. algorithmique sur les graphes)

## 3 Non déterminisme

Avec les prédicats réversibles ou naturellement avec certains prédicats, plusieurs solutions peuvent être trouvées par le moteur d'inférence ProLog.

[61]:
```
%%writefile prog.pl
/* membre(E,L) est vrai ssi E est dans la liste L */
membre(E,[E|_L]).
membre(E,[_F|L]):-
  membre(E,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[62]:
```
!echo "membre(E,[1,2,3]), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
E=1
E=2
E=3
Liste des résultats : [(membre(1,[1,2,3]),write(E=),writeln(1)),(membre(2,[1,2,3
]),write(E=),writeln(2)),(membre(3,[1,2,3]),write(E=),writeln(3))]
```

Ces solutions :

- peuvent être redondantes (c'est complet, mais redondant ; en travaillant sur la complétude, sans la perdre, l'objectif est alors de ne plus avoir de règles redondantes)
- peuvent être en nombre infini (alors il faut prévoir un mécanisme de coupure, pour éviter une boucle infinie, ou se restreindre à la première solution
  - parmi les solutions en nombre infini, attention à garantir que toutes les solutions arrivent si l'on attends assez longtemps.

Exemple de solution redondante (pour membre) :

[71]:
```
%%writefile prog.pl
/* membre(E,L) est vrai ssi E est dans la liste L */
membre(E,[E]).
membre(E,[E|_L]).
membre(E,[_F|L]):-
  membre(E,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[72]:
```
!echo "membre(E,[1,2,3]), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
E=1
E=2
E=3
E=3
Liste des résultats : [(membre(1,[1,2,3]),write(E=),writeln(1)),(membre(2,[1,2,3
]),write(E=),writeln(2)),(membre(3,[1,2,3]),write(E=),writeln(3)),(membre(3,[1,2
,3]),write(E=),writeln(3))]
```

attention à ne pas réduire la complétude en voulant éviter la redondance

[73]:
```
%%writefile prog.pl
/* membre(E,L) est vrai ssi E est dans la liste L */
membre(E,[E]).
membre(E,[_F|L]):-
  membre(E,L).
```

```
main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[74]:
```
!echo "membre(E,[1,2,3]), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
E=3
Liste des résultats : [(membre(3,[1,2,3]),write(E=),writeln(3))]
```

## 3.1 Enumérations classiques

Parmi les prédicats non déterministes, on trouve tous les prédicats d'énumérations d'ensembles (finis ou pas) :

- énumération d'un ensemble fini (membre, intervalle)
- énumération d'un ensemble infini (entier naturel, entier pair, entier relatif, etc.)
- énumération d'un produit cartésien (ex. : couple de booléens, trio de nombres, etc.), d'une liste d'entier
- énumération des parties d'un ensemble fini, d'une permutation

Un point commun à la plupart de ces programmes, les récursivités ne s'écrivent pas en 2 cas (1 cas de base, 1 cas de propagation ; sinon cela ne serait pas non-déterministe), mais avec au moins 3 cas (parfois cela peut être un peu caché).

Pour vérifier que ces programmes sont corrects, souvent, cela peut être une bonne idée de compter le nombre de solution (et vérifier que c'est bien ce que la combinatoire devait donner)

### 3.1.1 Exemples

**Exemple pour les entiers relatifs**, attention à panacher positifs et négatifs.
Pour comprendre comme cela marche (ou pas), dessiner des arbres d'exécution.
Rem. : Dans tous les cas, on peut prouver la correction, la complétude et la terminaison.

[78]:
```
%%writefile prog.pl
/* entierRelatif(N) est vrai ssi N est un entier relatif */
entierRelatif(0).
entierRelatif(N):-
  entierPositif(N).
entierRelatif(N):-
  entierNegatif(N).

entierPositif(1).
entierPositif(N):-
  entierPositif(M),{N=M+1}.
```

```
entierNegatif(-1).
entierNegatif(N):-
  entierNegatif(M),{N=M-1}.

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- use_module(library(clpq)), main.
```

```
Overwriting prog.pl
```

[ ]:
```
!echo "entierPositif(E), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

[ ]:
```
!echo "entierNegatif(E), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

[ ]:
```
!echo "entierRelatif(E), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

Pour éviter de n'avoir que la moitié des entiers relatifs il faut prévoir 2 solutions quand cela termine (une solution positive et une négative) :

[90]:
```
%%writefile prog.pl
/* entierRelatif(N) est vrai ssi N est un entier relatif */
entierRelatif(N):-
  entierRelatif(0,N).

entierRelatif(N,N).
entierRelatif(M,N):-
  {N = -M}.
entierRelatif(M,N):-
  {P = M+1},
  entierRelatif(P,N).


main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- use_module(library(clpq)), main.
```

```
Overwriting prog.pl
```

[ ]:
```
!echo "entierRelatif(E), write(\"E=\"), writeln(E)." | swipl -g halt -s prog.pl
```

**Exemple pour Parties**

[95]:
```
%%writefile prog.pl
/* partie(P,L) est vrai ssi P est une partie de L */
partie([],[]).
partie([E|P],[E|L]):-
```

```
  partie(P,L).
partie(P,[_E|L]):-
  partie(P,L).


main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[97]:
```
!echo "partie(P,[1,2,3]), write(\"P=\"), writeln(P)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
P=[1,2,3]
P=[1,2]
P=[1,3]
P=[1]
P=[2,3]
P=[2]
P=[3]
P=[]
Liste des résultats : [(partie([1,2,3],[1,2,3]),write(P=),writeln([1,2,3])),(par
tie([1,2],[1,2,3]),write(P=),writeln([1,2])),(partie([1,3],[1,2,3]),write(P=),wr
iteln([1,3])),(partie([1],[1,2,3]),write(P=),writeln([1])),(partie([2,3],[1,2,3]
),write(P=),writeln([2,3])),(partie([2],[1,2,3]),write(P=),writeln([2])),(partie
([3],[1,2,3]),write(P=),writeln([3])),(partie([],[1,2,3]),write(P=),writeln([]))
]
```

**Exemple pour permutation**

[98]:
```
%%writefile prog.pl
/* permutation(P,L) est vrai ssi P est une permutation de L */
permutation([],[]).
permutation([E|P],L):-
  membre(E,L,R),
  permutation(P,R).

membre(E,[E|L],L).
membre(E,[F|L],[F|R]):-
  membre(E,L,R).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[99]:
```
!echo "permutation(P,[1,2,3]), write(\"P=\"), writeln(P)." | swipl -g halt -s␣
↪prog.pl
```

```
Votre requête ?
P=[1,2,3]
P=[1,3,2]
P=[2,1,3]
P=[2,3,1]
P=[3,1,2]
P=[3,2,1]
Liste des résultats : [(permutation([1,2,3],[1,2,3]),write(P=),writeln([1,2,3]))
,(permutation([1,3,2],[1,2,3]),write(P=),writeln([1,3,2])),(permutation([2,1,3],
[1,2,3]),write(P=),writeln([2,1,3])),(permutation([2,3,1],[1,2,3]),write(P=),wri
teln([2,3,1])),(permutation([3,1,2],[1,2,3]),write(P=),writeln([3,1,2])),(permut
ation([3,2,1],[1,2,3]),write(P=),writeln([3,2,1]))]
```

# 4 Non-déterminisme + Contraintes = résolution combinatoire, contraintes en premier

L'association des contraintes et du non-déterminisme donne une programmation combinatoire (parfois efficace) pour la résolution de problèmes ; souvent, si elle n'est pas efficace, elle est au moins simple à mettre en oeuvre.

La démarche classique :

```
probleme(donnees,solution):-
  enumération(donnees, candidatsSolutions),
  vérificationContraintes(candidatsSolutions,solution).
```

est remplacée par une résolution combinatoire, contraintes en premier :

```
probleme(donnees,solution):-
  vérificationContraintes(candidatsSolutions,solution),
  enumération(donnees, candidatsSolutions).
```

Exemple, coloriage de la carte de la Suisse (et pays limitrophes) :

- le problème : 4 couleurs, 5 pays, éviter de colorier les pays voisins avec la même couleur.
- solution classique : énumération des 4*4*4*4*4 (1024) cartes possibles, et pour chaque carte vérifications de contraintes de proximité (en général, les premiers coloriages sont uniformes, et pour sortir des solutions uniformes, cela prends du temps)
- solution aléatoire (pas si mauvaise, mais aléatoire)
- solutions combinatoire, contraintes en premier : mettre en place les contraintes se fait en une 8 étapes (une fois pour toute), l'énumération des couleurs évite les coloriages uniformes, c'est plus rapide

# 5 Contrôle de l'exécution : Coupure, ordre supérieur, gel, assert

## 5.1 Coupure : !

- mécanisme de base avec un arbre
- not ou + :
  `not(P):= P, !, fail.`
  `not(P).`
  mais attention au not(not(P)), ce n'est pas équivalent à P
- définition des coupure verte (ne réduit pas l'espace de solution, ex. une coupure après la requête pour une recherche de la premier solution) et des coupure rouge (peut réduire l'espace des solutions)

[105]:
```
%%writefile prog.pl
membre(E,[E|_L]).
membre(E,[_F|L]):-
  membre(E,L).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[106]:
```
!echo "not(membre(1,[2,0,2,4]))." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [not(membre(1,[2,0,2,4]))]
```

[107]:
```
!echo "not(membre(0,[2,0,2,4]))." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : []
```

[109]:
```
!echo "not(membre(X,[2,0,2,4]))." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : []
```

[110]:
```
!echo "not(not(membre(X,[2,0,2,4])))." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [not(not(membre(_2612,[2,0,2,4])))]
```

[111]:
```
!echo "membre(X,[2,0,2,4])." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [membre(2,[2,0,2,4]),membre(0,[2,0,2,4]),membre(2,[2,0,2,4
```

```
]),membre(4,[2,0,2,4])]
```

### 5.2 Autres

- Ordre supérieur (c'est findAll, call, apply : déjà vu ou utilisé ; peut permettre l'introspection)
- Gel (c'est une manière de gérer les contraintes : un prédicat est gelé sur une varaible, le moteur attend tant que la variable est inconnu, quand la variable est connue, l'exécution du prédicat est lancée)
- Assert (on ajoute des règles au programme par programme), cela peut donner naissance à des optimisations, et à une forme de programmation dite programmation dynamique (avec mémoïsation)