Introduction à PIV

Notation des intervalles de réels

cc : closed, closed [..]
oo open,open]..[
co, oc

Dans X=co(-1,1)=[-1,1[les bornes sont -1 et 1 et X NE PEUT PAS PRENDRE 1 car open en 1 mais X peut valoir -1 car fermé.

Saisi à la volée

```
consult.                % au clavier
consulting ..... % rentrer des règles/faits
   ....
end_of_file.            % ^D sous unix possible.
```

Pour rentrer un programme : compile, consult, reconsult...

• p 22 : Exemple d'incomplétude :

```
X=oo(-1,1), Y= oo(-1,1), gt(X,Y), lt(X,Y).
   ➔ X=oo(-1,1), Y= oo(-1,1)
```

NB: notons le '=' dans X=oo(-1,1). Il y a conversion en ~ par PIV.

NB : Ici, les 3 premiers ou les 2 premiers+4^e sont correctes mais la prise en compte de tous pose le problème de la complétude.

PIV résout exactement une contrainte mais devient incomplet quand on a plusieurs contraintes ensemble.

Incomplétude de PIV (à cause de la représentation approximée (approchée) des réels IEEE) :

- Si PIV donne une réponse alors le problème d'origine (problème math, physique,.. modélisé en PIV) a une solution
- Si PIV échoue alors le problème d'origine n'a pas de solution
- Si le problème a une solution alors PIV l'approche
  - Si le problème n'a pas de solution, PIV risque de ne pas s'en rendre compte (ex ci-dessus)

Problème (bug ?) corrigé en V1.1 (à enlever ou en faire un exemple des solveurs

```
X=cc(1,10) , Y=cc(1,10), Z=cc(1,10),
        square(Z)=square(X) + square(Y), Z=5 .
   ➔ Z = 5,                  % réponses moins précise, solveur  linéaire inadéquat
        Y ~ cc(1,10),
        X ~ cc(1,10).

X=cc(1,10) , Y=cc(1,10), Z=cc(1,10),
        square(Z)=square(X) .+. square(Y), Z=5 .
   ➔  Z = 5,           % réponses plus précise, solveur  non linéaire adéquat
        Y ~ cc(1,`<4.89898`),
        X ~ cc(1,`<4.89898`).

   Et (en mode simple)

X=cc(1,10) n int , Y=cc(1,10) n int, Z=cc(1,10) n int,
            square(Z)=square(X) .+. square(Y), Z=5 .
   ➔  Z = 5,
        Y ~ cc(3,4),
        X ~ cc(3,4).
```

*NB : **f(X,X)** : une contrainte ==> complet.*
*Mais c'est **il-existe A, f(A,A), eq(A,X)** ! ! !*
==> 2 contraintes => incomplet.
C-à-d : dès qu'on a rel(X,X) une contrainte binaire alors problème d'incomplétude.

Union et intersection

• p .23 'u' veut dire union (∪) **Commentaires : %** texte

```
X ~ -2 u 2                                  % ~  sur les claviers PC: alt 126
veut dire :
X appartient à {-2} ∪ {2}
```

Ex : mettre le mode union puis

```
set_prolog_flag(interval_mode, union).
sqrt(square(X))=2        ➔ X ~ -2 u 2
```

ZZZ : l'inconvénient de ce mode est que dans le cas d'une réponse général e (comme X ~ real) PIV essaiera d'énumérer le domaine pour donner toutes les réponses, ce qui peut aboutir à un " error: heap_overflow". Alors que sans ce mode union, il pourra se contenter de donner X ~ real.

'**n**' veut dire intersection (∩) (p.27)

Ex (en mode union):

```
X ~ cc(1,9) n  int.   ➔ X ~ 1 u 2 u .... u 9
X ~ co(1,9) n  int.   ➔ X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8.
                        % intervalle open
```

Dans ses réponses, PIV n'affiche plus les contraintes mais les intervalles.

int dénote tous les entiers (le grand N) mais c'est une contrainte (donc pas affichée)

Exemples de '**n**', '**int'**, '**times'**

```
% mode simple. Une réponse "real" ou tree ? provoquent un débordement

X ~ times(2,int).     ➔  X ~ real    % real veut dire "nombre non contraint"
X ~ 1 n times(2,int).     ➔  false.      %  times(2,int) : les entiers pairs.
X ~ 2 n times(2,int).     ➔ X = 2.
```

***ZZZ** : En mode union, on a souvent le message " error: heap_overflow" car on ne peut pas énumérer et afficher toutes les valeurs. Alors qu'en mode normal, on peut afficher " X ~ real" sans énumérer.*

*ZZZ : la requête **incomplète** times(2,int) donne erreur car elle est incomplète.*
'2 n 4' ou '2 n times(2,int)' donnent erreur de même que 'cc(1,5)' donne erreur

↳ Il faut une expression du genre X ~ times(2,int)
ou X ~ 1 n times(2,int).

Exemple (**nint** dénote les réels non entiers) :

```
X ~ times(2,int).     % times(2,int) dénote un réel
➔ X ~ real        % X est le double d'un entier quelconque : => X est pair.
                  % le type int n'existe pas comme contrainte affichée.
                  La requête  Int(X).      affiche:      X ~real      !

X ~ times(2,nint).
➔ X ~ real        % X est le double d'un non-entier : X est impair. (X est réel)

Et
X ~ times(2,int)+1          ➔ X ~ real      % suffit il pour les impairs ? (moi).

   X ~ 1 n times(2,int)+1 .  ➔ X = 1.
```

```
X ~ 12 n times(2,int)+1 .        ➔ false.
```

Pour que X soit un entier, il faut faire l'intersection de X avec **int**.

Exemples :

Pour avoir les nombres premiers entre 10 et 100, il suffit de les tester avec les diviseurs possibles qui sont les nombres premiers entre 2..7 (7 = sqrt(100/2)) :

```
X ~  cc(10,100) n int n  times(2,nint) .. (3,nint) ... 5 ... n times(7,nint).
```

et

```
X =cc(1,10), X ~ times(2,int).
➔ X ~ 2 u 4 u 6 u 8 u 10.
```

Exemple :

Pi est le nombre pi (non représenté en PIV car irrationnel).

ZZZ : il n'y a donc aucun 'nombre' de PIV qui peut précisément représenter pi car

'nombre' en PIV est un rationnel alors que pi est irrationnel.
On peut cependant donner pi dans un intervalle OUVERT (p.28).

Opérateur existentiel

• Existentiel : **X ex P** : il existe X P p. 24

```
X ex X=Y  ➔ Y ~ tree
X ex Y ex Z=f(X,Y) ➔   Z ~ f(tree, tree)
X ex X=X  ➔ true
```

• p. 25

'~' est comme '=' mais '=' pour des valeurs et '~' pour ensembles/intervalles.
PIV remplace nos écritures par le bon symbole.

PIV remplace par le bon quand il faut :
Terme ~ Terme : e.g. X~times(2,int) ou X=2+3
Terme ~ ss-terme =
ss-terme ~ Terme : Terme appartient à l'ensemble représenté par ss-terme
sous-terme ~ sous-terme : vrai si l'intersection des deux non vide (unification)

• p.26 :
On peut invariablement noter (notation Fonctionnelle et Relationnelle)
X=f(Y1,...,Yn) ou **f(X, Y1,...,Yn)** si f est une **contrainte**

Exemples :

```
X=sqrt(2.0).    ➔ X ~ cc(`>1.4142135`,`>1.4142136`).
sqrt(X,2.0).    ➔ X ~ cc(`>1.4142135`,`>1.4142136`).
```

```
ge(X,0)      équivaut à
X ~ ge(0)    où
             ge(0) : les nombres >= 0

u(X,1,2)     équivaut à
X ~ u(1,2)       où
             u(A,B) = l'union des ensembles A et B (ici {1,2})
```

En mode union :

```
% l'ensemble des X tels que leur carré > 1
gt(square(X),1)
ou
square(X) ~ gt(1)
➔ X ~ lt(-1) u gt(1).

% si Y est un carré alors il est positif ou nul.
X ex square(X)=Y.
➔ Y ~ ge(0)
```

Enumération

intsplit([les variables]).
Realsplit([les variables]).

Les opérateurs arithmétiques

• p.29 : Les opérateurs +,-,*,/ sont **partiels** sur R.

Les opérateurs +,-,*,/ sont utilisés pour les entiers (solveur linéaire).

Pour les réels (solveur non linéaire) :
plus, minus, times, div OU **.+. .-. .*. ./.**
+ et - unaires : **uplus, uminus** OU .+. .-.

(voir tableau plus bas)

ZZZ : on peut avoir des surprises si l'on mélange les solveurs.

```
.-.sqrt(4) = W .  ➔  W = -2.
-sqrt(4)=X .      ➔  X = -2.
```

Les opérateurs **+,-,*,/** travaillent dans le solveur **linéaire** !.

Exemples : trouver X, Y et Z tels que $25= X^2 + Y^2$ et $X+Y=7$

Rappel : les solveurs ne sont pas parfaits. Il faut souvent énumérer.

```
% Mode normale. Premier essai.
int(X), int(Y), X*X+Y*Y=25, X+Y=7 ,intsplit([X,Y]).
➔     Y ~ real,   X ~ real.                    % infinité de réponses

% Et en mode union.
int(X), int(Y), X*X+Y*Y=25, X+Y=7 ,intsplit([X,Y]).
➔     error: heap_overflow

% Mode union. contraintes sur les domaines. Contraintes pas suffisantes
X=cc(0,10), Y=cc(0,10) , X*X+Y*Y=25, X+Y=7.
➔ Y ~ cc(0,10), X ~ cc(0,10).

% Pour avoir plus de détails, X, Y et Z sont des entiers
X=cc(1,10) n int, Y=cc(1,10) n int, 25=X*X+Y*Y, X+Y=7 .
➔
Y ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10,
X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
```

Règle à vérifier : en mode union, si on veut les valeurs, il faut faire l'intersection?

```
% puisque polynôme 2e degré, opérations dans le solveur non linéaire
X=cc(0,10), Y=cc(0,10), 25=X.*.X.+.Y.*.Y, X.+.Y=7 .
➔
Y ~ cc(0,7),      % pour plus de détails, Il faut énumérer
X ~ cc(0,7).

% Enumération. Bonne solution (même dans le solveur linéaire)
X=cc(0,10), Y=cc(0,10) ,X*X+Y*Y=25, X+Y=7 ,intsplit([X,Y]).
➔     Y = 4, X = 3;
      Y = 3, X = 4.

% D'autres essais
%ajout des contraintes X > Y et X-Y=1 pour bien contraindre.
X gt Y, X*X+Y*Y=25, X+Y=7, int(X), int(Y), X-Y=1 . % gt = contrainte,
">"=test.
➔ Y = 3,  X = 4.

% énumération d'entiers (ne suffit pas). Ajout de contraintes int(X), int(Y).
X gt Y, X*X+Y*Y=25, X+Y=7, int(X), int(Y), intsplit([X,Y]).
➔  Y ~ real,  X ~ real.

% énumération de réels. Opération sur réels.
X gt Y, X.*.X.+.Y.*.Y=25, X.+.Y=7, realsplit([X,Y]).
➔
Y ~ cc(`>3.000061`,`>3.000122`),
X ~ cc(`<3.999878`,`<3.999939`);

Y ~ co(`>3.0000002`,`>3.000061`),
X ~ oc(`<3.999939`,`>3.9999997`);
....

% énumération de réels. Opération sur entiers.
X gt Y, X*X+Y*Y=25, X+Y=7, realsplit([X,Y]).
➔
Y ~ lt(-`>4e38`),
X ~ lt(-`>4e38`);

Y ~ lt(-`>4e38`),
X ~ oo(-`>4e38`,-`>3.4028232e38`);

Y ~ oo(-`>4e38`,-`>3.4028232e38`),
X ~ oo(-`>4e38`,-`>3.4028232e38`);
..... trop de réponses .....
```

Remarque sur le mode union :
En mode union, on a l'ensemble par extension de chaque réponse. Cette énumération a lieu pour chaque réponse. C'est pour quoi on a le message d'erreurs "over flow".
En mode normale, on a une réponse générale qui couvre toutes les réponses.

Par exemple, les réponses en mode union
X=1, X=3 , X=4 (X=2 absent)
sont regroupées et représentées en mode normale par
X ~ cc(1,4)
On constate que dans la réponse normale, on ne voit pas X # 2.

D'autres opérateurs

floor et **ceil** (le plus grand entier inférieur et le plus petit entier supérieur)
abs (valeur absolue)
min, max opérateurs binaires

Exemples :

```
X= max(A,B), A+B= abs(-10) , A-B=2 .
➔ B = 4, A = 6, X = 6.
```

Remarque : codage de "X in R" en PIV ? voir les opérateurs de domaine.

Réduction de domaines

• page 30 : des exemples de réduction de domaine.

L'ordre des contraintes n'a aucune importance.
La réduction s'applique aux entiers/réels.

Indexation de liste

• page 31 : indexation d'une liste (comme pour indexer un tableau)
Index(Ieme, Liste, I) même chose qu'élément **d'Eclipse** (syntaxe différente)**.**
Ieme est le Ième élément de Liste.

```
index(A,[1,3],2).  ➔  A = 3.
```

Autres notations : **Ieme = Liste : I = Ieme=index(Liste, I)**

```
A=[1,3]:2 .  ➔ A = 3.
```

Exemple :

```
index(C1,[bleu,blanc,rouge],I).  ➔  C1 ~ tree, I ~ cc(1,3).
Index(d,[a,b,c,d],X). ➔  X=4.
```

Dans un exemple comme celui de ‘nice-pair’ :

```
A=index([b,r,v,o],I), B=index([o,v,r,b],I), intsplit([I]).
➔ B = (o), I = 1, A = b;        % car o est une lettre réservée pour "conc"
➔ B = v, I = 2,A = r;
➔ B = r, I = 3, A = v;
➔ B = b, I = 4, A = (o).
```

Tableau des opérateurs

• page 31 :

uplus(t)	.+. t
uminus(t)	.-. t
plus(t1,t2)	t1 .+. t2
minus(t1,t2)	t1 .-. t2
times(t1,t2)	t1 .*. t2
div(t1,t2)	t1 ./. t2
upluslin(t)	+ t
umiuslin(t)	- t
pluslin(t1,t2)	t1 + t2
minuslin(t1,t2)	t1 - t2
timeslin(t1,t2)	t1 * t2
divlin(t1,t2)	t1 / t2
conc(t1,t2)	t1 o t2
n(t1,t2)	t1 n t2
u(t1,t2)	t1 u t2
index(t1,t2)	t1 : t2

NB :

- Les opérateurs unaires sont prioritaires sur les opérateurs binaires;
- * et / sont prioritaires sur + et - .
- Associativité des opérateurs binaires est gauche-droite :
- a op b op c : (a op b) op c (oui comme il faut)
- L'intersection est prioritaire sur l'union.

Un exemple intéressant : (**multiplication de domaines**)

```
X=cc(1,2) .*. 3. ➔ X ~ cc(3,6).
```

Remarque syntaxiques :

X=-1 pose problème, mettre un espace après '='

Enumération

• page 32 : Enumération

Exemple de triangle de Pythagore : $Z^2 = X^2 + Y^2$ et X,Y,Z dans 1..10.

```
set_prolog_flag(interval_mode, union).
X=cc(1,10), Y=cc(1,10), Z=cc(1,10), Z*Z=X*X+Y*Y.
➔ Z ~ cc(1,10), Y ~ cc(1,10),  X ~ cc(1,10).

% autre écriture du polynôme
X=cc(1,10), Y=cc(1,10), Z=cc(1,10), square(Z) = square(X) + square(Y).
➔ Z ~ cc(1,10), Y ~ cc(1,10),  X ~ cc(1,10).

% on précise que X, Y et Z sont des entiers
% square(X)=Y s'applique aussi bien aux entiers qu'aux réels.
%On filtre les entiers par "X=cc(1,10) n int"  ou par "int(X)"
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
          square(Z)=square(X) + square(Y).
➔ Z ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10,
  Y ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10,
  X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

%Mais, dans le solveur non linéaire
X=cc(1,10), Y=cc(1,10), Z=cc(1,10),  square(Z) = square(X) .+. square(Y).
➔
  Z ~ cc(`>1.4142135`,10),
  Y ~ cc(1,`>9.949874`),
  X ~ cc(1,`>9.949874`).

% et finalement.
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
          square(Z) = square(X) .+. square(Y).
➔ Z ~ 5 u 10,
  Y ~ 3 u 4 u 6 u 8,
  X ~ 3 u 4 u 6 u 8.

% Il faudra énumérer pour les valeurs précises. Par exemple
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
          square(Z) = square(X) .+. square(Y), intsplit([X]).
➔ Z = 5,  Y = 4, X = 3;
  Z = 5,  Y = 3, X = 4;
  Z = 10,    Y = 8, X = 6;
  Z = 10,    Y = 6, X = 8.
```

NB : A propos de 'Z .*. Z' pour 'square(Z)'

```
square(X) = X*X .          ➔ X ~ real.
Y=square(X).               ➔ X ~ real, Y ~ ge(0).
int(X), square(X)=Y. ➔ error: heap_overflow      en mode union
square(X)=Y, int(Y). ➔ error: heap_overflow
```

Le problème de mélange de solveurs supprimé dans V.1.1

```
% solveur linéaire
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
                          square(Z)=square(X) + square(Y), Z=5,X=3 .

➔    Z = 5,  Y = 4,  X = 3.

% solveur non linéaire
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
                          square(Z) = square(X) .+. square(Y), Z=5,X=3 .
➔    Z = 5,  Y = 4,  X = 3.
```

Différence des solveurs : Quand on donne (en mode union) :

```
X=cc(1,10) , Y=cc(1,10), Z=cc(1,10), square(Z)=square(X)+square(Y), Z=5 .
   ➔ Z = 5,
      Y ~ cc(1,10),
      X ~ cc(1,10).      ->  Aucune résolution n'est entreprise.

% Mais
X=cc(1,10) , Y=cc(1,10), Z=cc(1,10), square(Z)=square(X) .+. square(Y),
Z=5 .
   ➔ Z = 5,
      Y ~ cc(1,`<4.89898`),
      X ~ cc(1,`<4.89898`).
```

Le solveur linéaire ne le tente même pas d'éliminer des valeurs dans les domaines.

• Page 33 : **intsplit**

```
X=cc(1,10) n int, Y=cc(1,10) n int, Z=cc(1,10) n int,
   square(Z) = square(X) .+. square(Y), intsplit([Z,X,Y]).

➔ Z = 5,Y = 4,X = 3;
➔ Z = 5,Y = 3,X = 4;
➔ Z = 10,Y = 8,X = 6;
➔ Z = 10,Y = 6,X = 8.
```

• page 34 : **realsplit**

```
X=2 .*. sin(X).   ➔  X ~ cc(-2,2).
```

Pour avoir une réponse plus précise :

```
X=2 .*. sin(X), realsplit([X]).
X ~ oo(-`>1.8954951`,-`>1.8954933`); ..... 4 réponses
```

Le dessin de l'équation en page 34.

Liste, taille, concaténation, le type "list", le type "tree"

• page 36 :
X o Y est la même chose que **conc(X,Y)**
size(L) : la taille de la liste L

```
[1,2,3] o L = L o [1,2,3], size(L)=9 .
➔ L = [1,2,3,1,2,3,1,2,3].
```

Comme en PIII, la concaténation est retardée si nécessaire.

```
10 ~ size(L).
   ➔ L ~ [tree,tree,tree,tree,tree,tree,tree,tree,tree,tree].
N ~ size(L).          ➔  L ~ list, N ~ ge(0).
lt(0) = size(L).      ➔  false
N=le(0), N=size(L). ➔  L=[], N=0.

[a] o X = X o [a], 10= size(X).  ➔  X = [a,a,a,a,a,a,a,a,a,a].

 [tree] o X = X o [tree], 10= size(X).      % le type tree
➔ A ex
X = [A,A,A,A,A,A,A,A,A,A],
A ~ tree.
```

Ici, une variable existentielle de type **tree** est créée pour pouvoir afficher la liste X.

• Le type tree (type à créer ou à vérifier) :

```
tree(X).      ➔  X ~tree          % contrainte
write(tree).  ➔  _891    true.   % vérification : "tree" n'est pas un atome.
```

Donc, le mot " tree " a été considéré comme un type (et non un atome).
On peut faire idem avec **int, real**, ...

index et inlist

• page 36 : index et inlist

index(Neme, Liste, N)
Neme = index(Liste, N)
Neme ~ Liste : N

```
% 5 est le Nème élément dans la liste
5 ~  index([1,2,3,6, 7, 4,5],N).          ➔  N = 7.

% N=le range de 5 dans la liste : le Nème élément est 5
5 ~  index([1,2,5,3,6, 7,5, 4,5],N).      ➔  N ~ cc(3,9).

%En mode union
5 ~  index([1,2,5,3,6, 7,5, 4,5] , N).    ➔       N ~ 3 u 7 u 9.

% Autre écriture
5 ~  [1,2,5,3,6, 7,5, 4,5] : N.           ➔       N ~ 3 u 7 u 9.

% Quels sont les éléments de rang 1 à 4
X ~  index([1,2,5,3,6, 7,5, 4,5],N), N ~  cc(1,4).
➔      N ~ cc(1,4), X ~ cc(1,5)           ➔ Ce sont les éléments 1..5

% en mode union, on aurait plus de détails
➔      N ~ 1 u 2 u 3 u 4, X ~ 1 u 2 u 3 u 5.

% Remarquez l'absence de X=4 non visible en mode normale.
```

•Un problème intéressant :

Exemple : Quels sont les éléments d'une liste L dont la valeur est = le range dans cette même liste ? i.e. $I = X_i$.

Les éléments possibles :

ceux dont la valeur est <= à la taille de la liste (problème **approché**)

```
X  ~  index([1,2,5,3,6, 7,5, 11,4,5], X).
➔ X ~ cc(1,7).    % On s'arrête à 7, 11 est  plus grand que la taille de la liste.
```

Ce n'est pas une réponse **exacte**

En mode union, on aura les valeurs de X plus exactement.

```
set_prolog_flag(interval_mode,union).
X ~  index([1,2,5,3,6, 7,5, 11,4,5], X).
➔ X ~ 1 u 2 u 5 u 6 u 7.          % réponse approchée
```

- **Explication** de la réponse : parmi les réponses possibles, (1..7) :
  1 est dans la liste si la taille = 1
  2 est dans la liste si la taille = 2
  3 n'est pas dans la liste si la taille = 3
  4 n'est pas dans la liste si la taille = 4
  5 est dans la liste si la taille = 5
  6 est dans la liste si la taille =6
  7 est dans la liste si la taille =7

- Les éléments **exacts** : problème **complètement résolu**

```
X ~ index([1,2,5,3,6, 7,5, 11,4,5],X), intsplit([X]).
➔ X = 1; X = 2.
```

Donc, lorsqu'une réponse est donnée sous la forme d'un intervalle, la solution est seulement approchée. Ce n'est qu'en énumérant que l'on trouve la réponse exacte.

- **Exemple** : X est le 3eme élément de la liste L :

```
X ~ index(L,3).      ➔  L ~ [tree,tree,X | list], X ~ tree.
```

inlist est comme index mais s'occupe pas de l'indexation. Il traduit **l'appartenance**.
inlist(X, Liste) : X est membre le Liste

```
inlist(X,[1,2,4]). ➔ X ~ 1 u 2 u 4.
X ~ inlist([1,2,4]). ➔ X ~ 1 u 2 u 4.
X ~ inlist(L). ➔ X ~ tree, L ~ list.
```

Contrainte conditionnelle

- **if**. Page 38 et ? (voir p.171)
  if(a, b, c, d) ➔ si b=1 alors a=c sinon a=d. (remontée, fonctionnel ?)
  If/4 est une relation liant un booléen et 3 arbres.

```
A=if(X,Y=X, Y=2*X).
➔ Y ~ tree, A ~ tree, X ~ 0 u 1.
% rien ne permet de se prononcer sur une réponse plus exacte

A=if(X,Y=X, Y=2*X), X=1 .
➔ X = 1, A = (Y=1), Y ~ tree.
% X=vrai donc A=(Y=X=1). A = la valeur remontée (fonctionnel ?)

A=if(X,Y=X, Y=2*X), X=0 .
➔ X = 0, A = (Y=0), Y ~ tree.
% X=faux donc A=(Y=0). A = la valeur remontée. Y est un terme quelconque
```

On se sert de **if/4** pour construire des contraintes complexes dans lesquelles la vérification d'une contrainte conditionne la pose d'autres contraintes.

```
if(A, B, 1, 2).
➔ B = cc(0,1),   A = cc(1,2)

A=if(B, 1, 2), boolsplit([B]).
➔ B=0, A=2
➔ B=1, A=1

A ~ cc(1,3), A=if(B, cc(1,2), cc(4,5).
➔ B=1, A ~ cc(1,2)
```

if/4 est une **relation** et non une structure de contrôle (comme dans les langages impératifs).

```
A=if(1, tree, ntree)        ➔ false
```

car ntree est transformé en contrainte puis évalué et provoque un échec.

```
% Cette requête peut être mise à plat :
A= if(1, C, D), tree(C), ntree(D).
```

On voit que la contrainte ntree(D) apparaît explicitement, ce qui provoque un échec (toute variable est un arbre).
Moi : cela veut dire que malgré 1, ntree est évalué ? dépendance de C et D ?

En pratique, ce cas de figure se rencontre dans les requêtes du type :

```
A ~ if(bge(X,0), ln(X), ln(uminus(X))).        % contradiction
➔ false.
```

Ici, le problème vient du fait que ln(V,X) contraint X à être strictement positif alors que l'autre contrainte le contraint à être négatif, ce qui déclenche l'échec car dans A=ln(B), B doit etre >0.
Mais cela arrive aussi dans

```
A ~ if(1, ln(X), ln(uminus(X))).       % contradiction même si "1" ?
➔ false.
```

Notons que

```
A ~ if(bge(X,0), ln(X), ln(uminus(Y))).          % Y indépendant de X
➔      A ~ real,   Y ~ lt(0),   X ~ gt(0).
```

Et

```
ln(V,X), ln(V,uminus(X)).      % contradiction
➔ false.
```

Dans if(A,B,C,D), il faut absolument que les contraintes C et D soient valides (et non contradictoires ?) car elles sont posées avant même l'évaluation. Et ce même si la valeur de A (e.g. 1) permet de dire que C sera le résultat final (donc D ne nous intéresse pas). Il faut que D soit aussi valide. On remarque par ailleurs que dans if(A,B,C,D), on n'a pas une indépendance entre C et D. c-a-d, ce n'est pas comme dans (C ; D) :

```
ln(V,X); ln(V,uminus(X)).          % OU
➔    V ~ real,   X ~ gt(0);
➔    V ~ real,   X ~ lt(0).
```

Moi : est-ce que dans if(A,B, C,D), C et D ne doivent pas être contradictoires ?

Moi :
ZZZ : -X n'est pas la même chose que uminus(X).

```
ln(V,- X).
➔    X ~ real,   V ~ real.

Alors que
ln(V,uminus(X)).
➔    V ~ real,   X ~ lt(0).
```

Moi : je n'arrive pas à comprendre si C et D doivent être indépendants ?

```
int(X), list(X).
➔    false.          Incompatibles. Peut être PAS.
                     Trouver autre cas de contradiction.

Mais
A ~ if(bge(X,0), int(X), list(X)).          % incompatibles ? pas sure.
➔    A ~ tree,  X ~ real.

Ou
A ~ if(bge(X,0), int(A), list(A)).
➔    A ~ tree,  X ~ real.
```

Remarque :
Dans **if(A,B,C,D)**, A doit être un booléen avec une valeur 0 ou 1.
La valeur de A ne doit pas être un résultat du genre X=6, gt(X,5) car cette requête réussit mais ne vaut pas 1.
Il faut écrire X=6, bgt(A,X,5) pour que A=1.
Notons que bgt(2,1) seul provoque une erreur car la valeur résultat n'est pas récupérée.
Mais bgt(2,1) peut être utilisé dans un if/4 :

```
X=6, A=if(bgt(X,5), Y=1, Y=2).
```

➔ A = (Y=1), X = 6, Y ~ tree.

Moi :: Question :
pouquoi ces requêtes échouent ?

>> A=if(gt(X,5), Y=X, Y=2*X).
false. ➔ car gt(X,5) ne vaut pas O/1

>> A=if(gt(X,5), Y=1, Y=2).
False. ➔ idem. Mauvais notation

>> X=6, A=if(gt(X,5), Y=X, Y=2*X).
false. ➔ idem

Alors que
>> X=6, F=gt(X,5).
➔ F = (6 gt 5), X = 6.
>> gt(6,5). ➔ true.

>> X=6, bgt(F,X,5) .➔ F = 1, X = 6.

>> X=6, A=if(bgt(F,X,5), Y=1, Y=2).
false. ➔ Ici, F est de trop (comme si F empêchait que 1 remonte !)

Par contre (bonne forme)
>>X=6, F=bgt(X,5), A=if(F, Y=1, Y=2).
➔ A = (Y=1), F = 1, X = 6, Y ~ tree.

>> X=6, A=if(bgt(X,5), Y=1, Y=2).
➔ A = (Y=1), X = 6, Y ~ tree.

Booléens

• On peut s'en servir par tout.

Des entiers contraints à valoir 0 ou 1 (et non entre 0 et 1).
On a **impl**, **and**, **or**, **xor**,....

Exemple :
Pour écrire a=>b, b=>c, c=>a (i.e. a,b,c sont équivalentes : vrais (ou faux) en même temps).

% en mode union.
impl(A,B), impl(B,C), impl(C,A).
➔ C ~ 0 u 1, B ~ 0 u 1, A ~ 0 u 1.

On montre que B est un entier soit 0 soit 1.

```
impl(B,_), B=co(-1,1).   ➔ B = 0.
```

Ici, B prend ici 0 car l'intervalle est ouvert en 1 (B ne peut pas prendre 1)

```
impl(B,_), B=oo(0,1).   ➔ false.      % intervalle ouvert
impl(B,_), B=co(0,1).   ➔ B = 0.      % fermé en 0
impl(B,_), B=cc(0,1).   ➔ B ~ 0 u 1.  % fermé
```

Relations booléennes (prédicats)

• page 40

On peut combiner un booléen et les autres relations comme avec <u>la lettre **b au début**</u> pour récupérer le résultat de la relation.

☑ L'esprit de bxxxx est de pouvoir "manipuler" l'association entre certaine contrainte et sa "valeur de vérité". Cette association est une **relation**.

Format :
Si rel(X , Y) vaut vrai ou faux
Alors R=brel(X,Y) R vaut le résultat vrai ou faux de rel

Une autre forme de **R=brel(X,Y)** est **brel(R, X, Y)**.

Le résultat de brel peut participer à une opération arithmétique ou booléenne.

```
B= ble(X,0), X=3.
➔ X=3, B=0.     % le  donne  ble

% les X tels que X<= -1 ET X>=1 ?
and(ble(X,-1), ble(1,X)).
➔ false
```

Remarque :

```
and(ble(X,1), ble(1,X)).
➔ X=1
```

est la même chose que

```
le(X,1), le(1,X).       % conjonction
➔ X=1
```

mais ne pas écrire

```
le(X,-1) and le(1,X).
➔ false   % incohérent , faut la conjonction
          % (mais provoque pas d'erreur syntaxique)
```

MOI : and(X,Y) travaille sur "X,Y cc(0,1) n int", pas "le(A,B)" qui est vrai ou faux.
Pour pouvoir travailler avec 0 ou 1 (eg dans ad, or, ...), il faut bxx qui récupère les valeurs 0 ou 1 (bxx n'est pas seulement pour le retard).

```
% Les booléens ne valent que 0 ou 1
X=band(1,2).   ➔ false.        % incohérence ?
X=band(1,1).   ➔ X = 1.
X=bor(0,Y). ➔ X = Y, Y ~ 0 u 1.   % en mode union.
```

☑ Les relations bxxx permettent de **retarder la pose d'une contrainte :**

```
B=ble(X, -1), B=1.   ➔   B=1, X ~ le(-1).
La contrainte ble(X, -1) est posée quand B est fixé à 1, pas avant.

B=ble(X,-1).
➔ X ~ real, B ~ cc(0,1).  % aucune contrainte sur la valeur de X
```

☑ **Contrôler la pose d'une contrainte** ou de son **contraire** :

```
B=ble(X, -1), B=0.
➔ B = 0, X ~ gt(-1).      % contrainte imposée après B=0. gt est le contraire de le
```

Donc, bien faire attention à la différence entre le(X,0) et ble(V,X,0) :
l'un impose une contrainte, l'autre attend V ou X pour la suite.

Une remarque :

ble(V,X,0). ➔ X ~ real, V ~ cc(0,1).

Ci-dessous, il doit pouvoir raisonner et donner la valeur de X :

ble(V,X,0), V=X+1 . ➔ X ~ real, V ~ cc(0,1).

Pourquoi ne dit-il pas ? : (voir réponse)
V ne peut être que 0 ou 1
si V=0 alors gt(X,0) et X=-1 ➔ contradiction donc échec
Démo : ble(V,X,0), V=X+1, X= -1. ➔ false
si V=1 alors le(X,0) et X=0 ➔ succès
Donc, il doit pouvoir donner V=1, X=0 ?
Or, il le donne que si l'on énumère V.

Réponse: problème de solveur. On a : A SIGNALER

```
ble(V,X,0), V=X.+.1 .   ➔ X = 0, V = 1.
```

☑ **Construire des contraintes complexes :**

```
% Ici, binlist et bcc sont utilisés pour renvoyer un booléen pour and.
and(binlist(X,[3,4,5]), bcc(X,4.5,10)).   ➔  X = 5.

% Ce qui est différent de la requête ci-dessous qui renvoie une valeur pour X
inlist(X,[3,4,5]), cc(X,4.5,10).  ➔  X = 5.

% et de ci-dessous qui retard la pose des contraintes
A=band(binlist(X,[3,4,5]), bcc(X,4.5,10)).
➔     X ~ real,    A ~ 0 u 1.          % X inconnu, aucune contrainte posée
```

Remarque : dans

and(**binlist**(X,[3,4,5]), **bcc**(X,4.5,10)). ➔ X = 5.

Apparemment, binlist et bcc ne posent pas de contrainte mais alors pourquoi X=5?

Réponse :

D'abord, P. Bouvier dit qu'ils posent des contraintes car and(X,Y) n'est vrai que si X et Y sont=1. Imposer la contrainte and(X,Y) revient à imposer and(1,1). Alors,

```
binlist(X,[3,4,5]) = 1 et bcc(X,4.5,10)=1.   ➔  X = 5.
```

Ce qui revient à

```
binlist(X,[3,4,5]), bcc(X,4.5,10).          ➔  X = 5.          % conjonction.
```

Constat: Avant de partir dans un raisonnement sur les contraintes retardées, ne pas oublier l'esprit de bxxx qui est l'association de 0/1 à une contrainte (qui dans certains cas revient à du retardement, pas l'inverse).

Si une contrainte ordinaire (comme and) est construite avec de contraintes (bxxx), la contrainte ordinaire s'impose (normal !) et propage des valeurs, si possible, vars les bxxx

On peut le voir dans l'exemple ci-dessous (en mode intervalle) :

```
A=blist(B).       ➔  B ~ tree, A ~ 0 u 1.
```

Explication : on veut la valeur de vérité de list(B), A ne peut être que 0/1, B un arbre.

❋Idem dans

```
A=bint(X).          ➔  X ~ real, A ~ 0 u 1.
```

❋ Idem (problème du mode union)

```
and(bint(A), blist(B)).  ➔  error: heap_overflow
```

Par contre :

```
and(binlist(X,[3,4,5]), bcc(X,6,10)).   ➔  false
```

❋Un autre cas d'évaluation (and(1,1) est imposée):

Pour comprendre mettre à plat et énumérer X.

```
and(bcc(X,1,5), bcc(Y,X,2)).        ➔  Y ~ cc(1,2),      X ~ cc(1,2).
```

Les relations beq(X,Y) et bdif(X,Y) rendent une valeur de vérité si elles sont posées.

```
E=beq(X,Y), A=band(bcc(X,1,5), bcc(Y,2,4)).
➔
        Y ~ real,                % aucune contrainte
        X ~ real,                % on ne voit pas de trace de X=Y ? perdu?
        A ~ 0 u 1,
        E ~ 0 u 1.
```

***Question** : où est passé gt(X,Y) ?*

gt(X,Y), A=band(bcc(X,1,5), bcc(Y,2,4)).

➔

Y ~ real,
X ~ real,
A ~ 0 u 1.

perdu comme dans la question ?

gt(X,Y).

➔ *Y ~ real,*
X ~ real.

Réponse de Pascal : Non. Ce n'est pas perdu mais les solutions ne reflètent que les sous-domaines des variables et le système n'en sait pas assez pour déduire de meilleurs domaines.

Moi : mais, il pourrait répéter dans sa réponse gt(X,Y) ? Réponse : la réponse en tient compte, gt(X,Y) ne figure pas dans la réponse et donne toujours "real" comme dans gt(X,Y) tout court. Là aussi, le système ne répète pas gt(X,Y) dans sa réponse (PIV n'affiche pas toujours les contraintes).

Moi : une remarque :

J'ai mis à plat gt(X,Y), A=band(bcc(X,1,5), bcc(Y,2,4)) par :

```
gt(X,Y), A=band(K,L), K=bcc(X,1,5), L=bcc(Y,2,4).
```

Mais ca donne la même réponse. Je faisais une erreur en croyant que band devait prendre (1,1) mais non, c'est "and" qui le prend (un split de A le montre : toute les valeur possibles) d'où:

```
gt(X,Y), and(K,L), K=bcc(X,1,5), L=bcc(Y,2,4).
➔ L = 1,
      K = 1,
      Y ~ cc(2,4),
```

```
X ~ oc(2,5).
```

Remarquer que la simplification manuelle donne X ~ 3..5 (si X était entier) mais la réponse de PIV est X ~oc(2,5) car X et Y sont des réels.

Les bxxxx possibles

Il y a des bxxx (page 40 en bas) :
bimpl, bor, band,..., bcc, bco,..., ble, blt,..., beq, bdif, binlist, bint, bnint, ...

NB : pas de **bindex, btree**,...

Remarque : X=bxx(...) peut s'écrire bxx(X,...)
Exemple : X=bcc(A,1,3) peut s'écrire bcc(X, A, 1, 3).

Dans les bxxx(X,....),
si X est libre alors attendre
si X prend la valeur 1, la contrainte est posée
si X prend la valeur 0, l'inverse de la contrainte est posée

La forme bxxx retourne une valeur qui représente la valeur de vérité de la contrainte obtenue en enlevant la lettre **'b'** au début.

Remarque
Ca peut expliquer pourquoi il y a binlist mais pas bindex ? :
***inlist** est une contrainte mais **index** ressemble plutôt à un calcul ? ? ? ! ! !*

• p. 41 : la forme nxxx :

De même que **bxxx**, il y a **nxxx** pour la négation de la contrainte **xxx**.

La forme nxxx(X) contraint X à NE PAS vérifier xxx

xxx est la plupart du temps une relation d'appartenance à un sous domaine de PIV.

• p. 41 : Contraintes sur les arbres :

Les relations d'appartenance sur les arbres (avec leur forme nxxx/bxxx):
Appartenance aux arbres finis : **finite, infinite, bfinite, binfinite**
Appartenance aux listes : **list, nlist, blist, bnlist**
Appartenance aux feuilles : **leaf, nleaf, bleaf, bnleaf**
Appartenance aux identificateurs : **identifier, nidentifier, bidentifier, bnidentifier**

Appartenance aux réels : **real, nreal, breal, bnreal**

Exemples :

```
X = finite, X=f(f(X)). ➔ false.
X ~ finite, X=f(f(X)). ➔ false.
B=breal(B). ➔ B = 1.
B=leaf n list. ➔ B = []. % intersection
```

Certaines relations permettent de contraindre leur argument à appartenir au domaine du même nom : **Nom de relation = nom du domaine :**

```
tree, finite,          % l'inverse de finite est infinite
list, nlist, leaf, nleaf, identifier, nidentifier,
real, nreal,
cc(r1,r2), oo(r1,r2), ...          % ri = un réal
```

Ces noms peuvent apparaître dans les réponses de PIV avec la relation **compatible** (X ~ sous domaine)

On peut aussi avoir dans les réponses : nxxx(X), X n Y, X u Y (en mode intervalle)

NB : **int, nint et infinite** ne sont pas des sous domaines; seulement des contraintes. ??

NB (moi) :

```
ntree(X) ➔ false          % y a-t-il quelque chose qui ne soit un tree ?
==> ntree(X) : X n'existe pas
==> donc c'est l'inverse d'il-existe (ca serait = nex si ca existait ?)
==> Pour dire : quelque chose n'existe pas : ntree(X).
```

• page 42-43 : environnement de programmation,
options ligne de cmd, ISO, ...
(A mettre si on fait un mémento)

```
>> iso.        % pour se mettre en mode iso
?- prolog4.  % pour se mettre en mode PIV
>>
```

Les différences entre les deux syntaxes.
En ISO (Prolog Standard) :

- plus de pseudo-terme ; ils restent des vrais termes, NON EVALUES
- les contraintes doivent être en mode relationnel (préfixée, non fonctionnel)
- la notation fonctionnelle construit des termes (comme dans le prolog std)
- perte de précision sur les réels (en iso c'est le format IEEE)
- l'écriture "functor/arité" d'ISO pour la gestion des règles est interprétée comme une division linéaire en mode prolog4.

Entrée des règles

• p. 44 : les E/S

open, read, write, nl, close,....

Exemple :

```
open('fic.p4', read, S),        % S est le nom logique
   read(S,T),                        % read attend le '.' suivi de RC ou ' ' .
   close(S).

open('fout.data', write, S1), write(S1, 'salut tout le monde'), nl(S1),
   write(S1, toto(h,10/20)), nl(S1), close(S1).
```

A vérifier
Pour qu'un terme soit évalué lors de l'E/S, il doit être enfermé dans un foncteur comme le cas de toto(h,10/20).
Mais write(10/20) affiche bien 1/2

• p. 45 :

consult, reconsult % lit l'entrée courante. lenteur, consommation mémoire
consult(fic), reconsult(fic)
compile(fic)
end_of_file

Donner un paquet de règles d'un seul coup.
Si lecture au clavier alors pas de compilation.
Sous Unix, on peut lire 'dev/tty' pour lire au clavier ET compiler.

Factorisation des sorties

• P. 47 : :

```
% Par défaut ou avec
record(q_factorize, 0) .
X=f(g(2),g(2)).
➔ X = f(g(2),g(2)).

record(q_factorize, 1) .
X=f(g(2),g(2)).
➔
A ex
X = f(A,A), A=g(2).
```

Problèmes

• p. 48 - :

Mélange des solveurs numériques :

```
X=cc(1,2) .*. 3 - exp(1).     % le '-' linéaire binaire : minuslin
➔ X ~ real.                   % Noter l'opération sur le domaine de X

X=cc(1,2) .*. 3 .-. exp(1).   % le '-'binaire  sur réels: minus
➔ X ~ oo(`>0.281718`,`>3.2817182`).
```

Il ne faut pas mélanger des contraintes non linéaires avec des contraintes sur les réels.

Même genre de faute :

```
X = -pi.          ➔  X ~ real
X =  .-.pi.       ➔  X ~ oo(-`>3.1415927`,-`>3.1415925`).
X = uminus(pi). ➔  X ~ oo(-`>3.1415927`,-`>3.1415925`).
```

Mode union et overflow :

```
% en mode simple :
X ~  int, X  ~  cc(0,10).
➔X ~ cc(0,10).
```

```
% en mode union :
X ~  int, X  ~  cc(0,10).
➔ error : heap_overflow       % problème en mode union
```

La cause : PIV a essayé de construire l'ensemble (fini) d'entiers (représentés par des réels) mais il n'y a pas assez de mémoire pour construire cet ensemble.

Il faut d'**abord réduire** :

```
X  ~  cc(0,10), X ~  int.
➔ X ~ 0 u 1 u 2 u ... u 10.       % en mode simple, la réponse ne change pas.

%OU
X ~  int  n  cc(0,10).
➔ X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
```

Les contraintes glutonnes :

• Celles qui font intervenir des entiers :
int, nint, bint, bnint
• Celles qui font intervenir des fonctions périodiques :
floor, ceil, sin, cos, tan, cot. Plus généralement : les contraintes dont la fonction (ou sa réciproque) est non-continue

Le compilateur ré ordonne les contraintes au sein d'un pseudo-terme.

On peut par exemple donner l'une ou l'autre :

X ~ cc(0,10) n int.
➔ X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

X ~ int n cc(0,10).
➔ X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

Mais, pour cela (pour quoi? Glutonne ? et réordonner)

Il faut utiliser compile/recompile
Pas de compilation avec consult/reconsult, ...
Les débordements se produisent avec le mode union.

Spécification d'un intervalle en PIV

10 formes possibles (p. 71) :

real	(- infini, + infini).
ge(a)	X réel, X>= a
gt(a)	X réel, X>a
le(b)	X réel, X<= b
lt(b)	X réel, X< b
cc(a,b)	X réel, X dans [a,b]
oc(a,b)	X réel, X dans]a,b]
co(a,b)	X réel, X dans [a,b[
oo(a,b)	X réel, X dans]a,b[
ntree	l'ensemble/intervalle **vide**

Les sous domaines privilégiés de PIV

L'ensemble des arbres (X ~ tree)
L'ensemble des arbres finis (X ~ finite)
L'ensemble des arbres infinis (X ~ infinite)
L'ensemble des arbres qui sont des listes (X ~ list)
L'ensemble des arbres qui ne sont pas des listes (X ~ nlist)
Pour chaque entier n positif ou nul, l'ensemble des listes de taille n
Exemple avec n=3 (X ~ [tree,tree,tree])
Pour chaque n-uplet A1,...,An d'intervalles IEEE, l'ensemble des listes de la forme
[a1,...,an] avec ai dans Ai (X ~ [cc(1,2), co(4,8.1), gt(`>9.1`),real])
OU
Pour chaque n-uplet A1,...,An d'unions Ai d'intervalles IEEE, l'ensemble des listes
de la forme [a1,...,an] (X ~ [cc(1,2) u co(4,8), gt(9),real]) _

L'ensemble des arbres ayant un seul noeud (X ~ leaf)
L'ensemble des arbres ayant plus d'un seul noeud (X ~ nleaf)
L'ensemble des identificateurs (X ~ ident)
L'ensemble des arbres qui ne sont pas un simple identificateur (X ~ nident)
Pour chaque intervalles IEEE A, l'ensemble A (X ~ oc(`>2.1`, 9))
OU
Pour chaque union A d'intervalles IEEE, l'ensemble A
(X ~ oc(2.1,3) u cc(3,5) u gt(5))

L'ensemble des arbres qui ne sont pas un simple nombre réel (X ~ nreal)
Pour chaque arbre a doublement rationnel, l'ensemble {a}
(Y ex X=trio(X,Y,6), Y=duo(X,Y))
L'ensemble vide (X ~ ntree)

Les opérateurs importants

(pour les sous domaines privilégiés de PIV) :

abs/2, and/2, op trigos. P.74
band/2, bcc/4, bco/4, bdif/3, beq/3, bequiv/3, bfinite/2, bge/3, bgt/3, bidentifier/2, bimpl/3, binfinite/2, binlist/3, bint/2, ble/3, bleaf/1, blist/1, blt/3, bnidentifier/2, bnint/2, bnleaf/1, bnlist/1, bnot/2, bnprime/2 (non impl encore)**, ...**

voir page 74
Page 76 : spécifs des opérateurs classés
Pages 78.. axiomatisations à lire
page 92.... spécif programmes PIV, machine PIV,

Exemples de programmes

- **traduction nombres arabes <-> nombres romains**

Illustration des **listes** et de **size** :

```
    % la version symetrique de romain-arab
    % Algo :  ?      M       D       C       L   X    V   I
    %         5000  1000    500     100     50  10   5   1
    %               *               *           *
    % "*"  les romains intéressants.
    %
    %     Le nombre romaine sous forme d'une liste : ['I','I','X'...]
    %     Le nombre arabe sous forme d'une liste : ['1','2','5'...]
    %     De droite à gauche, on va traiter chiffre par chiffre

    %     Pour un nombre arabe A1 A2 A3,
    %     on a A3 : unités (I ou I,V ou I,X seront néc)
    %     pour A2 (dizaines), il faut écarter V,I romain (X : 10 romain)
    %                                   donc en présence de décan, il faut X
présent
    %     pour A1 (centaines), il faut  'C'  présent (donc sauter 2 positions
après X)
    %     le chiffre intéressant romain svt est M : les milliers
    %
    %     pour  '0' : éviter le nombre 0 (sous forme 0,00,000...) ok
    %     Pour '9' :ca vaut IX : vérifier qu'il y a X (et V et I) dans la liste de
référence

    go(A,R) :- romarab(A,R,['?', 'M', 'D', 'C', 'L', 'X', 'V', 'I']).
    romarab([],[],_).
    romarab(As o [0], R, S o [V,I]) :-          % si 0 à droite .. Pas le cas '00'.
       dif([0] o As, As o [0]),  romarab(As,R,S).
    romarab(As o [1], R o [I], S o [V,I]) :-    % 1 à droite, ajouter I
       romarab(As,R,S).
    romarab(As o [2], R o [I,I], S o [V,I]) :-      % 2 à droite, ajouter II
       romarab(As,R,S).
    romarab(As o [3], R o [I,I,I], S o [V,I]) :-  % 3 à droite, ajouter III ...
       romarab(As,R,S).
    romarab(As o [4], R o [I,V], S o [V,I]) :-  % 1 à droite, ajouter IV
       romarab(As,R,S).
    romarab(As o [5], R o [V], S o [V,I]) :-        % 1 à droite, ajouter V
       romarab(As,R,S).
    romarab(As o [6], R o [V,I], S o [V,I]) :-
       romarab(As,R,S).
    romarab(As o [7], R o [V,I,I], S o [V,I]) :-
       romarab(As,R,S).
    romarab(As o [8], R o [V,I,I,I], S o [V,I]) :-
```

```
        romarab(As,R,S).
    romarab(As o [9], R o [I,X], S o [X,V,I]) :-  % pour 9,= XI (faut X,I et le V au
milieu)
        romarab(As,R,S o [X]).        % on remet le troisième (tjs 1 saut de 2
romains)

    Questions :
    go([4],K).            ==>K = ['I','V'].
    go([1,4],K).          ==>K = ['X','I','V'].
    go(X,['V','I', 'X']).  ==>false.   Mauvais nombre
    go(X,['I', 'X']).      ==>X = [9].
    go([0,0],N).          ==>false.   Mauvais nombre
```

■ **Lemme de Shur**

Soit un entier n et m couleurs. On veut colorer tous les entiers 1,2,....,n de sorte que pour i,j,k entre 1..n, i+j=k entraîne les couleurs données à i, j et k ne sont pas toutes identiques.
Ex : n=5, les entiers 1,2,3,4,5 et on a
1+2=3 1,2,3 n'ont pas la même couleur
1+3=4 1,3,4 .... etc...

Shur a montrée que quel que soit m, il existe toujours un entier n assez grand pour lequel ce problème n'a pas de solution.

Illustration des **listes** et de **size** :
Ici, on prend m=3 et on va montrer que pour n=13 il y une solution mais pas pour 14.
Prendre éventuellement le programme page 98 (mais je n'ai pas bien compris la solution ?)

(A terminer éventuellement. Exemple difficile à comprendre)

■ Les entiers naturels

Exemple montrant les listes, contraintes, etc...

```
% Génération des entiers naturels
% Ce programme boucle à l'infini (c'est normal). Faut restreindre la profondeur

enumeration(L) :- nonegatifs(L), depuis(0,L).

nonegatifs([]).
nonegatifs([N| L]) :-int(N), ge(N,0), nonegatifs(L).        % ou "N ~ int n ge(0)"

depuis(_,[]).
depuis(N,La) :-
   dif(La,[]),
   superieurs(N,La,Lb),
   depuis(N.+.1, Lb).          % ZZZ ".+."

% superieurs(N,La,Lb) : On commence à 0 mais en cas de back-track, on impose,
% par la troisieme "superieurs" que les  nombres soient > N (au 1er coup, le
% dernier entier de la liste générée = 0 mais il devra etre > 0 !).
% Ceci va provoquer un échec car e.g. si on a produit une liste de 0, les zeros
% ne pourront pas etre > 0 donc échec et retoure à N+1....

% La nouvelle liste (Lb) sera utilisée pour les autres générations.
% Exemple :
% On génère une liste de 3 entiers = 0,0,0 ([X,Y,Z]) et N initial = 0
% sur un BackTrack, le 3eme "superieurs" impose à Z d'être > 0, échec, Lb contient
% X=0, Y=0, Z>0

superieurs(_,[],[]).
superieurs(N,[N|La],Lb) :-
   superieurs(N,La,Lb).
superieurs(N,[Na|La],[Na|Lb]) :-% passe en revue La et impose à ses élés Na
   lt(N,Na), superieurs(N,La,Lb).     % d'être > 0

/* Question :
 X ~ cc(2,5), Y = cc(1,4), Z=cc(3,6), Z=plus(X,Y),enumeration([X,Y,Z]).
➔
Z = 3,Y = 1, X = 2;
Z = 4,Y = 1,X = 3;
Z = 5,Y = 1,X = 4;
```

```
    Z = 6,Y = 1,X = 5;
    Z = 4,Y = 2,X = 2;
    Z = 5,Y = 3,X = 2;
    Z = 6,Y = 4,X = 2;
    Z = 5,Y = 2,X = 3;
    Z = 6,Y = 2,X = 4;
    Z = 6,Y = 3,X = 3.
    */
```

• De PIII à PIV : page 50.

• P . 61. Les bases de PIV :
prendre les pages74 à 78.
Les pages à lire sur l'axiomatisations.
• P. 93 : machine PIV

Divers

```
go :-
        for(I, 0, 15),
            for(J, 0, 15),
                write(I+J),
                fail.

for(X, X,B) :- X =< B.
for(X, A,B) :- A =< B, for(X, A+1, B).


moi :
>> consult.


union :- set_prolog_flag(interval_mode, union).
simple :- set_prolog_flag(interval_mode, simple).
end_of_file.
```