La **Pro**grammation **Log**ique : **PROLOG**

MASTER 1 TNSID

ISTV
Université de Valenciennes et du Hainaut-Cambrésis

E. ADAM

Introduction

- 1971 : création de Prolog par A. Colmerauer et P. Roussel à Luminy, suite à des travaux sur la détection automatique
- 1972, PROLOG I
- 1975, fin premier manuel PROLOG I
- 1977, premier compilateur PROLOG (Warren)
- 1978, multiplication des compilateurs PROLOG
- 1983,
    - machine abstraite (WAM) PROLOG (Warren) =>
        - Quintus-Prolog, SB-Prolog, SWI-Prolog, GNU-Prolog, …
    - PROLOG II -> Version commerciale
- 1989, PROLOG III -> solveur permettant de résoudre des contraintes linéaires
- 1996, PROLOG IV -> solveur, basé sur l'arithmétique des intervalles
- travaux sur compilation, implantations parallèle, algorithmes d'unification par contrainte, …

Terminologie de PROLOG

- *Base de faits* : ensemble de faits relatifs au problème traité, sous forme de clauses de Horn positives
  - Rex est un epagneul : epagneul(rex)
- *Base de règles* : ensemble de faits relatifs au problème traité, sous forme de clauses de Horn stricte
  - Un epagneul est un chien : epagneul(x) $\rightarrow$ chien(x)
- *Question* : demandée par l'utilisateur, sous forme de clause de Horn négative
- *Moteur d'inférences* : moteur non-monotone, à chaînage arrière, à régime par tentatives. Opère sur faits et règles. Tente de prouver l'inconsistance des faits et de la négation de la question

La syntaxe

- constantes :
  - chaînes de caractères commençant par une minuscule
  - nombres : entiers ou flottants
- VARIABLES :
  - chaînes de caractères commençant par une majuscule
- prédicats :
  - chaînes de caractères commençant par une minuscule
  - Peut posséder des arguments de type constante, variable ou prédicat :
    - pere(jerome, assia) ; pere(X, assia) ; non(pere(arnaud, assia))

La syntaxe

- Listes :
  - Suite d'objets séparés par des . et terminant par nil
    - a.b.c.nil ≡ [a,b,c]
  - Possibilités de sous-listes
    - a.(b.c).d.nil ≡ [a,[b,c],d]
- Les faits :
  - Ce sont des prédicats
    - epagneul(rex)
- Les règles : de la forme
  - predicatDeTete ← $predicat_1$, $predicat_2$,... $predicat_n$
    - epagneul(X) : – chien(X), chasseur(X).
    - Si X est un chien et X est un chasseur ALORS X est un epagneul

La syntaxe

- Conditions, conjonctions
  - grandPere(X,Y) : – pere(X,Z), pere(Z,Y).
  - grandPere(X,Y) : – pere(X,Z), mere(Z,Y).

  - X est le grandPere de Y SI
    X est le pere de Z ET Z est le pere de Y
    OU
    X est le pere de Z ET Z est la mere de Y

  - grandPere(X,Y) : – pere(X,Z), pere(Z,Y) ;
    pere(X,Z), mere(Z,Y).

La syntaxe

- Exemple de programme PROLOG :

```
epagneul(rex).
chien(X) :- epagneul(X).
mammifere(X) :- chien(X).

?- mammifere(X).
   X = rex ?
   yes
```

La syntaxe

- Charger le programme :
  - ?- [prog1].
    - charge le programme prog1.pl
  - ?- listing.
    - affiche le programme
  - ?- mammifere(X).
    - X = rex
  - ?- halt.
    - sortir de prolog

Principe de résolution

- système à une seule règle d'inférence ! sans axiome !
- Principe de résolution :

  Soit N une forme normale conjonctive (fnc),

  Soient C_1 et C_2 deux clauses de N

  Soit p un atome tq $p \in C_1$ et $\neg p \in C_2$

  Soit la clause $R = C_1 \setminus \{p\} \cup C_2 \setminus \{\neg p\}$

  Alors N et $N \cup R$ sont logiquement équivalentes

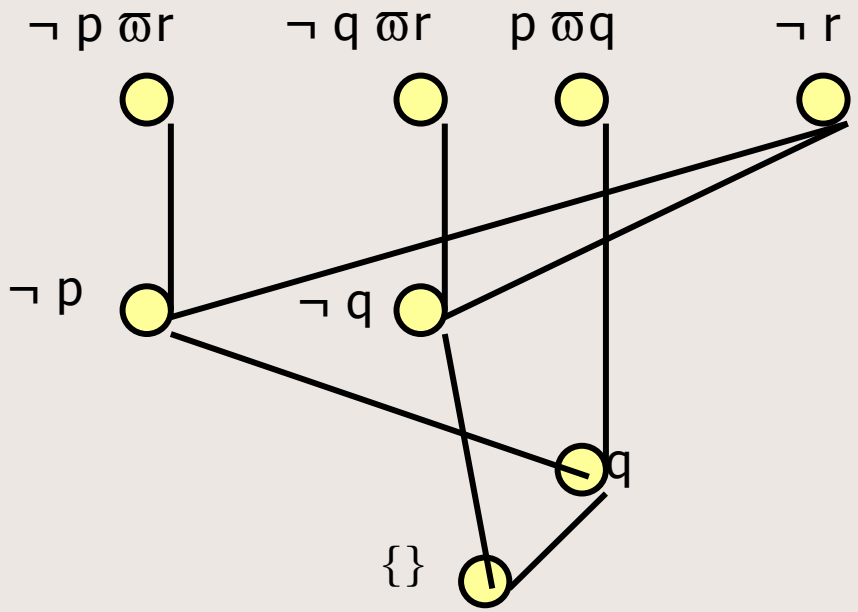
  R est appelée ***Résolvante*** de C_1 et C_2

Principe de résolution

Exemple : $\{ p \rightarrow r, q \rightarrow r \} \models (p \varpi q) \rightarrow r$?

Il faut prouver l'inconsistance de $\{\neg p \varpi r, \neg q \varpi r, p \varpi q, \neg r\}$

1 –	$\neg p \varpi r$	(hypothèse 1)
2 –	$\neg q \varpi r$	(hypothèse 2)
3 –	$p \varpi q$	(hypothèse 3)
4 –	$\neg r$	(hypothèse 4)
5 –	$\neg p$	résolution par 1 & 4
6 –	$\neg q$	résolution par 2 & 4
7 –	q	résolution par 3 & 5
6 –	$\{\}$	résolution par 6 & 7



Clauses de Horn

- clause de Horn : contient au plus un littéral positif
  - clause de Horn *stricte* : de la forme $p_1 \varpi \neg n_1 \varpi \neg n_2 \varpi \ldots \varpi \neg n_m$, $m \geq 1$
  - clause de Horn *positive* : de la forme p_1
  - clause de Horn *négative* : de la forme $\neg n_1 \varpi \neg n_2 \varpi \ldots \varpi \neg n_m$, $m \geq 0$

- Algorithme
  - soit une forme normale conjonctive N

    SI $\varnothing \in$ N ALORS N inconsistant => fin de la résolution

    SINON

    choisir $C \in$ N et $P \in$ N, tq P est une clause de Horn positive = p et $(\neg p) \in$ C

    choisir R la résolvante de P et C

    $N = (N \setminus \{C\}) \cup R$

    RETOUR DEBUT

Clauses de Horn

- Exemple :
  - Il faut prouver l'inconsistance de $\{(\neg p \vee r), (\neg r \vee s), (p), (\neg s)\}$

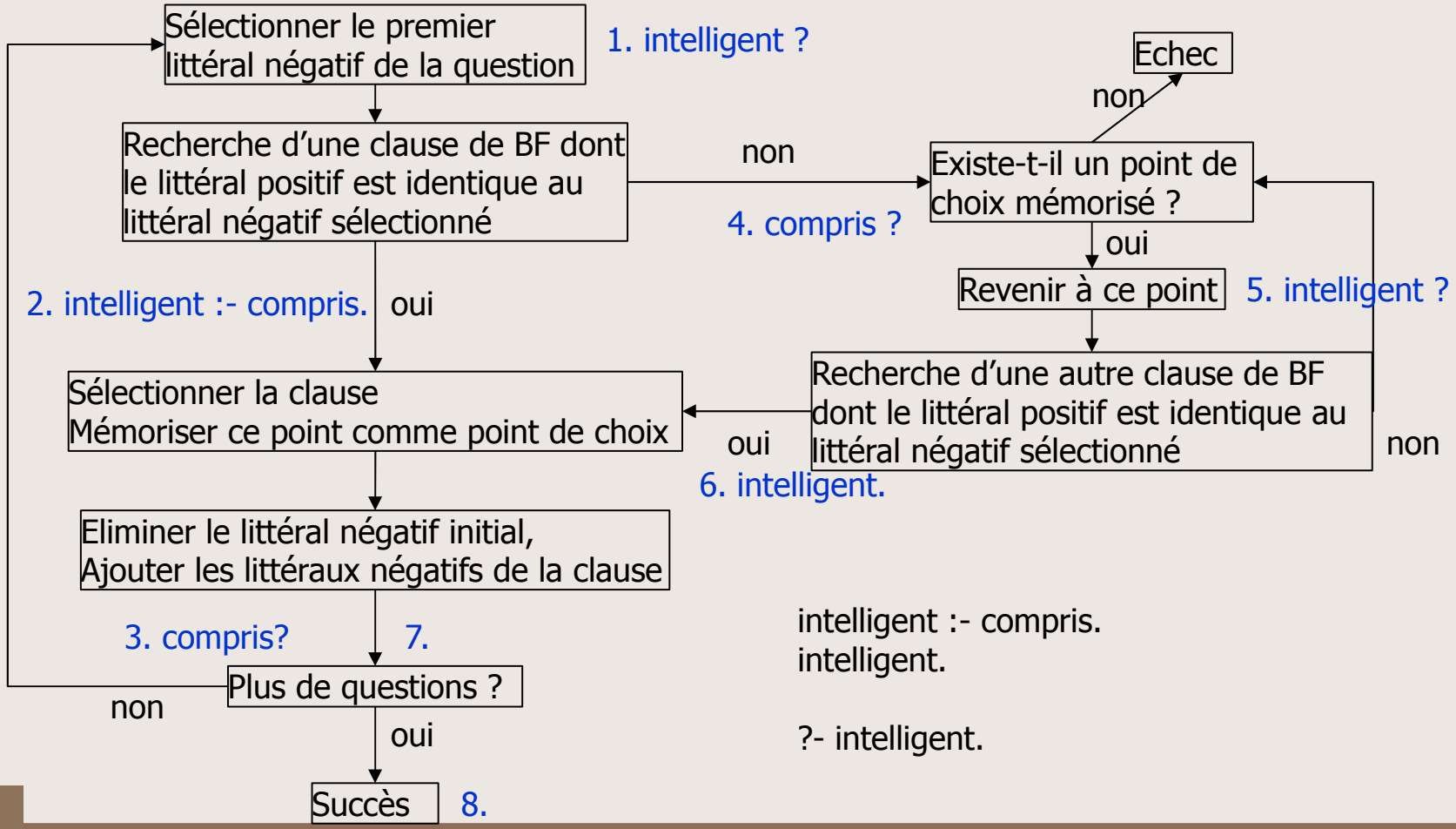
  1 – $\{(r), (\neg r \vee s), (p), (\neg s)\}$ par choix de P = p
  2 – $\{(r), (s), (p), (\neg s)\}$ par choix de P = r
  3 – $\{(r), (s), (p), (\varnothing)\}$ par choix de P = s
  donc l'ensemble des clauses est inconsistant

- Interprétation:
  - les clauses de Horn positives sont des faits
  - les clauses de Horn strictes sont des règles
    - $p_1 \vee \neg n_1 \vee \neg n_2 \vee \ldots \vee \neg n_m \Leftrightarrow (n_1, n_2, \ldots, n_m) \rightarrow p_1$
  - les clauses de Horn négatives sont les buts à atteindre

- on a donc prouvé : $(p \rightarrow r, r \rightarrow s) \models p \rightarrow s$

Moteur d'Inférences

- Démontre l'inconsistance d'une base des clauses de Horn

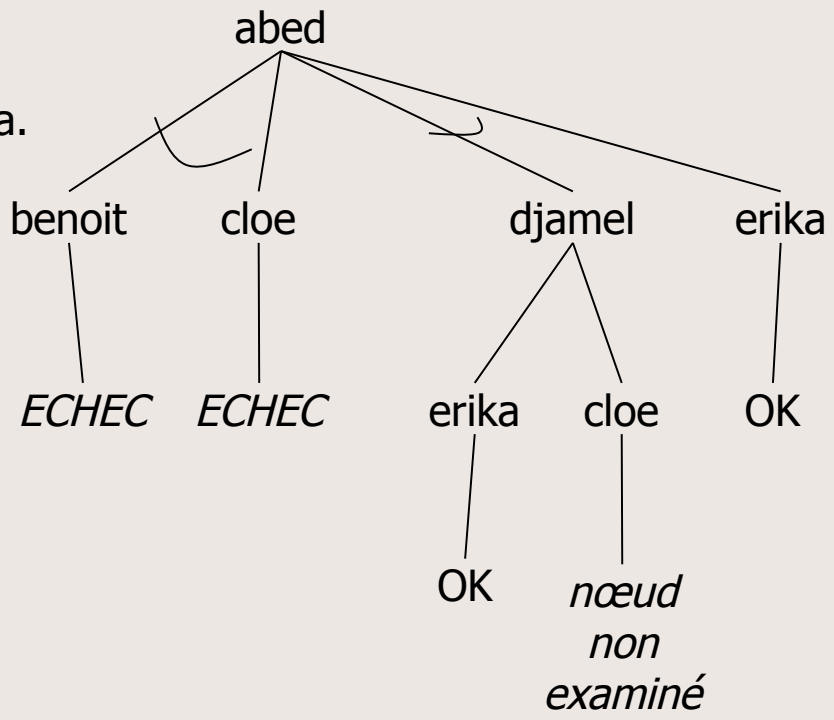


intelligent :- compris.
intelligent.

?- intelligent.

Résolution par parcours d'arbre ET - OU

- Exemple pour :
  - abed :- benoit, cloe.
  - abed :- djamel, erika.
  - djamel :- erika.
  - djamel :- cloe.
  - erika.
  - ?- abed.



- Que se passe-t-il si :
  - abed :- benoit.
  - benoit :- abed.
  - ?- abed.

Unification

- utilisation de prédicats à variables :
  - chien(fido).
  - mammifere(X) :- chien(X).
  - ?- mammifere(Y).
    - Y est une variable libre

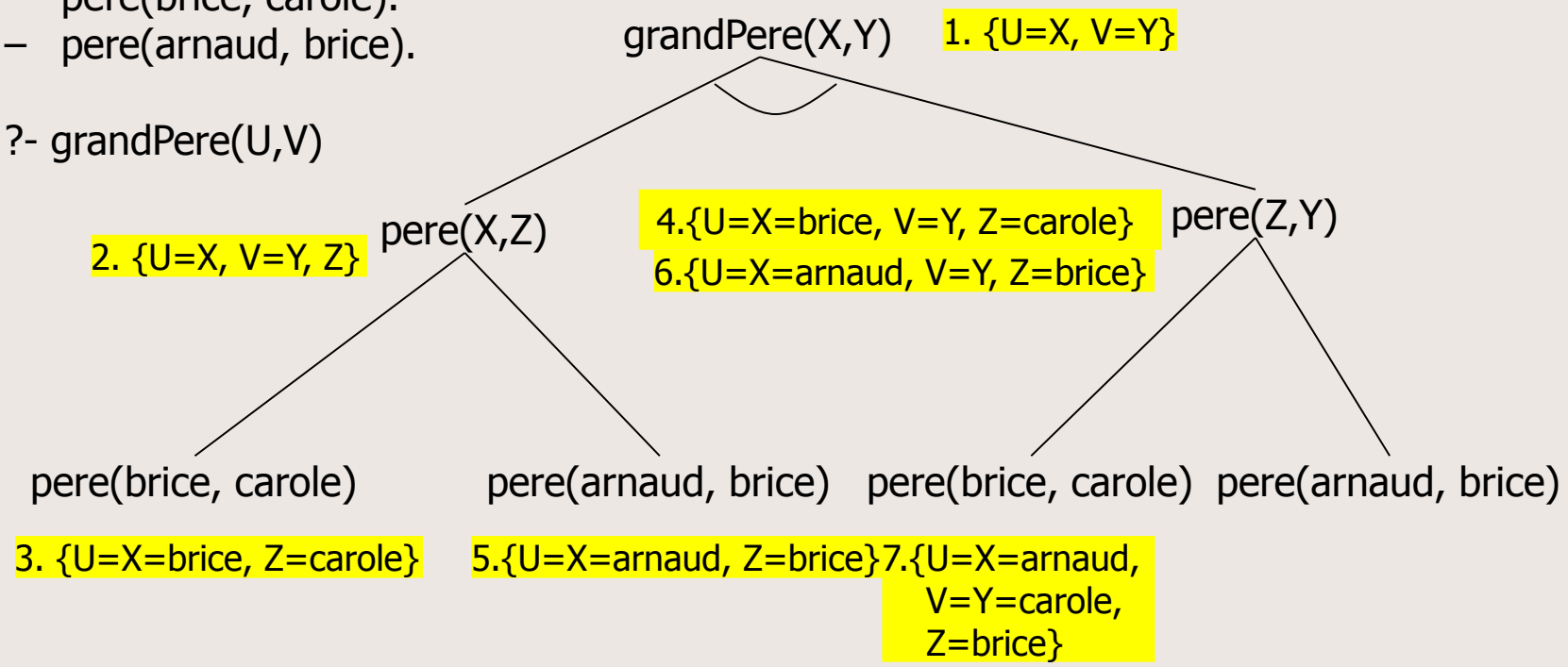
1. choisir une clause dont la tête est unifiable avec la question
   - mammifere(X) :- chien(X).
2. unifier la tete de la clause avec la question, X et Y sont liées. Y n'est plus libre, X oui
3. choisir une clause dont la tête est unifiable avec la question (chien(X))
   - chien(fido). → unification de chien(X) et chien(fido)
   - X est lié à la constante fido, par transitivité, Y également

   => affichage Y=fido

Unification

- Problème pour :
  - grandPere(X,Y) :- pere(X,Z), pere(Z,Y).
  - pere(brice, carole).
  - pere(arnaud, brice).

?- grandPere(U,V)



Exercice 1

- Rédiger le programme prolog capable de résoudre le problème suivant :
  - la chèvre est un animal herbivore
  - le loup est un animal cruel
  - un animal cruel est carnivore
  - un animal carnivore mange de la viande
  - un animal herbivore mange de l'herbe
  - un animal carnivore mange des animaux herbivores
  - les carnivores et les herbivores boivent de l'eau
  - un animal consomme ce qu'il boit ou ce qu'il mange
  - y a-t-il un animal cruel et que consomme-t-il ?
- vous répondrez en utilisant un arbre ET-OU

Exercice 2

- Soit le programme : (http://www710.univ-lyon1.fr/~nducloss/Prolog/Cours/)
  - bb(1).
  - bb(2).
  - cc(3).
  - cc(4).
  - dd(5).
  - dd(6).
  - aa(X,Y,Z) :- bb(X), cc(Y), dd(Z).

- Donner toutes les réponses à la requête
  - aa(X,Y,Z) dans l'ordre où Prolog les fournit

Listes

- ?- [A,B] = [1,2,3]
  - No
- ?- [A,B] = [1,2]
  - A = 1
  - B = 2
- ?- [A|B] = [1,2,3]
  - A = 1
  - B = [2,3]
- ?- [A|B] = [1]
  - A = 1
  - B = []

- Exercice : rédigez les fonctions existe(X, L), assemble(L1, L2, L3)

Listes et réécritures

- Rechercher un élément dans une liste :
  - // si X est le premier élément de la liste alors répondre Vrai
    - existe(X, [X | Z]) :- true.
  - // si X n'est pas le premier élément alors chercher dans la suite
    - existe(X, [Z | Y]) :- existe(X, Y).
  - Remarque 1 : Z n'est jamais utilisé, on peut le remplacer par une variable générique : '_' en prolog
    - existe(X, [X | _]) :- true.
    - existe(X, [_ | Y]) :- existe(X, Y).
  - Remarque 2 : un fait est toujours vrai. La première règle peut s'écrire sous la forme d'un fait. D'où :
    - existe(X, [X | _]).
    - existe(X, [_ | Y]) :- existe(X, Y).

Listes et réécritures

- Concaténer deux listes :
  - Si L1 est vide, le résultat est L2 :
    - assemble([], L2, Resul) :- Resul = L2.
      - L2 est unifié à Resul dans la partie droite. Prolog peut le faire lui même. On peut écrire :
        - assemble([], L2, L2).
  - Sinon, on dépile L1 et on empile Resul :
    - assemble([X|L1], L2, Resul) :- assemble(L1, L2, Resul2), ajoute(X, Resul2, Resul).

Listes

- Ce qui est équivalent à :
  - assemble([X|L1], L2, Resul) :- assemble(L1, L2, Resul2), Resul = [X | Resul2].
- Ou encore à :
  - assemble([X|L1], L2, [X | Resul2]) :- assemble(L1, L2, Resul2).
- Finalement, le code condensé pour concaténer deux listes est :
  - assemble([], L2, L2).
  - assemble([X|L1], L2, [X|Resul]) :- assemble(L1, L2, Resul).

Exercices

- Ecrire le programme permettant de donner :
  - l'intersection, la réunion, la réunion disjointe de deux listes.
- Le programme devra également permettre:
  - l'insertion d'un élément en queue, l'insertion à une position donnée, le remplacement d'un élément situé à une position donnée, la suppression et le remplacement d'un élément, la suppression et le remplacement de toutes les occurrences d'un élément, la recherche de l'élément minimal.
- Ecrire ensuite les règles permettant de trier une liste.

Un peu de calcul

- L'opérateur 'is' permet d'affecter le résultat d'un calcul à une variable.
  - Par exemple : X is 3*5.Les opérateurs possibles sont +, -, *, / et mod.
- Comparaison :
  - L'opérateur **=** retourne Yes si l'unification des deux termes réussit ou s'ils sont identiques :
    - pere(X) = pere(Y). tente d'unifier X et Y à la même valeur
    - pere(X) = pere(arnaud). unifie X à arnaud
  - L'opérateur **==** retourne Yes si les deux termes sont équivalents, il ne tente pas de réaliser l'unification.
    - pere(X) == pere(Y). répond faux sans tenter d'unifier
    - pere(X) == pere(X). tente d'unifier X à une valeur d'un fait
    - pere(X) == pere(arnaud). répond faux

Un peu de calcul

- Comparaison :
  - L'opérateur **=@=** retourne Yes si les deux termes sont structurellement égaux.
    - pere(X) =@= pere(Y). tente d'unifier X et Y à la même valeur
    - pere(X) =@= pere(arnaud). répond faux
  - L'opérateur **=:=** calcule la valeur des deux termes avant de tester leurs égalités.
    - 5 =:= 3+2. répond vrai
  - Pour exprimer la négation, il suffit de placer **** devant ces opérateurs.
  - Les opérateurs **<**, **>**, **=<**, **>=** et **=\=** effectuent le calcul avant de comparer les valeurs.

Un peu de calcul

- Le code suivant effectue la somme des entiers de 0 à N:
  - sum(N, S) :- sum(N, S, 0, 1). *initialisation*
  - sum(N, S, Sum, M) :- M =< N, *tant que M<=N*
    M1 is M + 1, *M1 <- M+1*
    Sum1 is Sum + M, *Sum1 <- Sum + M*
    sum(N, S, Sum1, M1).
  - sum(_, S, S, _).
    - cette dernière règle peut s'écrire
      sum(_, S, Somme, _) :- S = Somme.

    si M>N on place le résultat dans S, quelque soit la valeur des autres termes.

  - Ecrire le programme calculant la somme de 1 à N et le programme calculant la factorielle. Chaque programme ne contient qu'un seul type de règle défini en deux entrées.

Contrôles

- Prédicats de contrôle de recherche :
  - fail : échoue toujours
  - true : réussit
  - Repeat : réussit toujours
- Le **cut (!)** bloque le back tracking
  - exemple : écrire la règle Non(X) qui répond faux si X est vrai, faux sinon
    - non(X) :- X, !, fail.
    - non(_) :- true.
  - Remarque : en prolog, par cette règle, tout ce qui n'est pas vrai est supposé faux…

Négation

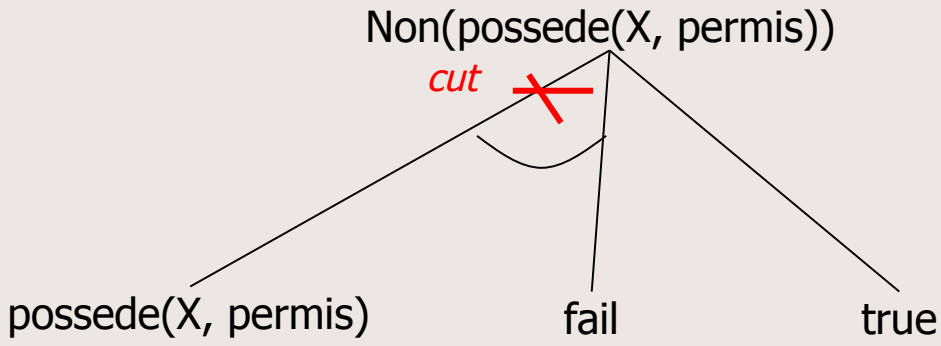
- Soit la base de faits :
  - personne(adrianna).
  - personne(brad).
  - personne(carla).
  - possede(brad, permis).
- Que se passe-t'il si Non est défini sans le cut :
  - non(X) :- X, fail.
  - non(X) :- true.
- A la question *non(personne(brad))?* Prolog répond *true* !
  - prolog effectue la première règle, teste 'personne(brad)', le trouve dans la base de fait et répond faux.
  - prolog effectue ensuite la règle suivante, et répond vrai (true).
  - *Donc, dans tous les cas, prolog passe par cette dernière ligne et répond toujours vrai à la question non(X) ? X étant un prédicat quelconque…*

Négation

- Replaçons le cut :
  - non(X) :- X, !, fail.
  - non(_) :- true.
- A la question *non(personne(brad))?* Prolog répond faux !
  - prolog effectue la première règle, teste 'personne(brad)', le trouve dans la base de fait, ***coupe tout retour de choix d'autres solutions*** et répond faux.
- A la question *non(personne(didjey))?* Prolog répond true !
  - prolog effectue la première règle, teste 'personne(*didjey*)', ne le trouve dans la base de fait, ***donc ne passe par par le cut***
  - prolog effectue ensuite la règle suivante, et répond vrai (true).

Négation

- A la question *non(possede(X, permis))?* Prolog répond faux !
  - ce qui signifie qu'il existe au moins une personne qui possède le permis



possede(brad, permis) ⟶ **No**

une solution a été trouvée, le cut empêche le choix d'autres solutions

- Pour connaître la liste des personnes sans permis, il faut poser la question :
  - *personne(X), non(possede(X, permis))?*
  - Ce qui force prolog à balayer l'ensemble des personnes pour tester si elle vérifie possede(X, permis)

Négation

- Soit le programme : (http://www710.univ-lyon1.fr/~nducloss/Prolog/Cours/)
  - bb(1).
  - bb(2).
  - cc(3).
  - cc(4).
  - dd(5).
  - dd(6).
  - aa(X,Y,Z) :- !, bb(X), !, cc(Y), !, dd(Z).

- Donner toutes les réponses à la requête
  - aa(X,Y,Z) dans l'ordre où Prolog les fournit

Prédicats

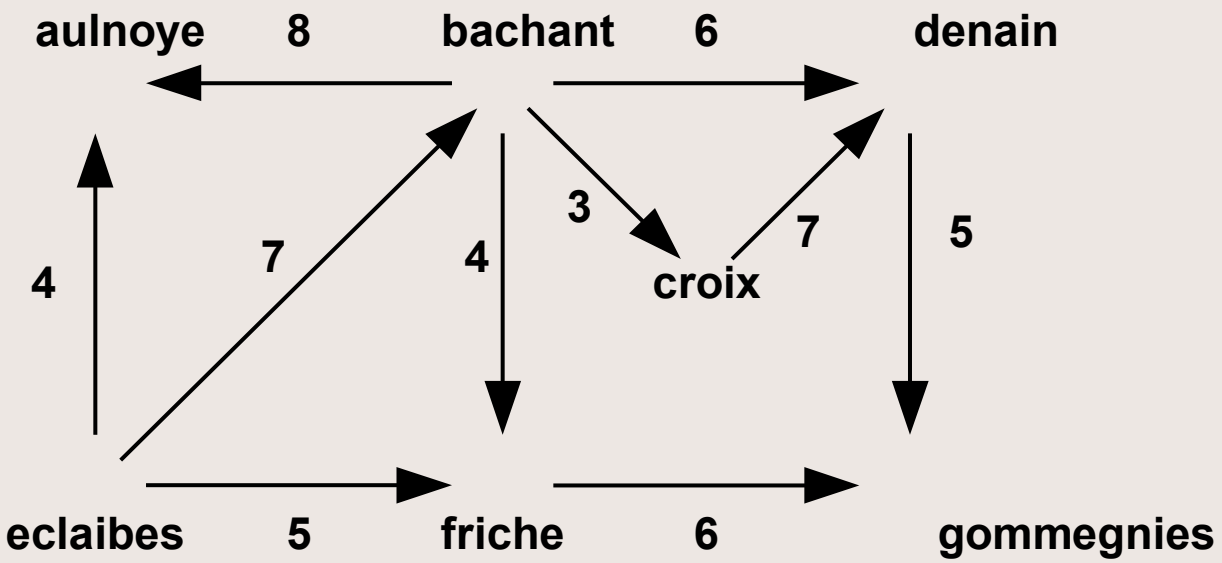
- IF THEN ELSE :
  - Utilisation de -> :
    - Si -> Alors; _Sinon :- Si, !, Alors.
    - Si -> _Alors; Sinon :- !, Sinon.
- \+ But : vrai si But ne peut être atteint

Méta Prédicats d'appel

- call(But) : invoque le But
- call(+Goal, +ExtraArg1, . . .)
  - Ajoute des arguments
- apply(+Terme, +Liste) : applique la liste des arguments
  - apply(plus(1), [2, X]) == plus(1, 2, X)
- not(+But)
- once(+But)
- ignore(+But) : Calls Goal as once/1, but succeeds,
  - ignore(But) :- But, !. ignore(_).

Exemple : Parcours de graphe

- Dans les graphes orientés, les chemins (les arcs) sont à sens unique entre deux sommets (ou nœuds).
- Supposons le graphe suivant :



Exemple : Parcours de graphe

Graphe Orienté

1. Décrire la base de faits sous la forme arc(bachant, aulnoye) décrivant cette carte.
2. Décrire les règles permettant de répondre aux questions :
   - 2.1. Peut-on aller de *eclaibes* à croix ?
   - 2.2. Comment aller de *eclaibes* à croix ?
   - 2.2. Quels sont les chemins menant à *aulnoye* ?
   - 2.3. Comment aller de *eclaibes* à *gommegnies* en passant par deux vias ?
   - 2.3. Comment aller de *eclaibes* à *gommegnies* avec un minimum de deux étapes?

Remarque : On décrira la règle `chemin(A, B, Liste)`,`Liste` étant l'ensemble des villes entre `A` et `B`.

Exemple : Parcours de graphe

Graphe Orienté – éléments de solution

1. pour aller de B, vers B, le chemin parcouru est [B]

   chemin(B, B, Trajet) :- Trajet = [B].

   ou

   chemin(B, B, [B]).

2. Pour aller de A vers B, il faut qu'il existe un arc de A vers C et un chemin de C vers B, le trajet est donc A auquel on concatène le chemin de B à C

   chemin(A, B, Trajet) :-

   arc(A, C), chemin(C, B, PetiteRoute), Trajet = [A|PetiteRoute].

   Ou

   chemin(A, B, [A|PetiteRoute]) :-

   arc(A, C), chemin(C, B, PetiteRoute).

3. exemple de question :

   ? **chemin(eclaibes, croix, T).**

Exemple : Parcours de graphe

Voyage coûteux

3. Définissez maintenant le prédicat arc en faisant intervenir le coût (exemple : arc(friche, gommegnies, 6)).
4. Définissez la règle chemin(A, B, Liste, Cout) donnant, pour chaque chemin, la liste des villes entre A et B ainsi que le coût du voyage.
5. Transcrire les questions :
    - 5.1. Quels sont les chemins entre bachant et gommegnies passant par denain, et quels sont leurs coûts ?
    - 5.2. Quels sont les chemins partant de eclaibes avec un coût supérieur ou égal à 11 ?

Exemple : Parcours de graphe

Voyage coûteux– éléments de solution

1. pour aller de B, vers B, le chemin parcouru est [B] et le coût est 0

   chemin(B, B, Trajet, Cout) :- Trajet =[B], Cout is 0.

   ou

   chemin(B, B, [B], 0).

2. Pour aller de A vers B, le cout est la somme de l'arc de A vers C et du chemin de C vers B

   chemin(A, B, [A|PetiteRoute], Cout) :-

   arc(A, C, coutAC), chemin(C, B, PetiteRoute, CoutCB), Cout is CoutAC + CoutCB.

3. exemple de question :

   (cout d'un trajet eclaibes, croix, pas d'affichage du trajet)

   ? **chemin(eclaibes, croix, _, C).**

Exemple : Parcours de graphe

Graphe Non Orienté

6. Ecrire maintenant la règle voisin(A, B) permettant de définir si deux villages ou deux villes sont voisins.

7. Ecrire la règle chemin(A, B, Trace, Liste) où Trace est la liste des villes déjà visitées.

8. Afficher les chemins possibles entre aulnoye et croix.

9. Ajouter maintenant la prise en compte du coût.

10. Afficher les chemins entre aulnoye et gommegnies de coût inférieur à 20.

Exemple : Parcours de graphe

Graphe Non Orienté – éléments de solution

1. Ville voisines

voisine(X, Y) :- arc(X, Y).

voisine(X, Y) :- arc(Y, X).

2. chemin(A, B, Historique, Liste) :

pour aller de B, vers B, le chemin parcouru est [B], l'historique est inutile

chemin(B, B, _, [B]).

pour aller de A vers B, il faut qu'il existe une ville voisine C de A et un chemin de C vers B, si on n'est pas déjà passé par C !

chemin(A, B, Historique, [A|PetiteRoute]) :-

voisine(A, C), NvelHisto = [A|Historique],

not(member(C, NvelHisto)),

chemin(C, B, NvelHisto, PetiteRoute).

Contraintes en Prolog

- Certains compilateurs permettent la programmation par contraintes :

  – SWI-Prolog par utilisation :
    - du module de solveur sur un domaine fini

      ```
      » :- use_module(library('clp/bounds')).
      ```

    - du module la consistance d'arc (réduction du domaine des variables à chaque ajout de contraintes)

      ```
      » :- use_module(library('clp/clp_distinct')).
      ```

  – GNU-PROLOG
    - si le module fd (solveur sur domaine fini) a été installé

Contraintes en Prolog

- Programmation par contraintes.
- Soit :
  - **L** une liste de Variables,
  - **D** le domaine de valeurs de ces variables,
  - **C** la liste des contraintes
- La résolution d'un CSP consiste à affecter à chaque variable de L une valeur dans D respectant les contraintes C

Contraintes en Prolog

- Affectation **totale** : toutes les variables reçoivent une valeur
- Affectation **partielle** : au moins une variable est sans valeur

- Affectation **consistante** : toutes les contraintes sont respectées
- Affectation **inconsistante** : au moins une contrainte n'est pas respectée

- CSP **sur-contraint** : trop de contraintes => pas de solution
  - Donner une valeur (une priorité) aux contraintes => CSP valué (VCSP)
- CSP **sous-contraint** : beaucoup de solutions différentes
  - Retenir la solution optimale => CSOP

Contraintes en Prolog : Carre Magique

- 9 chiffres de 1 à 9 doivent être placés dans un carre 3x3 de telle sorte que les sommes des lignes, des colonnes, des diagonales soient identiques

- **Ensemble des variables :**
  - Carre = [A0, A1, A2,
    B0, B1, B2,
    C0, C1, C2],
- **Domaine de valeur : 1 – 9**
  - Carre in 1..9,
  - vars_in(Carre, 1, 9),

fd_domain(Carre, 1, 9),

swi-prolog *gnu-prolog*

Contraintes en Prolog : Carre Magique

- **Ensemble des contraintes :**
  - Unicité des chiffres :
  - all_distinct(Carre), *swi-prolog* fd_all_different(Carre), *gnu-prolog*
  - Sommes des lignes, colonnes, diagonales
    - LigneA #= A0+A1+A2, LigneB #= B0+B1+B2 , LigneC #= C0+C1+C2, Colonne0 #= A0+B0+C0, Colonne1 #= A1+B1+C1, Colonne2 #= A2+B2+C2, Diag0 #= A0+B1+C2, Diag1 #= A2+B1+C0, LigneA #= LigneB, LigneB #= LigneC, LigneC #= Colonne0, Colonne0 #= Colonne1, Colonne1 #= Colonne2, Colonne2 #= Diag0, Diag0 #= Diag1,
- **Affecter une valeur aux variables:**
  - label(Carre). *swi-prolog* fd_labeling(Carre). *gnu-prolog*

Contraintes en Prolog : Carre Magique

- **Quelques contraintes Swi-Prolog avec clp/bounds et clp/distinct:**

- **all_distinct(*Lc*)** : Unicité des valeurs des variables contraintes de Lc
- *VARc* **in** *B..H* : contraint *VARc* aux valeurs entre *B* et *H*
- *Lc* **in** *B..H* : valeurs de *Lc* contraintes aux valeurs entre *B* et *H*
- *Ac* **#<** *B* : *Ac* est contraint d'être inférieur à *B*
- Ac **#>** B, Ac **#=** B, Ac **#\=** B, Ac **#=<** B, Ac **#>=** B
- **sum**(*Lc*, *Op*, *valeur*)
  - *Lc*= listes de variables contraintes, *Op* = opérateur binaire (#=, #<, ...)
  - Contraint la somme des variables à être *Op* à *valeur*

Contraintes en Prolog : Carre Magique

- **Quelques contraintes Swi-Prolog avec clp/bounds et clp/distinct:**

- **in_domain**(*VARc*) : affecte une valeur à *VARc*
- **label**(*ListeC*) : chaque variable reçoit une valeur respectant les contraintes

Contraintes en Prolog : Carre Magique

- **Quelques contraintes avec gnu-Prolog :**

- **fd_all_different(*Lc*)** : Unicité des valeurs des variables contraintes de *Lc*
- **fd_element**(*I, Liste, VARc*) : contraint *VARc* à être égale au *I*ème élément de *Liste*
- **fd_element_var**(*I, ListeC, VARc*) : idem mais *Liste* peut contenir des variables contraintes
- **fd_atmost**(*N, ListeC, V*) : au plus *N* variables de *ListeC* peuvent être égale à *V*
- **fd_atleast**(*N, ListeC, V*) : au moins *N* variables **...**
- **exactly**(*N, ListeC, V*) : exactement *N* variables **...**

Contraintes en Prolog : Carre Magique

- **Quelques contraintes avec gnu-Prolog :**

- Ac **#<** B : A est contraint d'être inférieur à B
- Ac **#>** B, Ac **#=** B, Ac **#\=** B, Ac **#=<** B, Ac **#>=** B

- **fd_domain**(*ListeC, Haut, Bas*) : définit le domaine de chaque variable de *ListeC*

- **fd_labeling**(*ListeC*) : chaque variable reçoit une valeur respectant les contraintes

Contraintes en Prolog : sur domaine fini

- **Avec swi-prolog:**

  - Utilisation de clpfd « :- use_module(library(clpfd)). »
    - Entier, boolean
    - Possibilité d'énumération
  - Par rapport à la gestion sur entiers, ajout de :
    - #\ Q : vrai ssi Q est faux
    - P #\/ Q : vrai ssi P ou Q est vrai
    - P #/\ Q : vrai ssi P et Q sont vrai
    - P #<==> Q : vrai ssi P et Q sont équivalents are equivalent
    - P #==> Q : vrai ssi P implique Q
    - P #<== Q vrai ssi Q implique P
    - sum(Vars, Op, Expr) : somme des variables OP Expression

Contraintes en Prolog : sur domaine fini

- **Exemple :**

```
:- use_module(library(clpfd)).

puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
   Vars = [S,E,N,D,M,O,R,Y],
   Vars ins 0..9,
   all_different(Vars),
   S*1000 + E*100 + N*10 + D +
   M*1000 + O*100 + R*10 + E #=
   M*10000 + O*1000 + N*100 + E*10 + Y,
   M #> 0, S #> 0.
```

- `puzzle(As+Bs=Cs), append([As,Bs,Cs], Vs), label(Vs).`

Contraintes en Prolog : sur domaine fini

- Options de placement de valeurs :
  - labeling(Options, Vars)
    - Options =
      - leftmost : définition de variables dans l'ordre d'apparition. Par défaut
      - ff : First fail. D'abord la variable de plus faible domaine
      - min : variable dont la valeur serait la plus petite
      - max : variable dont la valeur serait la plus grande
      - up: tente les éléments en ordre ascendant. Par défaut
      - down: tente les éléments en ordre descendant.

Contraintes en Prolog : sur réel et rationnel

- **Avec swi-prolog:**

  Utilisation de `clpq ou clpr`

  « :- use_module(library(`clpr`)). »

  « :- use_module(library(`clpq`)). »

- **{}(+Contraintes) : ajoute les contraintes à la base de contraintes**
- **entailed(+Contrainte) : succès si Contrainte est vraie selon la base**
- **inf(+Expression, -Inf) : calcule le minimum de l'expression**
- **minimize(+Expression) : idem**
- **sup(+Expression, -Sup) : calcule le maximum de l'expression**
- **maximize(+Expression) : idem**
- . . .

Contraintes en Prolog : sur réel et rationnel

- **dump(+Cible, +Valeurs, -Reponse) :**
  - **Retourne les contraintes sur les variables de cible, en remplaçant les variables par les Valeurs**
- **Exemple de contraintes :**
  - **{X =:= Y} {C = cos(B)}, {A = B + C} .....**
- **{}(+Contraintes) : ajoute les contraintes à la base de contraintes**

- `{ 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y }, sup(Z, Sup).`
  - **Sup = 310.0**

Simplex en Prolog

- **Résolution de problèmes linéaires**
  - assignment(+Cout, -Assignement) : Cout est une liste de listes (représente la matrice quadratique des couts), element (i,j) est le cout de l'assignement de l'entité i à l'entité j. Un assignement de cout minimal est calculé et représenté comme une matrice d'adjacence
  - **constraint(Contraintes, S0, S)** : ajoute une contrainte d'inégalité linéaire
  - **gen_state(Etat)** : génére un état initial correspondant à un programme linéaire vide
  - maximize(Objectif, S0, S) : maximise la fonction d'objectif
  - minimize(Objective, S0, S)
  - objective(Etat, Objectif) : Unifie l'objectif avec le résultat de la fonction objectif. L'Etat doit correspondre à une instance résolue

Simplex en Prolog

- shadow price(Etat, Nom, Valeur) : Unifie la valeur avec le prix caché de la contrainte linéaire Nom
- transportation(Requetes, Demandes, Couts, Transport) : Requetes et Demandes sont des listes de nombres positifs. Leurs sommes doivent être égales. Le couts représente la matrice des coùts. Un plan de transport de coût minimum est calculé
- variable_value(Etat, Variable, Valeur) : Valeur est unifiée avec la valeur de Variable. Etat correspond à une instance résolue

- **Exemple**

```
:- use_module(library(simplex)).

post_constraints -->
   constraint([0.3*x1, 0.1*x2] =< 2.7),
   constraint([0.5*x1, 0.5*x2] = 6),
   constraint([0.6*x1, 0.4*x2] >= 6),
   constraint([x1] >= 0),
   constraint([x2] >= 0).

resolution(S) :- gen_state(S0), post_constraints(S0, S1),
   minimize([0.4*x1, 0.5*x2], S1, S).

?- resolution(S), variable_value(S, x1, Val1), variable_value(S,
   x2, Val2).
```

Simplex en Prolog

- **Exemple 2 :**

- knapsack_constrain(S) :-
  gen_state(S0),
  constraint([6*x(1), 4*x(2)] =< 8, S0, S1),
  constraint([x(1)] =< 1, S1, S2),
  constraint([x(2)] =< 2, S2, S).
- knapsack(S) :- knapsack_constrain(S0), maximize([7*x(1), 4*x(2)], S0, S).
- ?- knapsack(S), variable_value(S, x(1), X1), variable_value(S, x(2), X2).

Types de données

- var(+Term) : une variable libre
- nonvar(+Term) : une variable non libre
- integer(+Term) : lié à un entier
- float(+Term) : lié à un réel
- rational(+Term) : lié à un nombre rationnel (un entier est un nombre rationnel).
- rational(+Term, -Numerator, -Denominator) : lié à un nombre rationnel avec le numérateur et le dénominateur donné
- number(+Term) : lié à un réel ou un entier
- atom(+Term) : lié à un atome
- string(+Term) : lié à une chaîne

Types de données

- atomic(+Term) : lié à un atome, une chaîne, un entier ou un réel
- compound(+Term) : lié à un prédicat (pere(X))
- callable(+Term) : lié à un atome ou un prédicat
- ground(+Term) : vrai si le terme ne contient pas de variable libre
- cyclic term(+Term) : vrai si le terme contient des cycles
- acyclic term(+Term) : vrai si le terme ne contient pas de cycles; il peut être exécuté en un temps fini

Base de données

- ***RETRAITS :***
  - abolish(+Name, +Arity): enlève la clause Name d'aritée Arity
  - retract(+Term): efface le terme
  - retractall(+Head) : efface tous les termes dont la tête s'unifie avec Head
  - erase(+Reference) : efface l'élément pointé par Référence
- ***AJOUTS***
  - assert(+Term) : ajout un fait ou une clause dans la base, en fin de liste
  - asserta(+Term) : ajout un fait ou une clause dans la base, en tête de liste
  - assertz(+Term) : idem
  - Possibilité d'associer des références, d'obtenir des clés

Base de données

- ***AJOUTS***
  - recorda(+clé, +Terme) : ajout un fait ou une clause dans la base, avec clé = petit entier
  - recordz(+clé, +Terme) : idem

- ***Anoncer la dynamicité*** :
  - **dynamic** *+PredicateIndicator*, . . .
  - Exemple :
    - :- **dynamic**
      afficher/0,
      parent/2.

Base de données

- ***Compiler*** :
  - compile predicates(:ListOfNameArity)
    - Attention : plus d'ajout possible
- ***Autres fonctions***
  - *multifile* +PredicateIndicator, . . .
  - *discontiguous* +PredicateIndicator, . . .
  - *index(+Head)* : indèxe les clauses avec les mêmes têtes

Recherche de solutions

- findall(+Motif, +But, -Liste)
  - Liste est la liste de Motif répondant au But
  - findall(X, personne(X), Liste).
    - Liste = [adrianna, brad, carla]
- bagof(+Motif, +But, -Liste)
  - Idem que findall, mais échoue en cas de non solution
- setof(+Motif, +But, -Liste)
  - Idem que bagof, mais sans doublon dans Liste

Recherche de solutions

- L'opérateur ^ signale les variables devant rester libres.
  - triplet(a,b c).
    triplet(a,b d).
    triplet(d,e f).
    triplet(g,h i).
    bagof(Z, triplet(X, Y, Z), Liste).
  - X = a, Y = b, Liste = [c, d]
    X = d, Y = e, Liste = [f]
    ...
  - bagof(Z, X^Y^triplet(X, Y, Z), Liste).
  - X = _G333, Y=_G343, Liste = [c, d, f, i]

fonctions

- functor(predicat, nomPredicat, nbArgs)
  - Prend en entrée :
    - Un predicat
    - Un nom de predicat
    - Le nombre de variables
      - Ex: ?- functor(pere(amine, susanne), P, A).
        - » P = pere
        - » A = 2

fonctions

- arg(noArgument, predicat, valeur/variable)
  - Prend en entrée :
    - Un numéro d'argument
    - Un predicat
    - Une valeur ou variable
      - Ex: ?- arg(2, pere(amine, susanne), F).
        - » F = susanne
      - Ex: ?- arg(2, pere(amine, F), susanne).
        - » F = susanne

fonctions

- La notation « =..[a1, a2, ..., an] »
  - Permet de définir des prédicats:
    - Ex: ?- P=..[pere,amine, susanne]
      - P=pere(amine, susanne)
    - EX: ?- pere(amine, susanne) =.. X.
      - X=[pere, amine, susanne]