$\forall \equiv \neg \exists \neg$

```
fact(X,FX) :-  Y is X-1,
               fact(Y,FY),
               FX is X*FY.
```

PROLOG :- programmation logique.

Sylvain Piechowiak

$\forall x \, p(x) \rightarrow \exists y \, p(y)$

$\forall x$

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT CAMBRÉSIS
Le Mont Houy 59313 VALENCIENNES CÉDEX 9 Tél 33 (0)3 27 51 14 38 fax 33 (0)3 27 51 13 16

Prolog :- programmation logique.

Historique

- **1930** Calcul des prédicats (J. Herbrand)
- **1965** Principe de résolution (J. A. Robinson)
- **1970** Utiliser la logique comme « langage de programmation »
  clauses de Horn (R. Kowalski)
  Q-systèmes (A. Colmerauer)
- **1972** Langage inventé par A. Colmerauer et P. Roussel à Marseilles et par R. A. Kowalski à Edinburgh
- **1977** Premier compilateur Prolog (D. H. D. Warren) , Université d'Édimbourg
- **1980** Lancement du programme japonais (informatique de la 5ème génération)
- **1990** Prolog évolue vers la « Programmation par Contraintes »

Prolog :- programmation logique.

Bibliographie

- **The Art of Prolog, L. STERLING & E. SHAPIRO, Edts Masson 1990**
- **Prolog, F. GIANNESINI, H. KANOUI, R. PASERO, M. VAN CANEGHEM, InterEditions, 1985**
- **Implementing Prolog – Compiling Prolog Programs 1 & 2, D.H.D. WARREN, DAI Reaserch Repport 39 & 30, University of Edinburgh, 1977**

SWI-Prolog

- **norme Edinburgh,**
- **disponible sur différentes plate formes : LINUX, UNIX et WINDOWS**
- **à l'adresse: http://www.swi.psy.uva.nl**

Différents modes de programmation

programmation impérative

- nécessite l'expression, par le détail, du « comment »
- traduction d'une démarche algorithmique où le passage des données vers les résultats est décrit comme une suite d'actions
- PASCAL, C, ADA

exemple en PASCAL

```
function estPremier(N:integer):boolean;
var i : integer;
      ok, fin:boolean;
begin
if N = 2
   then estPremier := true
   else   begin
          ok := true; fin := false;
          i := 3;
          while not fin do
               if i = N
                        then   fin := true
                        else if (N mod i) = 0
                        then   begin
                                  ok := false;
                                  fin := true
                                  end
                         else    i := i +2
        estPremier := ok
       end
end;
```

Les différents modes de programmation

programmation fonctionnelle

- **description du résultat comme une composition de fonctions agissant sur les objets, relève aussi d'un mode de pensée algorithmique**
- **connue par Lisp, le langage de prédilection des premiers chercheurs en Intelligence Artificielle**
- **LISP, CAML, SCHEME**

exemple en LISP

```
(define estPremier (lambda n)
(if (= n 2)
        true
        (estPremier2 3 n)))

(define estPremier2 (lambda n diviseur)
(if (= n diviseur)
        true
        (if (= (mod n diviseur) 0)
                false
                (estPremier2 n (+ diviseur 2)))))
```

Les différents modes de programmation

programmation « orientée objet »

- ce que l'on fait % à quoi on le fait
- plus qu'une simple technique d'implantation, représente la synthèse idéale de tous les progrès en matière de programmation.
- structure : démarche de conception qui donne la priorité à la description des invariants des processus
- traitement : décentralisation du déclenchement des actions, modules communicants et faiblement couplés
- langage : mélange de techniques déclaratives et procédurales
- SMALLTALK, C++, JAVA

exemple en SMALLTALK

dans la classe Integer

```
isPremier
| i |
(self = 2) ifTrue: [^true];
i := 3;
true whileTrue: [
    (i = self) ifTrue: [^true];
    ((self mod: i) = 0)
            ifTrue: [^false]
            ifFalse[ i := i + 2]].
^true
```

Les différents modes de programmation

programmation logique

- **le langage se charge du « comment »**
- **description du problème à résoudre à partir de l'inventaire des objets concernés et des propriétés et relations qu'ils vérifient**
- **expression formelle de la connaissance qui porte à la fois des éléments implicites et le moyen de les rendre explicites**
- **mécanisme général et universel, moteur intégré au langage qui simule une partie de nos facultés de raisonnement**
- **facteur de non déterminisme qui conduit à entreprendre des actions qui ne conduisent pas forcément au résultat ou peuvent conduire à plusieurs résultats**

exemple en PROLOG

```
premier(2).
premier(N) :- pasDeDiviseur(3,N).
pasDeDiviseur(N,N).
pasDeDiviseur(M,N) :-
        N mod M > 0,
        M2 is M + 2,
        pasDeDiviseur(M2,N).
```

on spécifie les propriétés du résultat du programme et non pas le processus pour arriver à ce résultat (aspect opérationnel)

Intérêts : facilité de compréhension et facilité d'écriture

Prolog :- programmation logique.

Définition

Programmer en PROLOG c'est d'abord décrire un ensemble de clauses à partir desquelles on essaie ensuite de prouver un but.

Hypothèse du « *monde clos* »

Tout ce qui est déductible est considéré comme « *vrai* »

Tout ce qu'on n'arrive pas à déduire est considéré comme « *faux* ».

Le *vrai* et le *faux* n'ont donc pas exactement le sens qu'on leur donne habituellement d'un point de vue logique (habituellement, une expression est fausse si on arrive à prouver qu'elle l'est !!!)

Prolog :- programmation logique.

On utilise Prolog comme un démonstrateur de théorèmes pour clauses de Horn.

Une clause est une formule de la forme $L_1 \vee L_2 \vee \ldots \vee L_n$ où L_i est appelé littéral. Un littéral L_i est dit négatif s'il est de la forme $\neg P_i$, il est positif s'il est de la forme P_i

Une clause de Horn est une clause ayant au plus un littéral positif.

On appelle :

- **fait : une clause ayant un seul littéral (positif)**
- **règle : une clause ayant un littéral positif et au moins un littéral négatif**
- **requête : une clause sans littéral positif**

personne(henri, dupond)

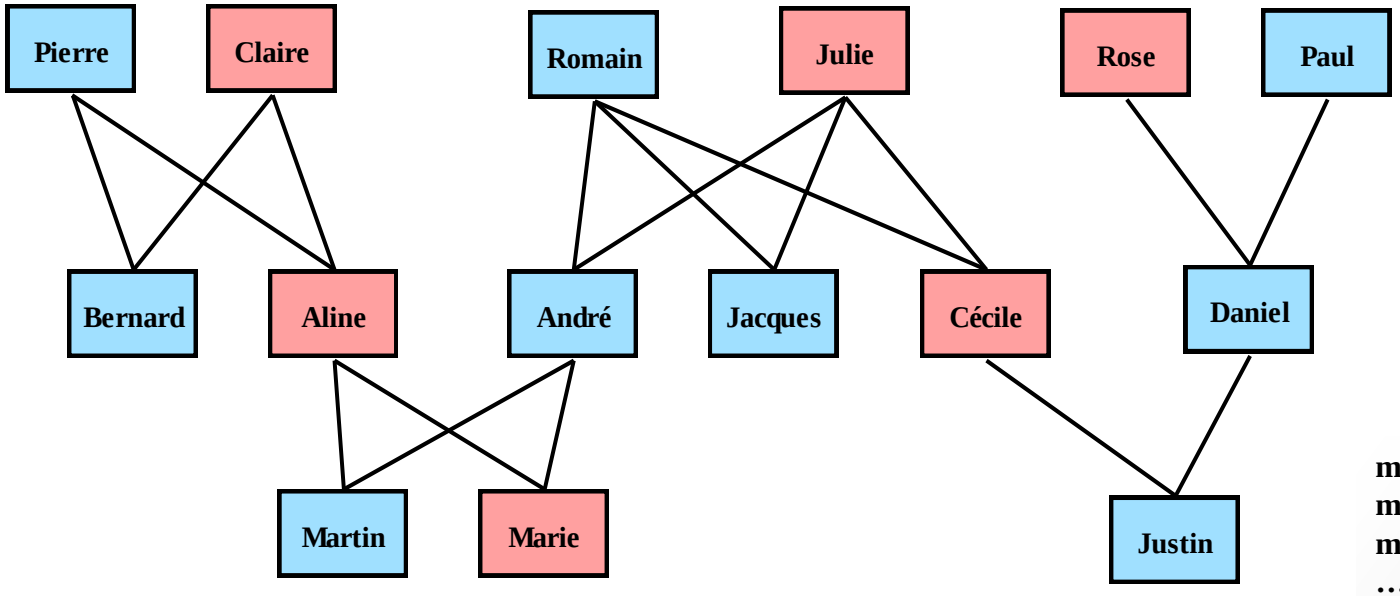
grand-pere(x,y) $\vee \neg$ pere(x,z) $\vee \neg$ parent(z,y)
$\equiv$ pere(x,z) $\wedge$ parent(z,y) $\rightarrow$ grand-pere(x,y)

grand-pere(X,henri).

```
personne(henri, dupond).

grand-pere(x,y) :- pere(x,z), parent(z,y).

grand-pere(X,henri).
```

Prolog :- 1er exemple.

représentation d'une famille




```
masculin(pierre).
masculin(romain).
masculin(paul).
…
feminin(claire).
feminin(julie
…
parent(pierre,bernard).
parent(pierre,aline).
parent(claire,bernard).
…
```

Prolog :- 1er exemple.

```
masculin(pierre).
masculin(romain).
masculin(paul).
masculin(bernard).
masculin(andré).
masculin(jacques).
masculin(daniel).
masculin(martin).
masculin(justin).
feminin(claire).
feminin(julie
feminin(rose).
feminin(aline).
feminin(cecile).
feminin(marie).
parent(pierre,bernard).
parent(pierre,aline).
parent(claire,bernard).
parent(claire,aline).
parent(romain,andre).
parent(romain,jacques).
parent(romain,cecile).
parent(julie,andre).
parent(julie,jacques).
parent(julie,cecile).
parent(rose,daniel).
parent(paul,daniel).
parent(aline,martin).
parent(aline,marie).
parent(andre,martin).
parent(andre,marie).
parent(cecile,justin).
parent(daniel,justin).
```

interrogation de la base:

```
? feminin(rose).
yes
? feminin(marcel).
no
? masculin(X).
X=pierre ;
X=romain ;
X=paul
? parent(X,cecile).
X=romain ;
X=julie ;
no
? parent(rose,X).
X=daniel
yes
?
```

Prolog :- 1er exemple.

```
masculin(pierre).
masculin(romain).
masculin(paul).
masculin(bernard).
masculin(andré).
masculin(jacques).
masculin(daniel).
masculin(martin).
masculin(justin).
feminin(claire).
feminin(julie
feminin(rose).
feminin(aline).
feminin(cecile).
feminin(marie).
parent(pierre,bernard).
parent(pierre,aline).
parent(claire,bernard).
parent(claire,aline).
parent(romain,andre).
parent(romain,jacques).
parent(romain,cecile).
parent(julie,andre).
parent(julie,jacques).
parent(julie,cecile).
parent(rose,daniel).
parent(paul,daniel).
parent(aline,martin).
parent(aline,marie).
parent(andre,martin).
parent(andre,marie).
parent(cecile,justin).
parent(daniel,justin).
```

descriptions de liens de parenté supplémentaires

un individu X est frère d'un autre Y si c'est un garçon et si les 2 individus ont un même parent P.

$\forall X \exists Y$ (frere(X,Y) -> (masculin(X) $\wedge \exists P$ (parent(P,X) $\wedge$ parent(P,Y))))

```
frere(X,Y) :- masculin(X), parent(P,X), parent(P,Y).
```

```
? frere(andre,Y).
Y=jacques ;
Y=cecile;
no
```

$\wedge$ est représenté par ,

$\vee$ est représenté par ;

Prolog :- 1er exemple.

```
masculin(pierre).
masculin(romain).
masculin(paul).
masculin(bernard).
masculin(andré).
masculin(jacques).
masculin(daniel).
masculin(martin).
masculin(justin).
feminin(claire).
feminin(julie
feminin(rose).
feminin(aline).
feminin(cecile).
feminin(marie).
parent(pierre,bernard).
parent(pierre,aline).
parent(claire,bernard).
parent(claire,aline).
parent(romain,andre).
parent(romain,jacques).
parent(romain,cecile).
parent(julie,andre).
parent(julie,jacques).
parent(julie,cecile).
parent(rose,daniel).
parent(paul,daniel).
parent(aline,martin).
parent(aline,marie).
parent(andre,martin).
parent(andre,marie).
parent(cecile,justin).
parent(daniel,justin).
```

```
frere(X,Y) :- masculin(X), parent(P,X), parent(P,Y).
soeur(X,Y) :- feminin(X), parent(P,X), parent(P,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
cousin(X,Y) :- grandparent(Z,X), grandparent(Z,Y).
```

etc…

Prolog :- éléments de bases.

En PROLOG on manipule des termes

- **les constantes**
- **les variables**
- **les structures ou termes composés**

Prolog :- éléments de bases.

les constantes

- **les atomes : ce sont des chaînes alphanumériques qui commencent par une lettre en minuscule.**

  toto

  tOTO

  t123Tx56

- **les nombres**

  19

  -65

  -3.1415

  10E-7

Prolog :- éléments de bases.

les variables

leur identificateurs commencent par une lettre majuscule ou par _ (underscore)

Toto

_tOTO

___T___t___6

_ est une variable muette dont la valeur n'est pas accessible. On l'utilise lorsqu'il n'est pas utile de connaître la valeur d'une variable.

? parent(aline,_).

yes

? parent(aline,X).

X=martin ;

X=marie

yes

Prolog :- éléments de bases.

les structures ou termes composés

Prolog :- faits et règles.

un programme Prolog est un ensemble de faits et de règles

les faits sont de la forme : `fait.`

les règles sont toutes de la forme: `A :- B1, B2, B3, …, Bn.`

`A` est constitue tête de la règle

B_1, B_2, B_3, …, B_n est son corps

la règle `A :- B1, B2, B3, …, Bn.` peut se lire de différentes manières:

- pour prouver A, il faut d'abord prouver B_1 puis prouver B_2 puis … puis prouver B_n.

- pour effacer le but A, il faut effacer le sous-but B_1, … puis effacer le sous-but B_n.

$B_1 \wedge B_2 \wedge \ldots \wedge B_n \rightarrow A \equiv A \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$ (clause de Horn)

```
soeur(X,Y) :- feminin(X), parent(P,X), parent(P,Y).
```

Prolog :- éléments de bases.

les règles doivent être déclarées par paquets.
2 règles font partie du même paquet si elle ont la même tête.
un paquet de règles exprime la possibilité de plusieurs choix possibles (OU)

$$\begin{cases} A :- B_1^1, B_2^1, ..., B_{n_1}^1. \\ A :- B_1^2, B_2^2, ..., B_{n_2}^2. \\ ... \\ A :- B_1^m, B_2^m, ..., B_{n_m}^m. \end{cases}$$

$$\Leftrightarrow$$

$$\left(B_1^1 \wedge B_2^1 \wedge ... \wedge B_{n_1}^1\right) \vee \left(B_1^2, B_2^2, ..., B_{n_2}^2\right) \vee .. \vee \left(B_1^m, B_2^m, ..., B_{n_m}^m\right) \rightarrow A$$

rem: cette équivalence n'est pas exacte car l'ordre des règles est important en Prolog. Les premières règles sont traitées en premier. Alors que d'un point de vue logique on sait que (x ∨ y) ↔ (y ∨ x)

Prolog :- éléments de bases.

```
masculin(pierre).
masculin(romain).
masculin(paul).
masculin(bernard).
masculin(andré).
masculin(jacques).
masculin(daniel).
masculin(martin).
masculin(justin).
feminin(claire).
feminin(julie
feminin(rose).
feminin(aline).
feminin(cecile).
feminin(marie).
```

```
parent(pierre,bernard).
parent(pierre,aline).
parent(claire,bernard).
parent(claire,aline).
parent(romain,andre).
parent(romain,jacques).
parent(romain,cecile).
parent(julie,andre).
parent(julie,jacques).
parent(julie,cecile).
parent(rose,daniel).
parent(paul,daniel).
parent(aline,martin).
parent(aline,marie).
parent(andre,martin).
parent(andre,marie).
parent(cecile,justin).
parent(daniel,justin).
```

Prolog :- éléments de bases.

```
procédure prouver(But: liste d'atomes logiques )
        si But = [ ] alors
           /* le but initial est prouvé : afficher les valeurs des variables du but initial */
        sinon soit But = [B1, B2, .., Bn]
           pour toute clause (A'0 :- A'1, A'2, ..,A'r) du programme:
              (où les variables ont été renommées)
              s :=  plus_grand_unificateur(B1 ,A'0)
              si (s < > échec) alors prouver([s(A'1), s(A'2), .. s(A'r), s(B2), .. s(Bn)], s(But-init}))
              finsi
           finpour
        finsi
    fin prouver
```

Quand on pose une question à l'interprète Prolog, celui-ci exécute dynamiquement l'algorithme prouver. L'arbre constitué de l'ensemble des appels récursifs est appelé arbre de recherche.

Remarques

- stratégie de recherche incomplète (on peut avoir une suite infinie d'appels récursifs)
- stratégie de recherche dépendante de l'ordre de définition des clauses dans un paquet et de l'ordre des atomes logiques dans le corps d'une clause (on prouve les atomes logiques selon leur ordre d'apparition dans la clause).

Prolog :- programmation logique.

remarque : Prolog ne manipule que des clauses de Horn donc certaines connaissances ne peuvent être représentées ni traitées.

exemple : on représente la connaissance « *tout individu X est parent d'un autre individu Y si X est le père de Y ou si X est la mère de Y* » par le programme suivant

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).
```

$\forall X \forall Y(pere(X,Y) \rightarrow parent(X,Y))$
$\forall X \forall Y(mere(X,Y) \rightarrow parent(X,Y))$

$\forall X \forall Y(\neg pere(X,Y) \vee parent(X,Y)) \vee \forall X \forall Y(\neg mere(X,Y) \vee parent(X,Y))$

$\forall X \forall Y(\neg pere(X,Y) \vee parent(X,Y)) \vee (\neg mere(X,Y) \vee parent(X,Y))$

$\forall X \forall Y(\neg pere(X,Y) \wedge \neg mere(X,Y)) \vee parent(X,Y))$

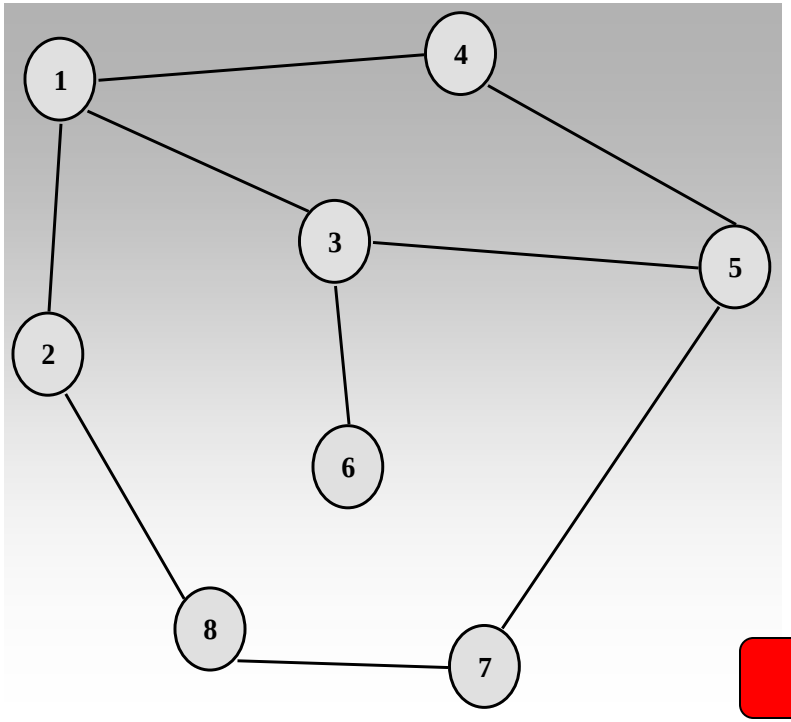
$\forall X \forall Y(\neg (pere(X,Y) \vee mere(X,Y)) \vee parent(X,Y))$

$\forall X \forall Y((pere(X,Y) \vee mere(X,Y)) \rightarrow parent(X,Y))$

logiquement cette connaissance aurait du s'exprimer par :

$\forall X \forall Y((pere(X,Y) \vee mere(X,Y)) \leftrightarrow parent(X,Y))$

Prolog :- 2ème exemple.




```
arc(1,2).
arc(1,3).
arc(1,4).
arc(2,8).
arc(3,6).
arc(3,5).
arc(4,5).
arc(5,7).
arc(7,8).
```

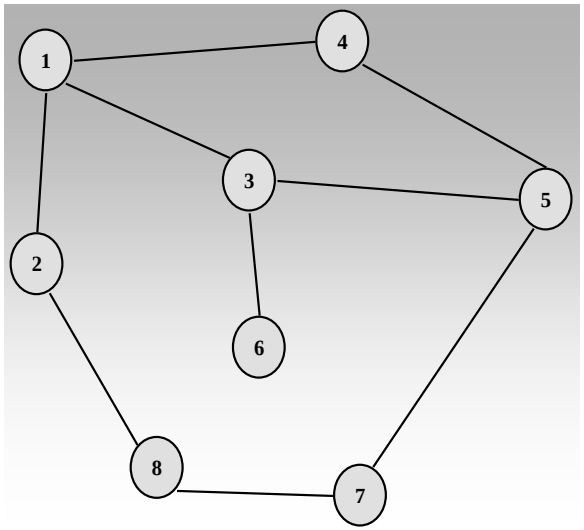
pb de bouclage

```
arc(A,B) :- arc(B,A).
```

dans ce graphe on distingue des nœuds et des liens.
un lien relie 2 nœuds et il est non orienté.
ie qd on peut aller du nœud A vers le nœud B on peut également aller du nœud B vers le nœud A.

```
lien(A,B) :- arc(A,B).
lien(A,B) :- arc(B,A).
```

Prolog :- 2ème exemple.




```
arc(1,2).
arc(1,3).
arc(1,4).
arc(2,8).
arc(3,6).
arc(3,5).
arc(4,5).
arc(5,7).
arc(7,8).
lien(A,B) :- arc(A,B).
lien(A,B) :- arc(B,A).
```

il y a un chemin de A vers B si
il y a une suite de nœuds liés entre-eux
dont le 1er nœud est A et le dernier est B

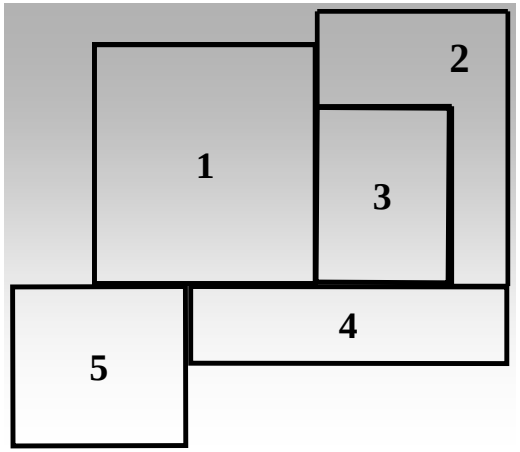
il y a un chemin de A vers B si
il y a un lien entre A et B ou bien
il existe un nœud C lié à A et il y a un chemin de C vers B

$$\text{chemin}(A,B) \Rightarrow \left(\text{lien}(A,B) \vee \exists C \left(\text{lien}(A,C) \wedge \text{chemin}(C,B)\right)\right)$$

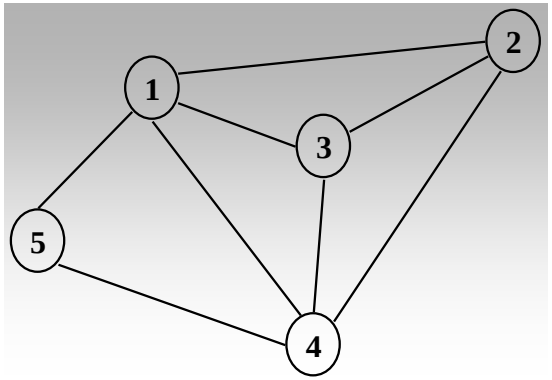
```
chemin(A,B) :- lien(A,B).
chemin(A,B) :- lien(A,C), chemin(C,B).
```

Prolog :- 3ème exemple.

coloriage d'un graphe planaire

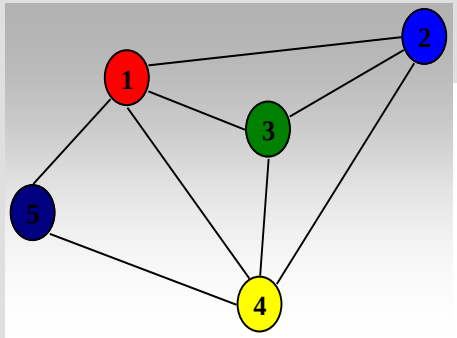


représentation par un graphe d'adjacence



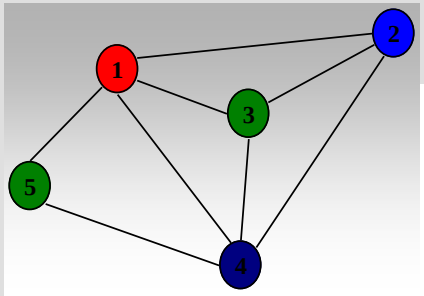

```
arc(1,2).
arc(1,3).
arc(1,4).
arc(1,5).
arc(2,3).
arc(2,4).
arc(3,4).
arc(4,5).

adjacent(A,B) :- arc(A,B).
adjacent(A,B) :- arc(B,A).
```

coloriage a




```
coloriage(1,rouge,a).
coloriage(2,bleu,a).
coloriage(3,vert,a).
coloriage(4,jaune,a).
coloriage(5,bleu,a).
```

coloriage b




```
coloriage(1,rouge,b).
coloriage(2,bleu,b).
coloriage(3,vert,b).
coloriage(4,bleu,b).
coloriage(5,vert,b).
```

Prolog :- 3ème exemple.

```
arc(1,2).
arc(1,3).
arc(1,4).
arc(1,5).
arc(2,3).
arc(2,4).
arc(3,4).
arc(4,5).

adjacent(A,B) :- arc(A,B).
adjacent(A,B) :- arc(B,A).

coloriage(1,rouge,a).
coloriage(2,bleu,a).
coloriage(3,vert,a).
coloriage(4,jaune,a).
coloriage(5,bleu,a).

coloriage(1,rouge,b).
coloriage(2,bleu,b).
coloriage(3,vert,b).
coloriage(4,bleu,b).
coloriage(5,vert,b).
```

```
mauvais_coloriage(UnColoriage) :-
        adajacent(N1,N2),
        coloriage(N1, Couleur, unColoriage),
        coloriage(N2, Couleur, unColoriage).
```

```
? mauvais_coloriage(a).
no
```

```
mauvais_coloriage(Noeud1, Noeud2, UnColoriage) :-
        adajacent(Noeud1, Noeud2),
        coloriage(Noeud1, Couleur, unColoriage),
        coloriage(Noeud2, Couleur, unColoriage).
```

```
? mauvais_coloriage(N1,N2,b).
N1 = 2  N2 = 4
? mauvais_coloriage(N1,N2,b), coloriage(N1, C, b).
N1 = 2  N2 = 4 C = bleu
```

autre problème: ne plus se contenter de reconnaître un graphe correctement coloré mais calculer un coloriage

Prolog :- arithmétique.

une sorte d'affectation : is
à ne pas confondre avec l'identité =

```
? X = 1 + 2.
X = 1+2

? X is 1 + 2.
X = 3
```

les opérateurs usuels existent : +, *, /, mod (il s'agit de symboles de fonctions !!!)

les comparateurs

- > et <
- >= et =<
- =:= égalité numérique
- =\= diségalité numérique

```
?- 1 =:= 2.
No

?- 1 =\= 2.
Yes
```

Prolog :- arithmétique.

définition du PGCD D de X et Y

- **si X et Y sont égaux, D vaut X**
- **si X < Y alors D est le PGCD de X et de Y - X**
- **si Y < X alors échanger le rôle de X et Y**

```
pgcd(X, Y , D) :- X =:= Y, D is X.
```

```
pgcd(X, Y , D) :- X = Y  ,  D is X.
pgcd(X, X , D) :-                D is X.
pgcd(X, X , X).
```

```
pgcd(X, Y , D) :-
        X < Y ,
        pgcd(X, Y-X, D).
```

```
pgcd(X, Y , D) :-
        X < Y ,
        Y1 is Y - X,
        pgcd(X,Y1,D).
```

```
pgcd(X, Y, D) :-
        X > Y,
        pgcd(Y,X,D).
```

Prolog :- arithmétique.

Attention !!!

X = Y	**réussit si X s'unifie avec Y sinon échec**
X is Y	**réussit si Y est une expression arithmétique complètement instanciée à l'appel et X est une variable libre sinon il y a erreur**
X =:= Y	**réussit si X et Y sont deux expressions arithmétiques de même valeur sinon, il y a échec ou erreur selon le cas**
X == Y	**réussit si les termes sont identiques (pas simplement unifiables)**
X \== Y	**réussit si les termes ne sont pas identiques**

Prolog :- les listes.

Définition

Constructeur de liste : |

Une liste peut-être définie de manière récursive par

- **c'est soit la liste vide: [ ]**
- **c'est soit la liste qui se compose d'une tête et d'une queue. Cette queue est elle-même une liste: [Tete | Queue]**
- **la tête est une constante, ou un objet composé, …**

Exemple : construire les listes suivantes à l'aide du constructeur de liste

[1, 2, 3, 4]

[a, b, c, d, [e, f, g, [h, [ ], i]]]

[[[[ ]]]]

[[[ ], [[ ], [ ]]], [ ]]

Prolog :- les listes, exemples.

- **ex1: égalité de 2 listes :** **egales(L1,L2)**
- **ex2: appartenance d'un élément à une liste :** **membre(X,L)**
- **ex3: concaténation de 2 listes :** **concatener(L1,L2,L3)**
- **ex4: suppression d'un élément d'une liste :** **supprimer(-X,-L1,+L2)**
- **ex5: fusion de 2 listes :** **fusionner(-L1,-L2,+L3)**
- **ex6: tri de 2 listes :** **trier(-L1,+L2)**
- **ex7: renverser une liste :** **renverser(-L1,+L2)**

Prolog :- les listes, ex1.

définition du prédicat egales(L1, L2) : egales(L1, L2) ⇔ L1 et L2 sont égales

L1 et L2 sont égales si:

- **elles sont vides toutes les deux**
- **L1 = [X1 | LA] et L2 = [X2 | LB] et X1 = X2 et LA et LB sont égales**

exemple

```
? egales([a,b,c],[a,b,c]).
yes

? egales([a,b],[]).
no
```

version 1

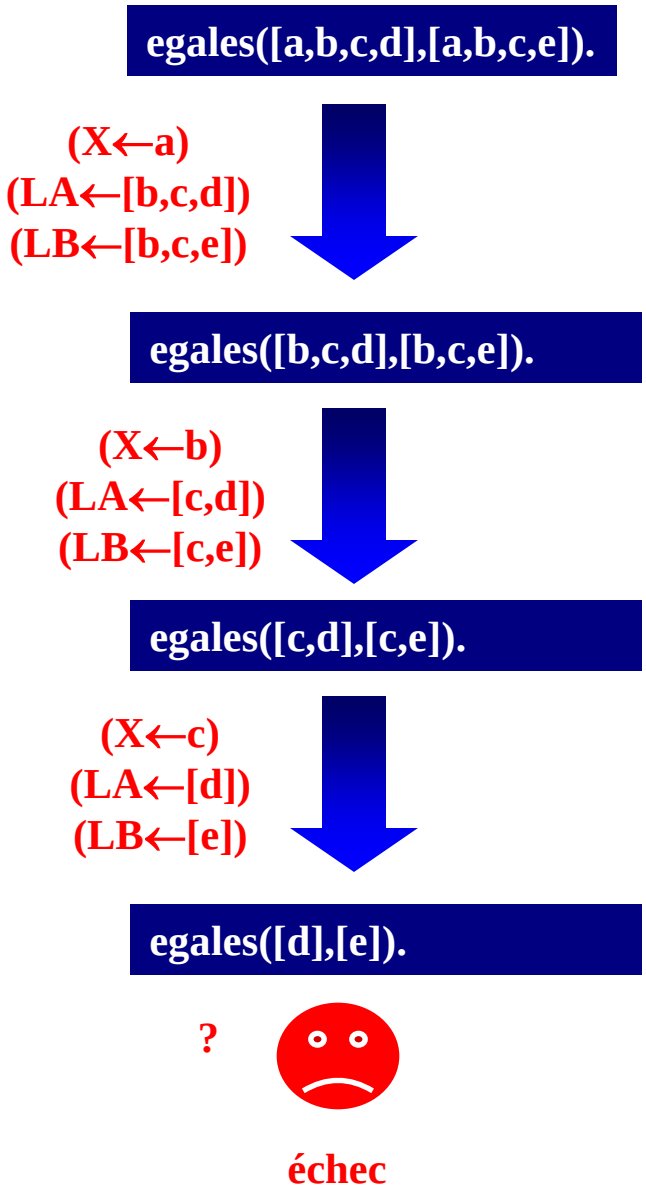
```
egales([],[]).
egales([X1 | LA] , [X2 | LB]) :- X1 = X2, egales(LA, LB).
```

version 2

```
egales([],[]).
egales([X | LA] , [X | LB]) :- egales(LA, LB).
```

Prolog :- les listes, ex1.

```
egales([],[]).
egales([X | LA] , [X | LB]) :- egales(LA, LB).

? egales([a,b,c,d],[a,b,c,e]).
```



Prolog :- les listes, ex2.

définition du prédicat membre(X, L) : membre(X,L) ⇔ X∈L

X est membre de L:

- **X est la tête de la liste L**
- **L de la forme [Y | L'] et X est membre de L'.**

exemple

```
? membre(a, [b,b,a,b,c]).
yes

? membre(a,[ ]).
no
```

cette ligne est correcte mais la compilation fera apparaître un warning:
« `Singleton variable [L]` »

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

le même warning apparaîtra ici:
« `Singleton variable [Y]` »

```
membre(X,[X | _]).
membre(X , [_ | L]) :- membre(X, L).
```

_ est une variable muette

Prolog :- les listes, ex2.

membre(X,[X | L]).
membre(X , [_ | L]) :- membre(X, L).

quels sont les listes L qui ont pour membre 1 ?

? membre(X, [a,b,c,d]).
X=a ;
X=b ;
X=c ;
X=d ;

no

, 1 | _G261] ;
_G260, 1 | _G264] ;

le même programme peut servir pour répondre à différentes requêtes !!!

els sont les éléments X membres des listes L ?

270]

=_G207
L=[_G269, _G207 | _G273]

yes

Prolog :- les listes.

```
membre(X,[X | L]).
membre(X , [_ | L]) :- membre(X, L).
```

pour indiquer à l'utilisateur du prédicat quels sont
les paramètres qui doivent être instantiés on utilise la convention:

- **(+) : le paramètre est une entrée**
- **(-) : le paramètre est une sortie**
- **(?) : le paramètre est une entrée / sortie**

De plus on indique le « type » de ces paramètres

membre(?*Atom* ,?*List*)

membre(+*Atom*,+*List*)

membre(?*Atom* ,+*List*).

Prolog :- les listes, ex3.

définition du prédicat concatener(L1, L2,L3)

concatener(L1, L2,L3) ⇔ L3 est la concaténée de L1 et L2

exemple

```
? concatener([a,b,c],[d,e],L).
L=[a,b,c,d,e]
yes
```

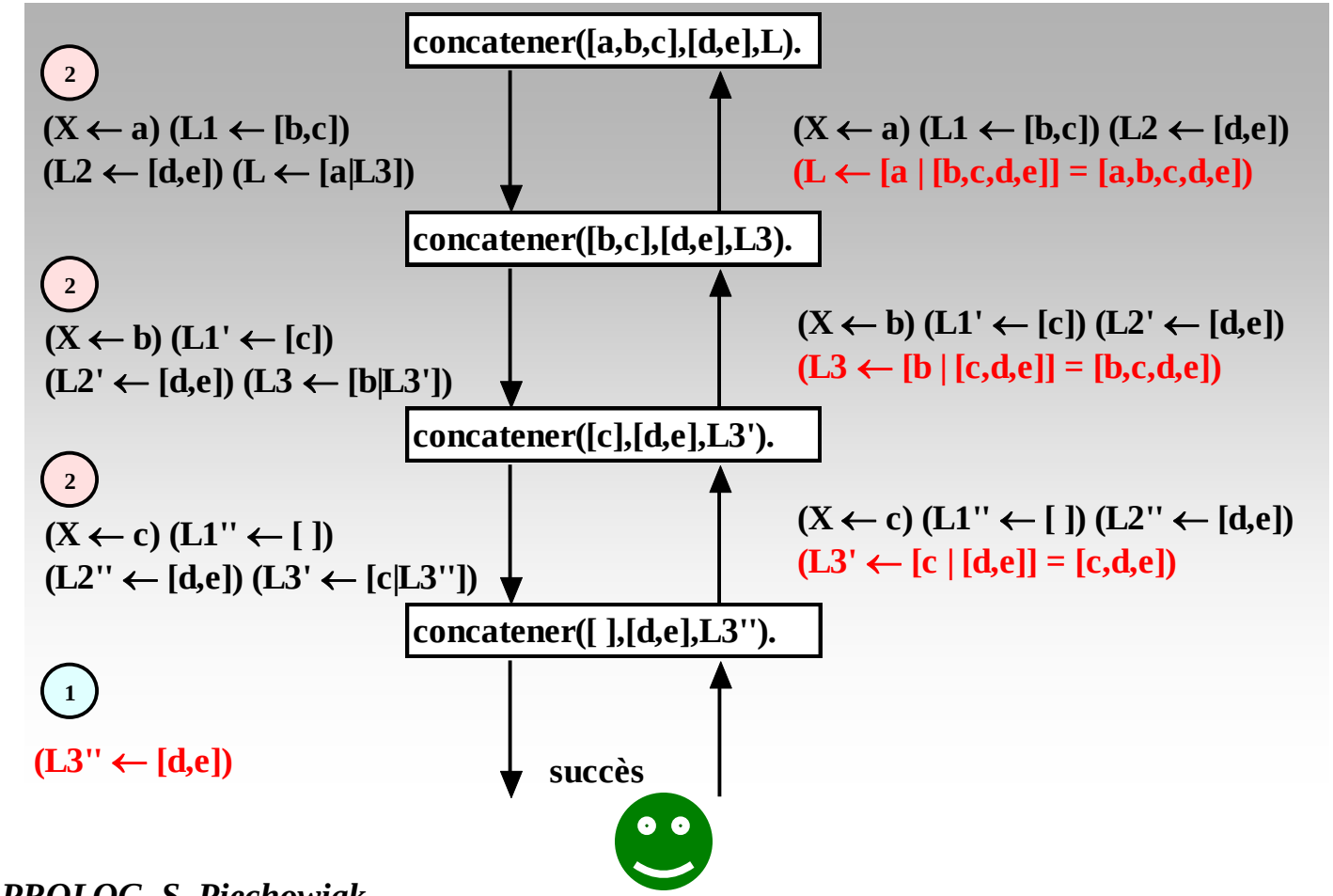
L3 est la concaténée de L1 et L2 si elle est composée des éléments de L1 puis des éléments de L2
si L1 est de la forme [X | L1']
alors L3 doit être de la forme [X | L3']
où L3' est la concaténée de L1' et de L2
si L1 est vide
alors la concaténeé de [ ] et L2 c'est L2

```
concatener([ ],L,L).
concatener([X|L1], L2 ,[X|L3]) :- concatener(L1,L2,L3).
```

Prolog :- les listes, ex3.

(1) concatener([],L,L).
(2) concatener([X|L1], L2 ,[X|L3]) :- concatener(L1,L2,L3).

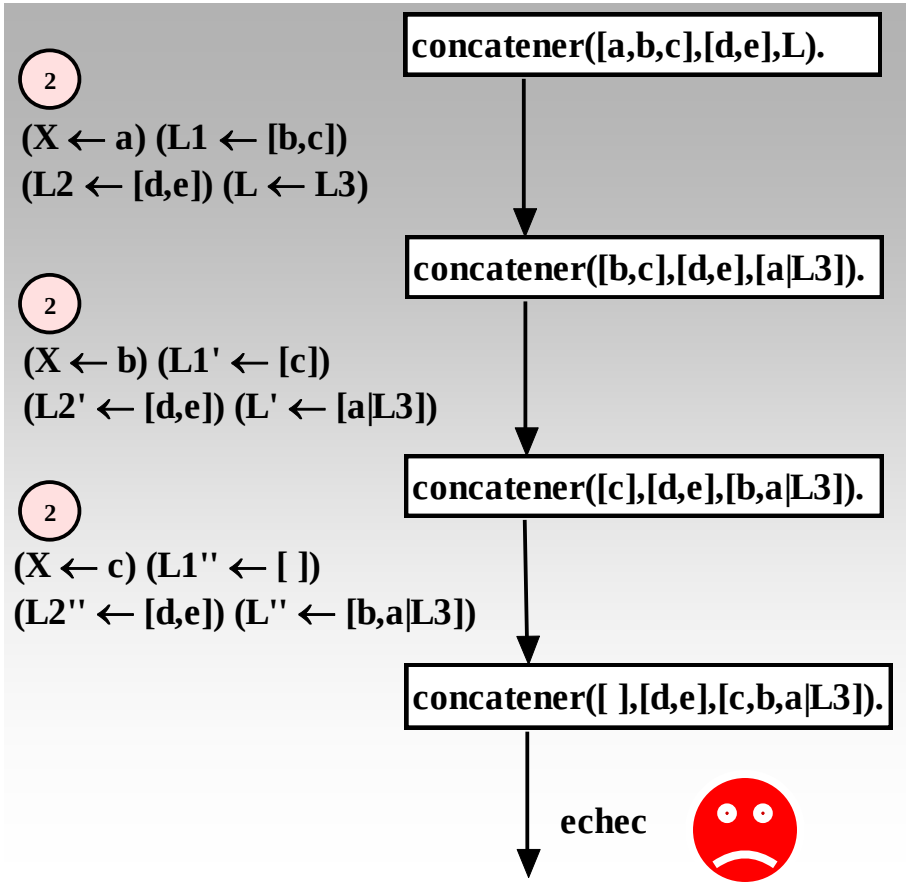
concatener([a,b,c],[d,e],L).



Prolog :- les listes, ex3.

que se passe-t-il avec cette 2ème version ?

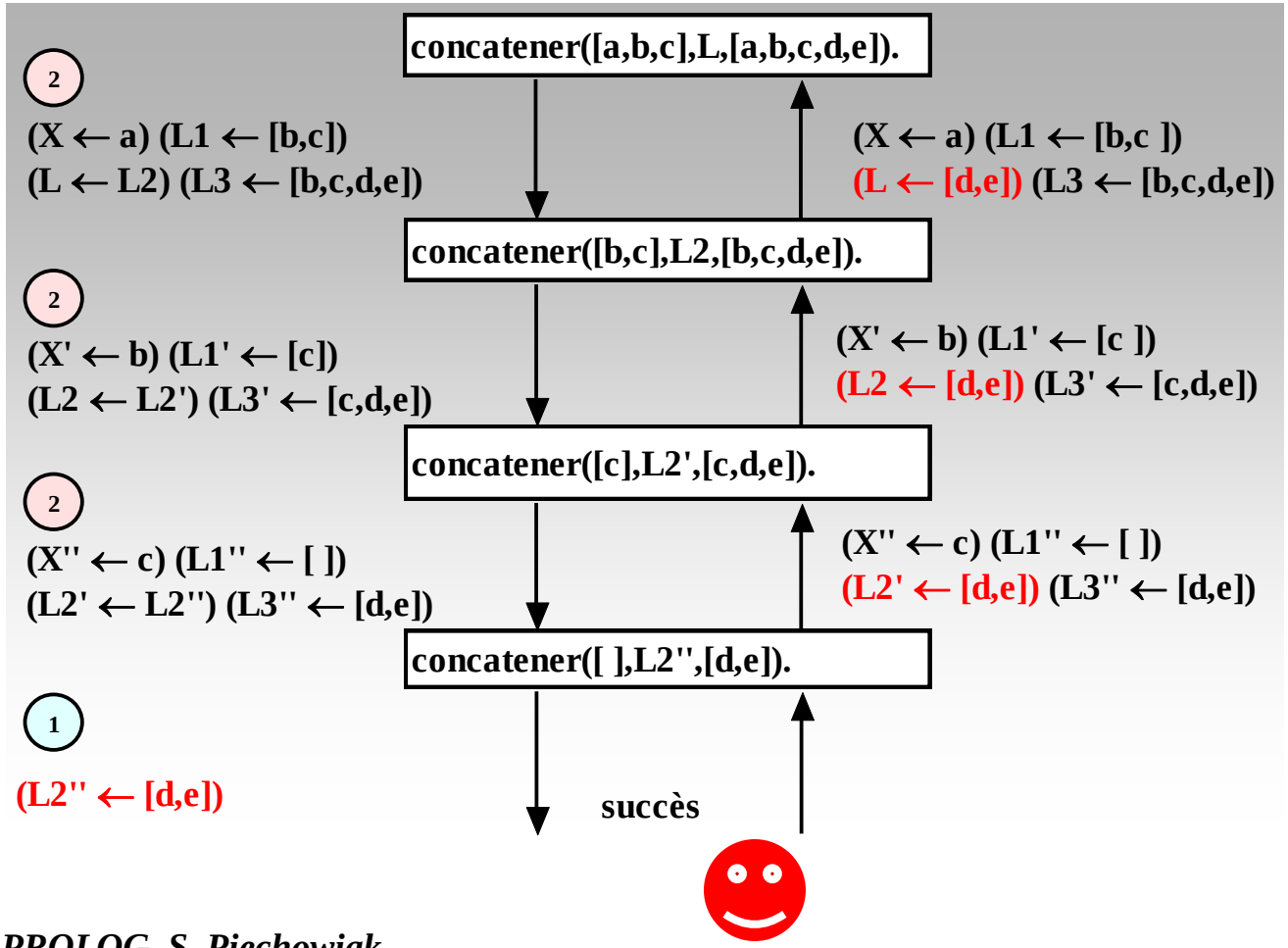
(1) concatener([],L,L).
(2) concatener([X|L1], L2 ,L3) :- concatener(L1,L2,[X|L3]).



Prolog :- les listes, ex3.

(1) concatener([],L,L).
(2) concatener([X|L1], L2 ,[X|L3]) :- concatener(L1,L2,L3).

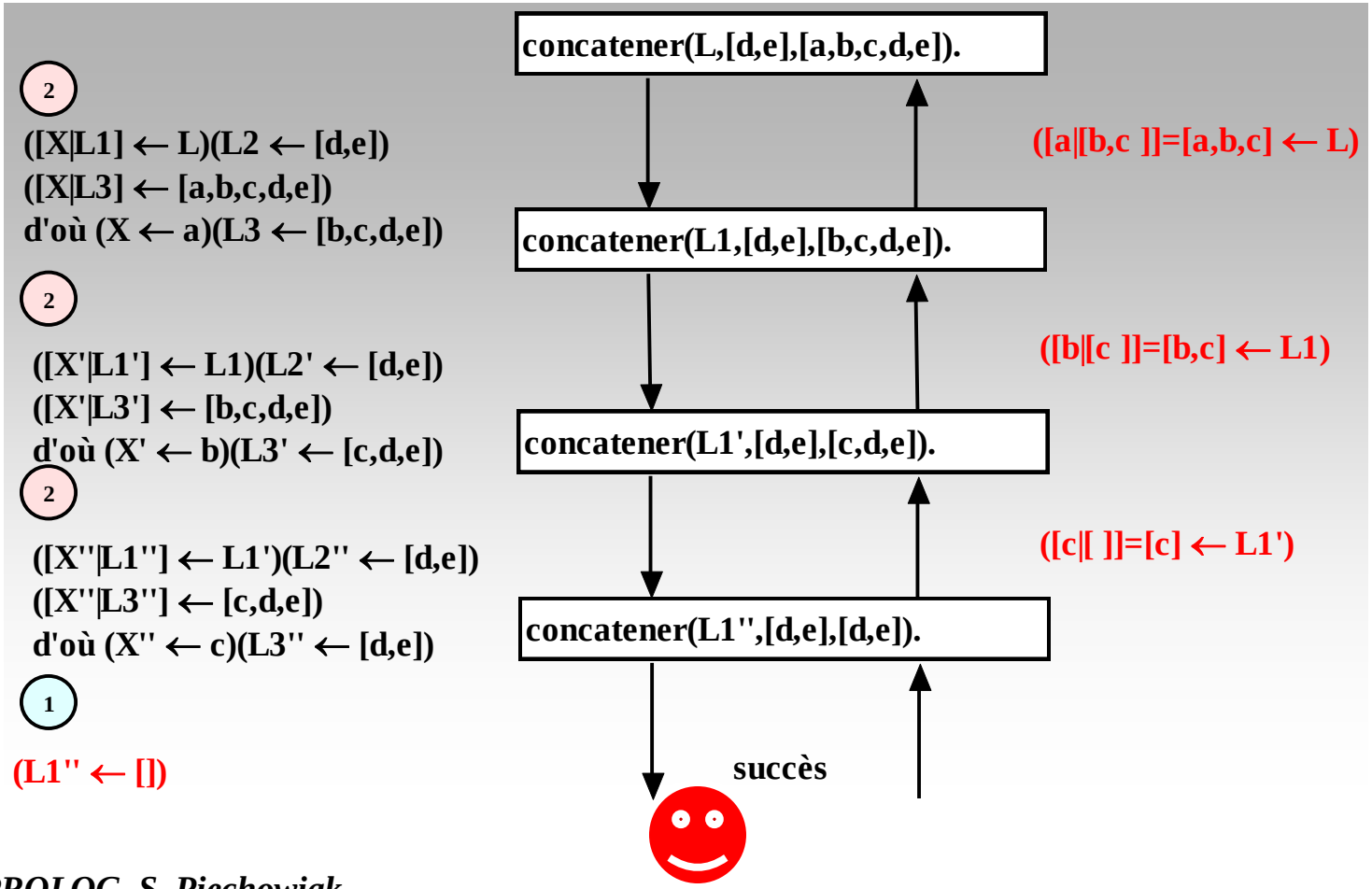
concatener([a,b,c],L,[a,b,c,d,e]).



Prolog :- les listes, ex3.

(1) concatener([],L,L).
(2) concatener([X|L1], L2 ,[X|L3]) :- concatener(L1,L2,L3).

concatener(L,[d,e],[a,b,c,d,e]).



Prolog :- les listes, ex4.

définition du prédicat supprimer(X, L1,L2)

supprimer(X, L1,L2) ⇔ L2 est la liste qui contient tous les éléments de L1 (dans le même ordre) sauf les occurrences de X

exemple

```
? supprimer(a,[a,b,a,a,a,d,e],L).
L=[b,d,e]

yes
```

si L1 est vide alors L2 l'est également

```
supprimer(X, [ ], [ ]).
```

si L1 est de la forme [X |L1']
alors L2 est égale à L1' de laquelle on retire les occurrences de X

```
supprimer(X, [ X | L1 ], L2) :- supprimer(X, L1, L2).
```

si L1 est de la forme [Y |L1']
alors L2 est égale à [Y | L1'] ou L1' est égale à L1 de laquelle on retire toutes les occurrences de X

```
supprimer(X, [ Y | L1 ], [ Y | L2] ) :- supprimer(X, L1, L2).
```

Prolog :- les listes, ex5.

définition du prédicat fusionner(L1, L2,L3)

fusionner(L1, L2,L3) ⇔ L3 comporte tous les éléments de L1 et de L2 en respectant l'ordre. la fusion n'a de sens que si L1 et L2 contiennent des éléments comparables et qu'elles sont dans le même ordre !

exemple

```
? fusionner([1,5,7,19],[2,6,14],L).
L=[1,2,5,6,7,14,19]

yes
```

si la liste L1 est vide, la fusion de [ ] avec L2 est L2
si la liste L2 est vide, la fusion de L1 avec [ ] est L1
si L1 et L2 ne sont pas vides, elles sont de la forme [X1 | L1'] et [X2 | L2'].
si X1 > X2, la fusion de L1 et L2 est de la forme [X2 | L3']
où L3' est la fusion de [X1 | L1'] avec L2'
sinon (X1 ≤ X2), la fusion de L1 et L2 est de la forme [X1 | L3']
où L3' est la fusion de L1' avec [X2 | L2']

```
fusion([],L,L).
fusion(L,[],L).
fusion([X1|L1],[X2|L2],[X1|L3]) :- X1 =< X2, fusion(L1,[X2|L2],L3).
fusion([X1|L1],[X2|L2],[X2|L3]) :- X1 > X2,  fusion([X1|L1],L2,L3).
```

Prolog :- les listes, ex6.

définition du prédicat trier(L1, L2)

trier(L1, L2) ⇔ L2 est composée des éléments de L1 triés

exemple

```
? trier([1,7,2,6],L).
L=[1,2,6,7]

yes
```

si L1 est vide, elle est triée: L2 = [ ].
si L1 n'est pas vide elle est de la forme [X | L1'].
On peut répartir tous les éléments de L1' en 2 listes LA et LB. Dans LA on place tous les éléments inférieurs à X et dans LB ceux qui sont supérieurs à X. La fusion de LA triée avec [X] avec LB triée est la liste recherchée.

```
trier([ ],[ ]).
trier([X|L], Resultat) :-
     eclater(X,L,LA,LB),
     trier(LA,LA2),
     trier(LB,LB2),
     fusion(LA2,[X | LB2],Resultat).
```

```
eclater(_,[ ],[ ],[ ]).
eclater(X,[Y|L],[Y|L1],L2) :-
     Y =< X,
     eclater(X,L,L1,L2).
eclater(X,[Y|L],L1,[Y|L2]) :-
     Y > X,
     eclater(X,L,L1,L2).
```

Prolog :- les listes, ex7.

définition du prédicat renverser(L1, L2)

renverser(L1, L2) ⇔ L3 est composée des éléments de L1 dans l'ordre inverse

exemple

```
? renverser([a,b,c],L).
L=[c,b,a]

yes
```

version 1

si L1 est vide, sa renversée est la liste vide
si la liste L1 est de la forme [X | L1'], sa renversée est la concaténée de la renversée de L1' avec la liste [X] (on ajoute X en fin de liste)

```
renverser([ ], [ ]).
renverser([X | L1], L2) :- renverser(L1, L3), concatener(Y3,[X], L2.YY
```

le programme est correct mais inefficace du fait de l'utilisation du prédicat concaténer pour ajouter un élément à la fin de la liste.

Prolog :- les listes, ex7.

version 2

on va utiliser un prédicat d'arité 3 dont l'un des arguments va être utilisé comme une pile

On remarque que des prédicats différents peuvent avoir le même nom. La distinction se fait par rapport au nombre des arguments.

2 arguments **3 arguments**

```
renverser(L1,L2) :- renverser(L1, [ ], L2).

renverser([ X | LX ], Pile, LZ ) :- renverser(LX, [X | Pile], LZ).
renverser([ ], LZ, LZ).
```

Prolog :- les listes, un prédicat bien utile.

On définit le prédicat selectionner(, ,) :

```
selectionner(X, [X|L], L).
selectionner(X, [Y|LY], [Y|LZ]) :- selectionner(X,LY,LZ).
```

qu'obtiendra-t-on avec les requêtes suivantes :

```
? selectionner(A, [1,2,3], B).

? selectionner(1, [1,2,3], B).

? selectionner(A, [1,2,3], [3]).

? selectionner(1,L,[2,3]).

? selectionner(1,L,L2).

? selectionner(A,L,[2,3]).

? selectionner(A,L,L2).
```

Pour la 1ère requête, donner l'arbre de dérivation.

Prolog :- les listes, un prédicat bien utile.

```
?- selectionner(A,[1,2,3],B).

A = 1
B = [2, 3] ;

A = 2
B = [1, 3] ;

A = 3
B = [1, 2] ;

No
?- selectionner(1,[1,2,3],B).

B = [2, 3] ;

No
?- selectionner(A,[1,2,3],[3]).


No
?- selectionner(1,L,[2,3]).

L = [1, 2, 3] ;

L = [2, 1, 3] ;

L = [2, 3, 1] ;

No
```

```
?- selectionner(1,L,L2).

L = [1|_G319]
L2 = _G319 ;

L = [_G384, 1|_G388]
L2 = [_G384|_G388] ;

L = [_G384, _G390, 1|_G394]
L2 = [_G384, _G390|_G394]

Yes
?- selectionner(A,L,[2,3]).

A = _G341
L = [_G341, 2, 3] ;

A = _G341
L = [2, _G341, 3] ;

A = _G341
L = [2, 3, _G341]

Yes
?- selectionner(A,L,L2).

A = _G317
L = [_G317|_G319]
L2 = _G319 ;

A = _G317
L = [_G399, _G317|_G403]
L2 = [_G399|_G403] ;

A = _G317
L = [_G399, _G405, _G317|_G409]
L2 = [_G399, _G405|_G409]

Yes
```

Prolog :- les listes, exercices.

On souhaite représenter des ensembles par des listes …

- **exo1: appartenance d'un élément à un ensemble**
- **exo2: union de 2 ensembles**
- **exo3: intersection de 2 ensembles**
- **exo4: différence de 2 ensembles**
- **exo5: partitions d'un ensemble**
- **exo6: inclusion d'un ensemble dans un autre**

On souhaite calculer les permutations d'une liste …

Prolog :- les listes, exercices, une solution.

```
inter([],_,[]).
inter([X|L1],L2,[X|L3]) :- member(X,L2), !, inter(L1,L2,L3).
inter([_|L1],L2,L3) :- inter(L1,L2,L3).

union([],L,L).
union([X|L1],L2,L3) :- member(X,L2), !, union(L1,L2,L3).
union([X|L1],L2,[X|L3]) :- union(L1,L2,L3).
```

Prolog :- prédicats de contrôle.

Autres prédicats de contrôle

- `true` est un but qui réussit toujours

```
p(a,b).  ≡ p(a,b) :- true.
```

- `fail` est un but qui échoue toujours (on peut le définir simplement: il suffit d'utiliser un prédicat non défini donc qui ne peut être prouvé)

- `call(X)` est un méta but. Il considère X comme un but et essaie de le résoudre.

```
?- Y=b, X=member(Y, [a,b,c]), call(X).
Yes
```

Prolog :- le « CUT ».

Le CUT ou coupure ou coupe-choix (!) permet de réduire dynamiquement l'arbre de recherche des calculs.

Intérêt

Si la branche coupée correspond à une partie de l'espace de recherche qui ne comporte aucune solution, l'utilisation du CUT permet de réduire l'espace de recherche sans perte de solution: on parle de « coupure verte ».

Danger

Si la branche coupée correspond à une partie de l'espace de recherche qui comporte des solutions, l'utilisation du CUT conduit à des pertes de solutions: on parle de « coupure rouge ».

Prolog :- le « CUT ».

exemple

1. **nb (X,[ ],0).**
2. **nb(X,[X|L],N) :- nb(X,L,NL), N is 1 + NL.**
3. **nb(X,[Y|L],N) :- nb(X,L,N).**

1. **nb (X,[ ],0).**
2. **nb(X,[X|L],N) :- nb(X,L,NL), !, N is 1 + NL.**
3. **nb(X,[Y|L],N) :- nb(X,L,N).**

Prolog :- le « CUT ».

ex de coupure verte

Si la branche coupée correspond à une partie de l'espace de recherche qui ne comporte aucune solution, l'utilisation du CUT permet de réduire l'espace de recherche sans perte de solution: on parle de coupure verte.

ex de coupure rouge

```
minimum(X,Y,X) :- X =< Y, ! .
minimum(X,Y,Y).
```

```
? minimum(1,5,A).
A=1

yes
? minimum(1,5,5).

yes
```

si X est inférieur à Y alors le min est X sinon, il n'est pas nécessaire de comparer X à Y, et on conclue que le min est Y

Prolog :- le « CUT ».

ex de coupure rouge

```
minimum(X,Y,X) :- X =< Y, ! .
minimum(X,Y,Y).
```

```
? minimum(1,5,A).
A=1

yes
? minimum(1,5,5).

yes
```

si X est inférieur à Y alors le min est X sinon,
il n'est pas nécessaire de comparer X à Y,
et on conclue que le min est Y

```
minimum(X,Y,Z) :- X =< Y, !, X = Z .
minimum(X,Y,Y).
```

le prédicat minimum défini ne correspond pas à la définition mathématique du min !!!

Prolog :- le « CUT ».

Exercice: on défini le prédicat `max` de 5 façons différentes:

1
```
max(A,B,C) :- A >= B, C=A.
max(A,B,C) :- A < B, C=B.
```

2
```
max(A,B,A) :- A >= B.
max(A,B,B) :- B > A.
```

3
```
max(A,B,A) :- A >= B, ! .
max(A,B,B) :- B > A.
```

4
```
max(A,B,A) :- A >= B, ! .
max(A,B,B).
```

5
```
max(A,B,C) :- A >= B, !, C=A .
max(A,B,B).
```

que donnent, pour chaque prédicat les requêtes:

```
max(1,2,X).
max(2,1,X).
max(2,1,1).
max(1,2,2).
```

Prolog :- le « CUT ».

ex: la « négation par l'échec »

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

```
? minimum(1,5,A).
A=1

yes
? minimum(1,5,5).

yes
```

En résumé, les utilités du coupe-choix sont :

* **éliminer les points de choix menant à des échecs certains**
* **supprimer certains tests d'exclusion mutuelle dans les clauses**
* **permettre de n'obtenir que la première solution de la démonstration**
* **assurer la terminaison de certains programmes**
* **contrôler et diriger la démonstration**

Les dangers du coupe-choix sont :

* **supprimer des choix conduisant à des solutions**

Prolog :- la négation.

il y a qqs pbs avec la négation

il faut se souvenir que Prolog considère comme faux tout ce qu'il ne sait pas prouver

```
r(a).
q(b).
p(X) :- not( r(X) ).
```

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

```
?- q(X), p(X).
X = b

yes

?- p(X), q(X).

no
```

Prolog :- le monde clos.

`not` s'écrit également `\+`

- **not(X)**
  - **ne veut pas dire**
    - ***X est toujours faux***
  - **veut simplement dire**
    - ***Je n'ai pas assez d'information pour prouver X***

- **Prolog considère ce qui n'est pas vrai comme faux et vice-versa**
  - **c'est la théorie du monde clos**

- **A quoi peut servir : not(not(P)) ?**

Prolog :- Quelques outils/exemples

Prolog dispose de prédicats bien pratiques …

si +Condition alors +Action1 sinon +Action2 `:+Condition -> +Action1 ; +Action2`

```
If -> Then; _ :- call(If), !, call(Then).
If -> _; Else :- !, call(Else).

If -> Then :- call(If), !, call(Then).
```

Définir le prédicat repeat

Prolog :- accès aux clauses.

le prédicat prédéfini clause/2 permet d'examiner les clauses d'un programme

toute clause est un terme instance de :- (X,Y).
où X représente la tête de la clause et Y est son corps.

Prolog :- structuration des connaissances.

notion de foncteur permet de structurer la connaissance

```
individu( identite(dupont,paul,naissance(18,04,1963)),
          adresse(126,victor hugo,59000,lille),
          tel(0320567890),
          etc …).
individu( identite(dion,celine,naissance(10,11,1961)),
          adresse(123,soleil,62000,arras),
          tel(0123456789),
          etc …).
etc.
```

il est toujours possible d'utiliser des listes (de listes …)

```
[ [ [dupont,paul,[18,04,1963]] ,[126,victor hugo,59000,lille], 0320567890, etc …],
  [ [dion,celine,[10,11,1961]], [123,soleil,62000,arras], 0123456789, etc …],
  etc.
]
```

Prolog :- structuration des connaissances.

individu(identite(dupont,paul,naissance(18,04,1963)),
adresse(126,victor hugo,59000,lille),
tel(0320567890)).
individu(identite(dion,celine,naissance(10,11,1961)),
adresse(123,soleil,62000,arras),
tel(0123456789)).
etc.

quel est le numéro de tel de céline dion ?

```
?- individu(
    identite(dion,celine,_),
              _,
             tel(X)).

X= 0123456789

yes
```

inutile de considérer la date de naissance

inutile de considérer l'adresse

Prolog :- structuration des connaissances.

- **Consultation de termes structurés**
  - **functor /3**
    ?- functor(date(9, janvier, 1973), F, A)
    F = date, A = 3
  - **arg /3 : quel est le nième argument**
    ?- arg(3, date(9, janvier, 1973), F)
    F = 1973

- **Construction/déconstruction de termes structurés**
  - **=.. /2**
    ?- X =.. [date, 9, janvier, 1973]
    X = date(9, janvier, 1973)

Prolog :- structuration des connaissances.

définition de nouveaux opérateurs :
il s'agit d'une définition syntaxique qui facilite l'écriture de termes.

:- op(80, fy, non).
:- op(100, yfx, et).

non a et b est devenu un terme valide, il est équivalent à et(non(a), b)

---

précédence des opérateurs :

chaque opérateur possède une précédence (1..1200)

exemple :

+ a une plus forte précédence que /

a+b/c se lit a+(b/c)

---

résolution des ambiguïtés :

xfx opérateurs infixes non associatifs : les deux sous-expressions ont un niveau de précédence inférieur à celui de l'opérateur

xfy opérateurs associatifs à droite : seule l'expression de gauche doit avoir un niveau inférieur à l'opérateur

yfx opérateurs associatifs à gauche

Prolog :- « Typage » en Prolog.

il est possible de connaître la nature des entités manipulées grâce à quelques prédicats prédéfinis

- **var/1, nonvar/1**
- **integer/1, float/1, number/1**
- **atom/1, string/1, atomic/1**
- **compound/1**
- **ground/1**

si X est une variable et N un entier, on construit une liste de longueur N
si X est une liste, on calcule sa longueur

```
longueur(X,N) :- var(X), longueur1(X,N).
longueur(X,N) :- nonvar(X), longueur2(X,N).

longueur1([ ], 0).
longueur1([X | L], N) :- N > 0,
                         M is N - 1,
                         longueur1(L, M)

longueur2([ ], 0).
longueur2([X | L], N) :- longueur2(L,M), N is M + 1.
```

Prolog :- entrées/sorties.

en mode interactif, user (la console) désigne le fichier d'entrée et de sortie

see(+X) : X devient l'entrée.
seen : fermeture du flux d'entrée et user redevient le nouveau flux d'entrée.
seeing(?X) : unification de X avec le nom du flux d'entrée courant.

tell(+X) : X devient la sortie.
told : fermeture du flux de sortie et user redevient le nouveau flux d'entrée.
telling(?X) : unification de X avec le nom du flux de sortie courant.

append(+File) : similaire à `tell/1`, mais le pointeur de position se place en fin de fichier.

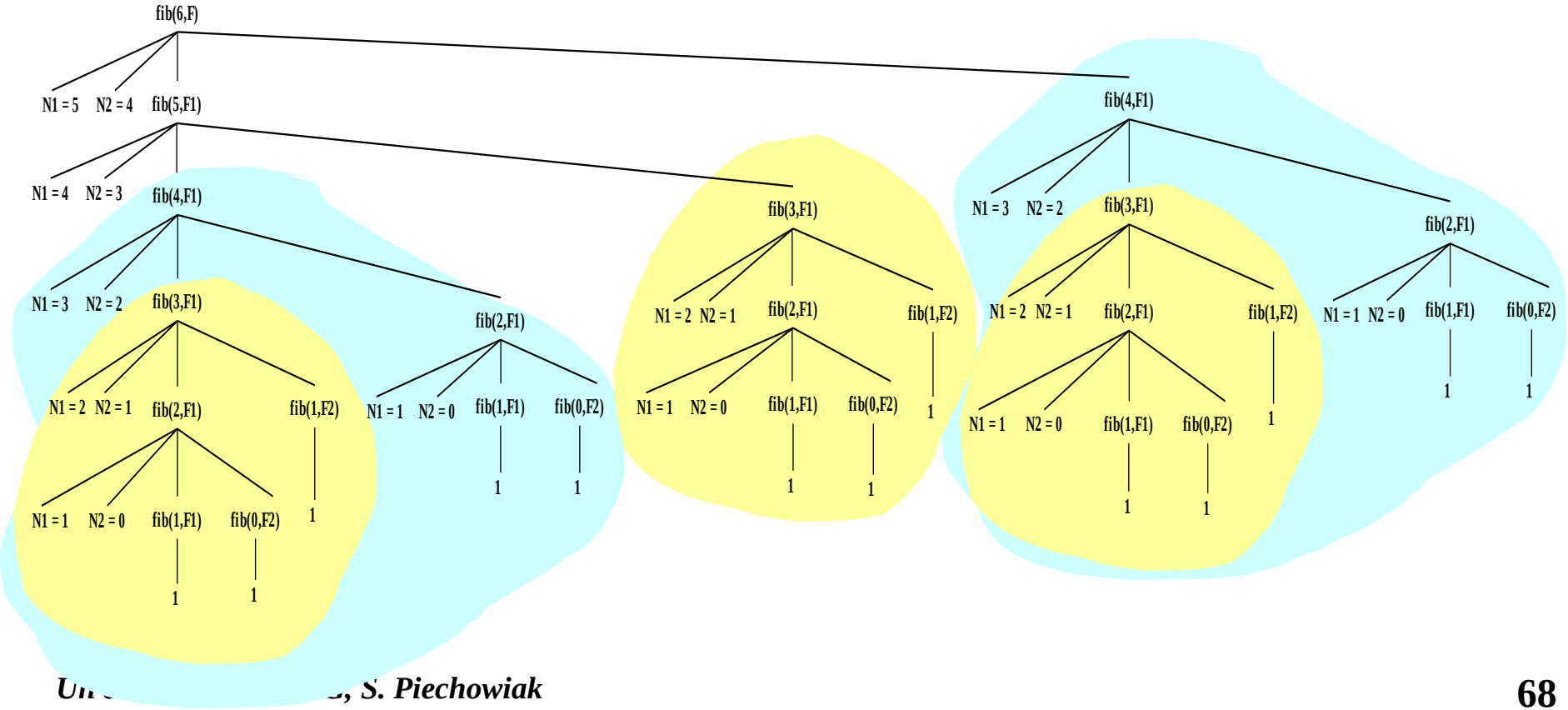
```
lire(File) :-    seeing(Old), see(File),
                      repeat,
                      read(Data),
                      process(Data),
                 seen, see(Old), !.

process(end-of-file)  :- !, fail.
process(Data) :- write_ln(Data), assertz(Data), fail.
```

```
?- lire('pbs-prolog\mon_fichier.txt').

yes
```

lecture des clauses depuis un fichier

Ajouter/Retirer des faits

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
     N1 is N - 1,
     N2 is N - 2,
     fib(N1, F1),
     fib(N2, F2),
     F is F1 + F2.
```

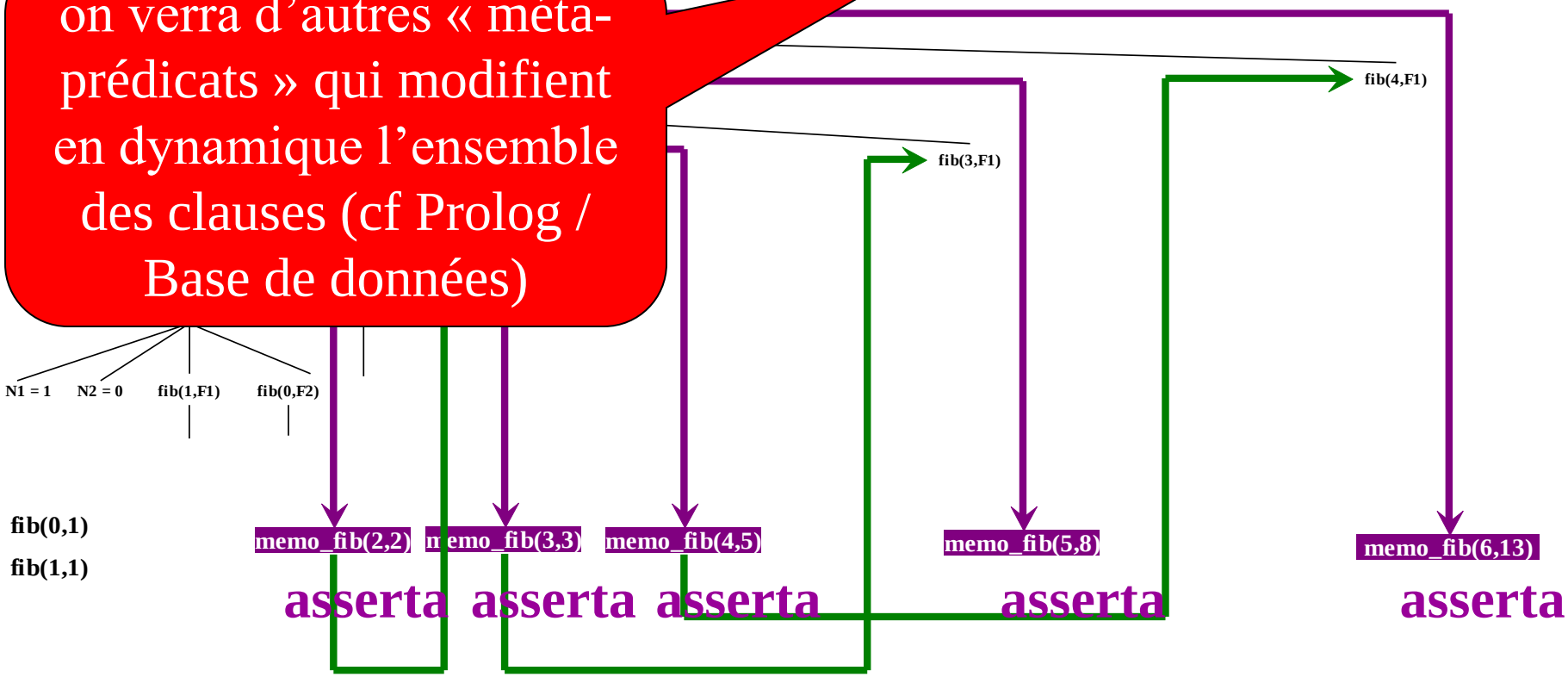


Ajouter/Retirer des faits

faiblesse: l'ajout « dynamique » de clauses est coûteux.

```
fib(N, F) :- memo_fib(N, F).
fib(0, 1).
fib(1, 1).
fib(N, F) :-
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2,
    asserta(memo_fib(N,F)).
```

on verra d'autres « méta-prédicats » qui modifient en dynamique l'ensemble des clauses (cf Prolog / Base de données)



Ajouter/Retirer des faits

**L'ajout « dynamique » de clauses est coûteux,
donc il faut eviter de l'utiliser**

Une autre solution consiste à mémoriser les solutions intermédiaires dans une liste annexe …

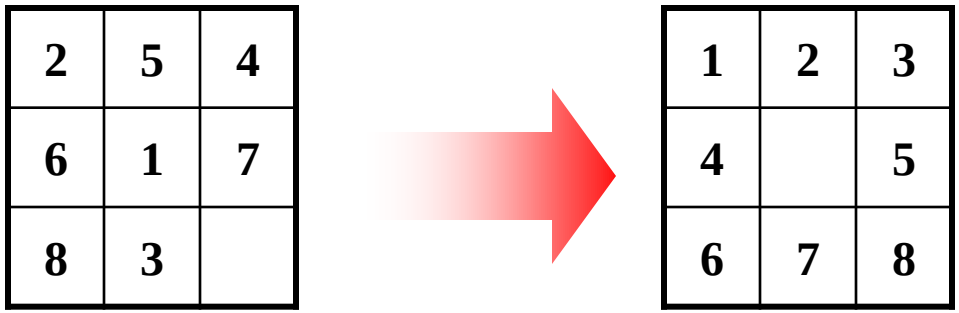
```
fib(N, F) :- fib(N, F, [ done(0,1), done(1,1) ].

fib( N, F, L, L) :- member(done(N,F), L).
fib(N, F, [done(N,F) | L2] ) :-
    N1 is N – 1,
    N2 is N - 2,
    fib(N1, F1, L, L1),
    fib(N2, F2, L1,L2),
    F is F1 + F2.
```

faiblesse: la liste est reconstruite à chaque utilisation du programme.

Le taquin

Il s'agit de d'ordonner un ensemble de nombres en respectant les déplacements autorisés.

Par exemple :



Le seul mouvement autorisé est de déplacé un nombre d'une case adjacente à la case vide vers cette case vide.

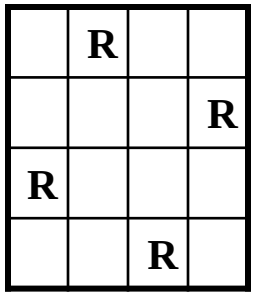
Pbs:

1. **Comment représenter un taquin ?**
2. **Comment représenter un état ?**
3. **Comment représenter un mouvement autorisé ?**
4. **Généralisation ?**

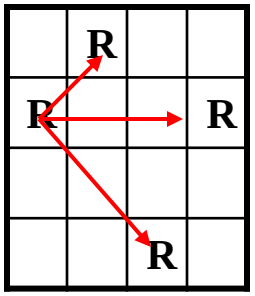
Les 8 reines

Pb: placer 8 reines sur un échiquier (8×8) de manière qu'aucune ne soit attaquée.

Exemples (ave 4 reines) :



est une solution



n'est pas une solution

On peut, par exemple, représenter les 8 reines par une liste

Rassembler tous les buts

pb: lorsqu'un problème possède plusieurs solutions, comment conserver toutes ces solutions ?

solution1 : construire une liste dans laquelle sont placées les solutions
solution2 : utiliser le prédicat `findall(+Var, +Goal, -Bag)`

```
permuter([],[]).
permuter([X|L1],L2) :-
        permuter(L1,L3),
        selectionner(X,L2,L3).

permutations(L,LResultat) :-
        findall(UnePermuation, permuter(L,UnePermutation), LResultat).
```

Database

abolish(+Functor,+Arity)	**: réussit si Term est une variable libre**
retract(+Term)	**: réussit si Term n'est pas une variable libre**
retractall(+Term)	**: réussit si Term est lié à un entier**
assert(+Term)	**: réussit si Term est lié à un réel**
asserta(+Term)	**: réussit si Term est lié à un entier ou un réel**
assertz(+Term)	**: réussit si Term est lié à un atom**