PROLOG

Prolog

- Langage conçu en 1973 par l'équipe d'Alain Colmerauer
- Un langage fondé sur la logique des prédicats du premier ordre restreintes aux clauses de Horn
- Un langage déclaratif
- Une mise en oeuvre de la réfutation
- Environnement utilisé SWI-Prolog (Université d'Amsterdam)

http://www.swi-prolog.org/

Les clauses de Horn

- Ce sont des clauses qui ont au plus un littéral positif
  - Un fait est composé d'un seul littéral positif
  - Une règle est composée de littéraux négatifs et d'un seul littéral positif
  - Une requête est composée uniquement de littéraux négatifs

Les faits

- pere(jean, paul).

- mult (0, X, 0).

Le produit d'un nombre par 0 est égal à 0

- Dans un fait, les variables sont quantifiées implicitement par un quantificateur universel

Les règles

On part de l'implication $(A1 \wedge A2 \wedge ... \wedge An \supset B)$.
Cette formule équivaut à $(B \vee \neg(A1 \wedge A2 \wedge ... \wedge An))$,
c'est à dire $(B \vee (\neg A1) \vee ... (\neg An))$.

Ces clauses ne contiennent qu'un seul littéral positif. On sépare alors les littéraux négatifs du littéral positif par le symbole ":-" et on termine la clause par un point. On écrit ainsi B:- A1, A2,..., An.

B s'appelle la tête de la clause et A1, A2, ..., An est le corps de cette même clause
Les clauses de Prolog peuvent s'interpréter de la façon suivante : pour satisfaire le but B, il est suffisant de satisfaire l'ensemble des conditions A1, ..., An.

Dans une règle, les variables sont quantifiées implicitement par un quantificateur universel

Les requêtes

- C'est une clause de Horn sans tête, une clause composée uniquement de littéraux négatifs. La requête $b1,\ b2,\ \ldots,\ bk\ ?$ s'interprète de la façon suivante : satisfaire la requête b_1, puis la requête b_2, ..., et enfin la requête b_k. On exprime ainsi une conjonction de buts

- Un programme PROLOG est composé de faits et de règles et une requête lance l'exécution

PROLOG : un outil de démonstration

- Pour modéliser un problème en PROLOG, il faut donc définir des règles et des faits. S'il faut rechercher la solution d'un problème P, la requête est simplement P.

- S'il faut démontrer une implication, $(A1 \wedge A2 \wedge ... \wedge An) \supset B$, il faut ajouter les littéraux $A1, A2, ..., An$ aux clauses déjà présentes, et poser la requête B.

- Dans une requête contenant des variables, intuitivement la requête s'interprète comme la recherche des valeurs des variables qui satisfont la requête.

La syntaxe SWI-Prolog

- Un **terme** est, soit une constante, soit une variable, soit un terme composé
- Une **constante** est soit un nombre (entier ou réel), soit une chaîne de caractères
- Une **variable** commence par une lettre majuscule ou par un blanc souligné.
- Un **terme composé** est, soit un terme fonctionnel, soit une liste
- Un **terme fonctionnel** est composé d'un symbole fonctionnel suivi de la liste de ses arguments séparés par une virgule. Père(jean, paul) est un terme d 'arité 2.
- Chaque argument peut lui-même être un terme.
- Les **listes** sont désignées par la liste de leurs éléments entre crochets. La liste vide sera notée []. Par exemple, [a,b,c] désigne une liste ordonnée composée des termes a, b et c, [[a,b],[c,g,l],m] est la liste composée de la liste [a,b], de la liste [c,g,l] et de l'élément m.

La syntaxe SWI-Prolog

- Un paquet de clauses est une suite ordonnée de clauses dont la tête est composée à partir du même prédicat

  - Exemple

```
pere(jeanne, paul).
pere(jules, pierre).
pere(X, Y) :- parent(X, Y), sexe(Y , masculin).
```

Mise en oeuvre de la résolution

- **L'effacement** : recherche du littéral complémentaire du but parmi les têtes des clauses disponibles dans l'espace de travail

  - Exemple

  /*C1*/pere(jeanne, paul).
  /*C2*/ pere(jules, pierre).
  /*C3*/ pere(X, Y) :- parent(X, Y), sexe(Y , masculin).

- La requête consiste en le but : père(jeanne, X).
- Le principe de résolution est appliqué avec les clauses :

  $\neg pere(jeanne, X) | pere(jeanne, paul) | pere(jules, pierre) | pere(X, Y)$
  $\vee \neg parent(X, Y) \vee \neg sexe(Y, masculin)$

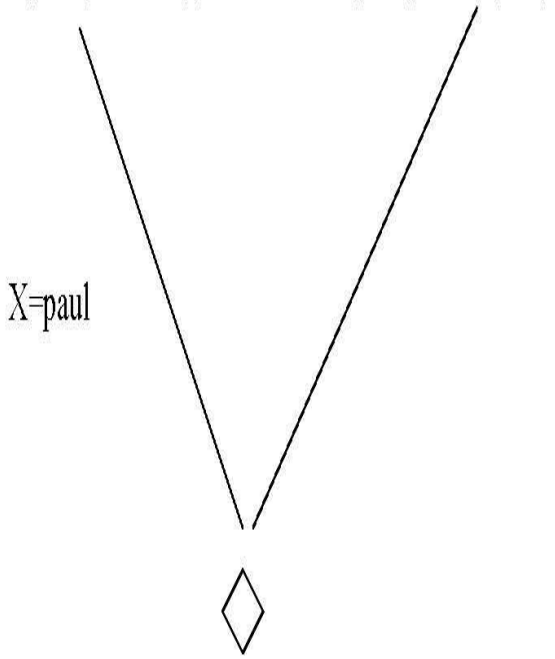
- La résolvante construite par l'interpréteur Prolog, reste implicite et ne vient pas s'ajouter à l'ensemble des clauses manipulées

not(pere(jeanne,X))

pere(jeanne, paul)

pere(jules, pierre)

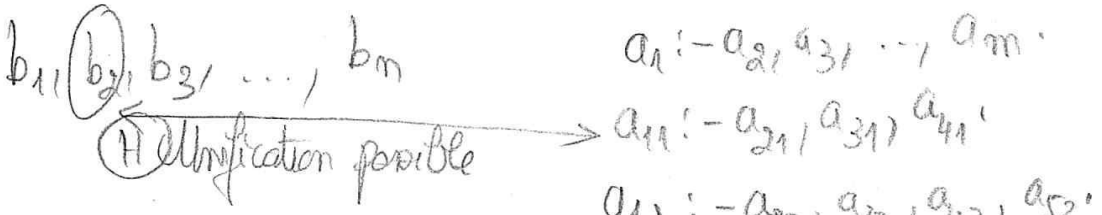
pere(X,Y) v not(parentX,Y))v not(sexe(Y, masculin))



Mise en oeuvre de la résolution

- Algorithme d'effacement
  - Prolog tente d'effacer les littéraux de la clause but, un à un dans l'ordre où ils se présentent
  - par exemple, b_1, b_2, b_3, ... b_n. provoque l'effacement de b_1, puis de b_2, ..., puis de b_n
    - EFFACER UN BUT b_i :
      - Prolog cherche la première clause a_1 :- a_2, a_3, ...a_m. dont la tête peut s'unifier avec b_i et place les autres clauses possibles en attente (pour un futur retour en arrière)
      - lors d'une unification, toutes les occurrences d'une même variable d'une clause reçoivent la même valeur
      - le but b_i est alors remplacé dans la suite des buts à satisfaire par le corps de la clause a_2, a_3, ..., a_m dont la tête a été unifiée avec b_i. Le but devient a_2, a_3, ..., a_m, b_{i+1}, ..., b_n ?

$b_1, b_2, b_3, \ldots, b_n.$

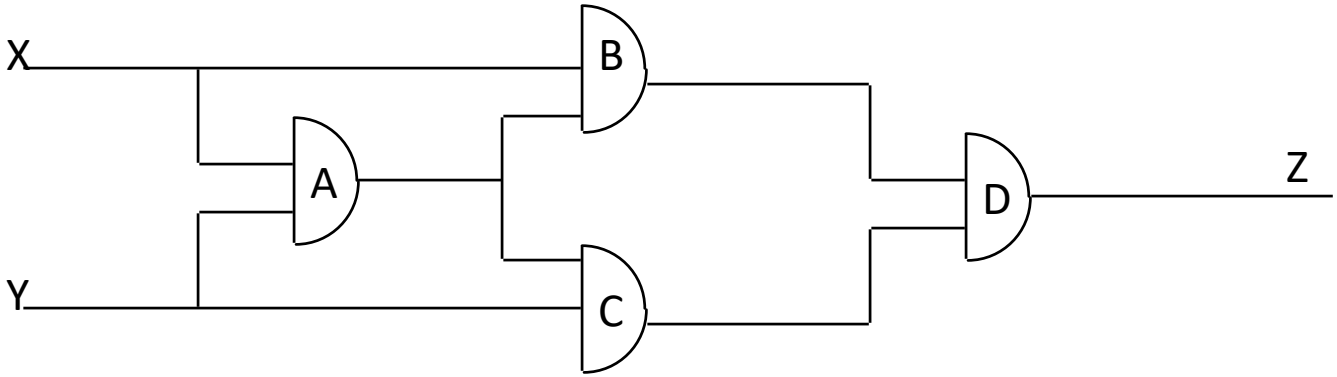


$b_1, a_{21}, a_{31}, a_{41}, b_3, \ldots, b_n$

Mise en oeuvre de la résolution

- Dès qu'un but ne peut s'effacer, il y a échec pour ce noeud de l'arbre de recherche et le retour arrière s'opère sur le choix des clauses en attente
- Si le but devient vide, la démonstration a réussi. L'interpréteur affiche l'instanciation des variables et le message YES puis effectue un retour en arrière vers les clauses en attente
- L'interpréteur PROLOG explore systématiquement toutes les branches de l'arbre de recherche et donne successivement toutes les solutions démontrables du problème proposé
- L'interpréteur Prolog explore cet arbre en profondeur d'abord puis de la gauche vers la droite

EXEMPLE : un circuit logique (1/2)



Les éléments de base sont des portes NAND dont voici la table de vérité :

nand(0,0,1).
nand(0,1,1).
nand(1,0,1).
nand(1,1,0).

EXEMPLE : un circuit logique (2/2)

- Le circuit représente un OU exclusif XOR que l'on va décrire à l'aide des élements du circuit
- xor(X,Y,Z) :- nand(X,Y,A),
  nand(X,A,B),
  nand(A,Y,C),
  nand(B,C,Z).

La requête :-xor(X,Y,Z) donne la table de vérité du XOR

X=0 Y=0 Z=0

X=0 Y=1 Z=1

X=1 Y=0 Z=1

X=1 Y=1 Z=0

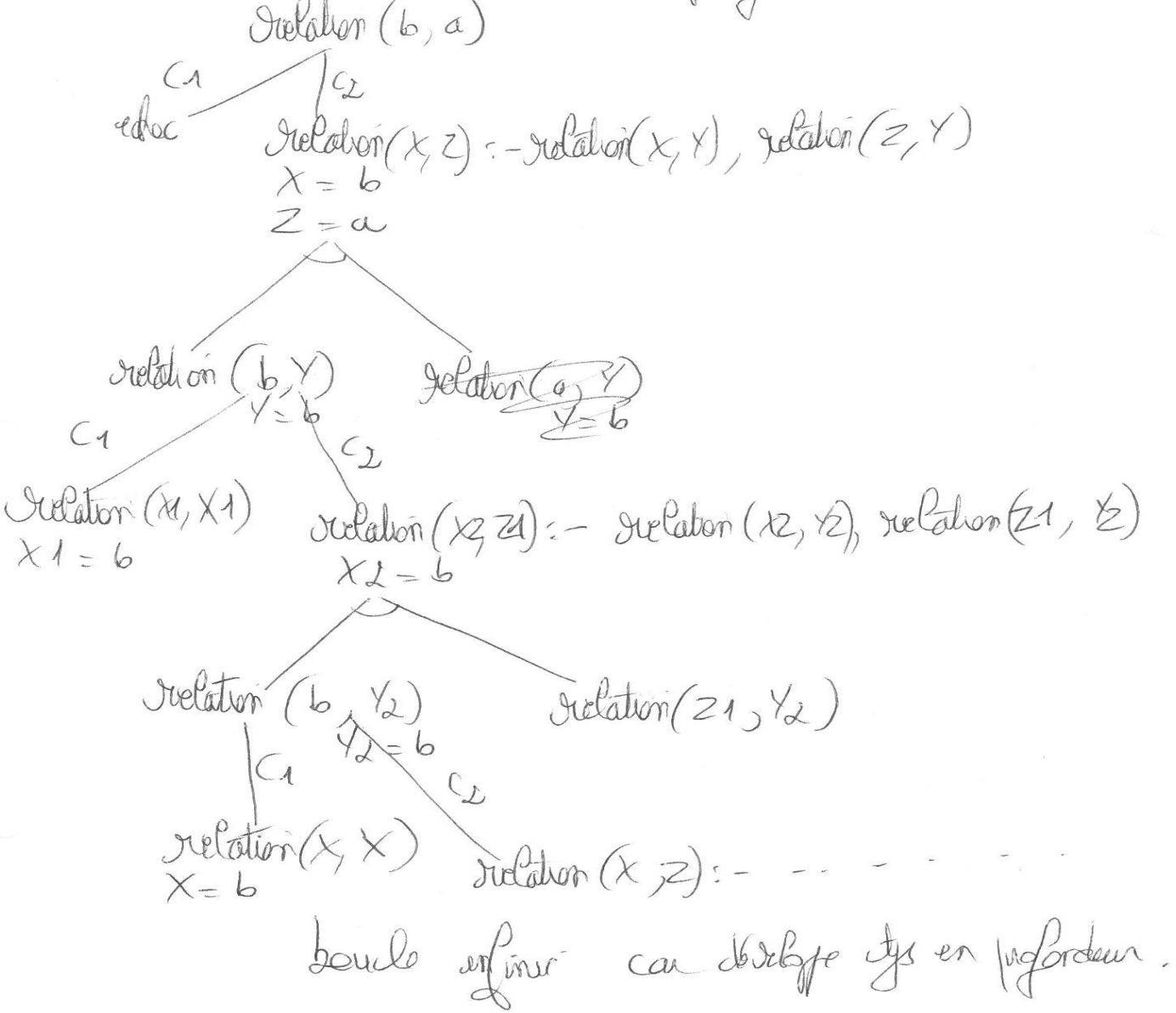
Les limites de la résolution en Prolog

- ORDRE DES CLAUSES
  - C1 relation(X,X).
  - C2 relation(X,Z):- relation(X,Y), relation(Z,Y).
  - C3 relation(a,b).

  Si notre but est relation(b,a)?

  ➔ le programme boucle et ne démontre pas la requête

① Arbre de la résolution en prolog



boucle infini car descente dfs en profondeur.

Les limites de la résolution en Prolog

- Si nous prenons maintenant le programme suivant :

A1 relation(a,b).

A2 relation(X,X).

A3 relation(X, Z):- relation(X,Y), relation(Z,Y).

relation(b,a) ?

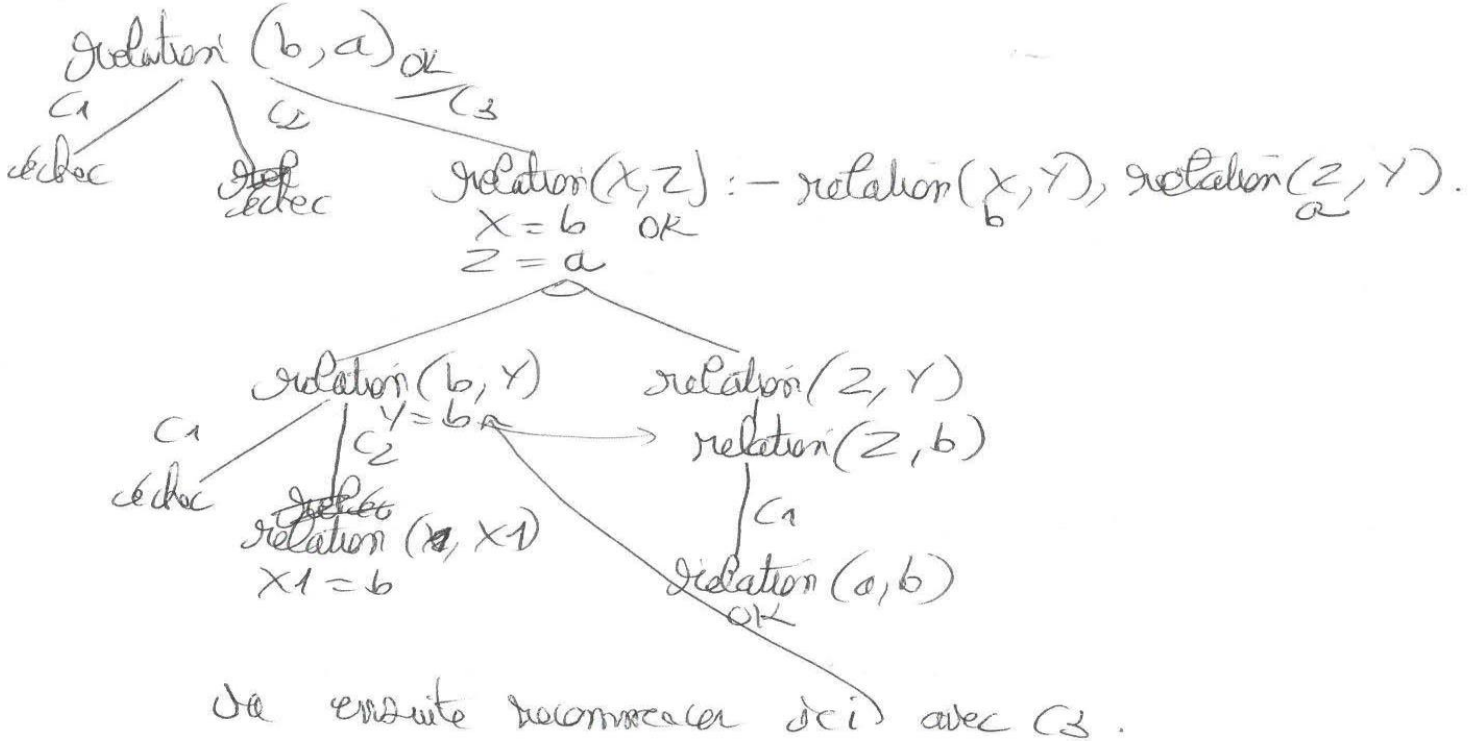
par A3 relation(b,Y1), relation(a, Y1) ?

par A2 relation(a,b) ?

par A1 : SUCCES

Arbre de la résolution en Prolog

Nouvelle version



Je ensuite recommencer ici avec C3.

Les limites de la résolution en Prolog

- L'ordre des littéraux

/*B1*/ ancêtre(X,Y):- parent(X,Z), ancêtre(Z,Y).

/*B2*/ ancêtre(X,Y):-parent(X,Y).

/*B3*/ parent(paul, jacques).

/*B4*/ parent(marie, jacques).

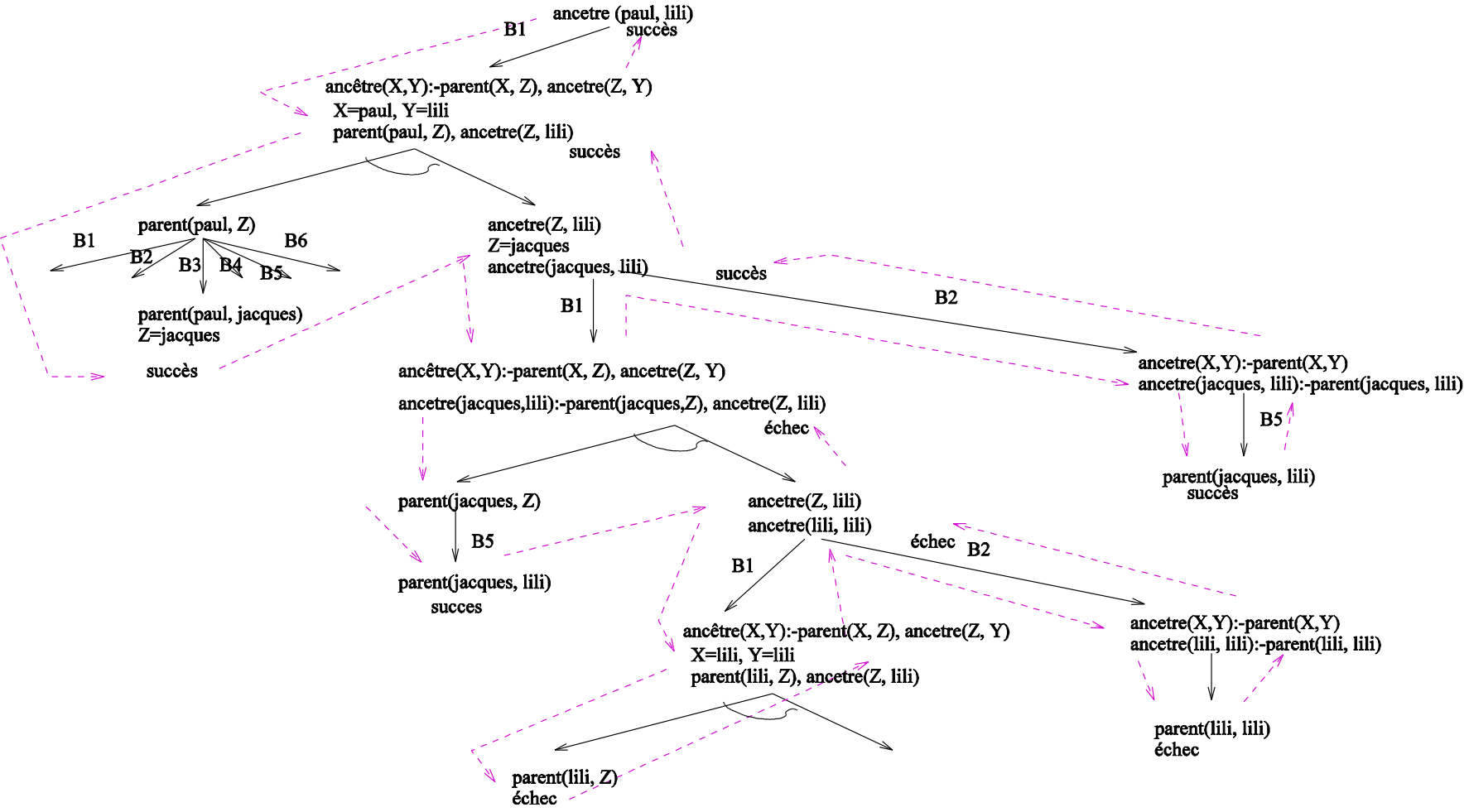
/*B5*/parent(jacques, lili).

/*B6*/ parent(martine, lili).

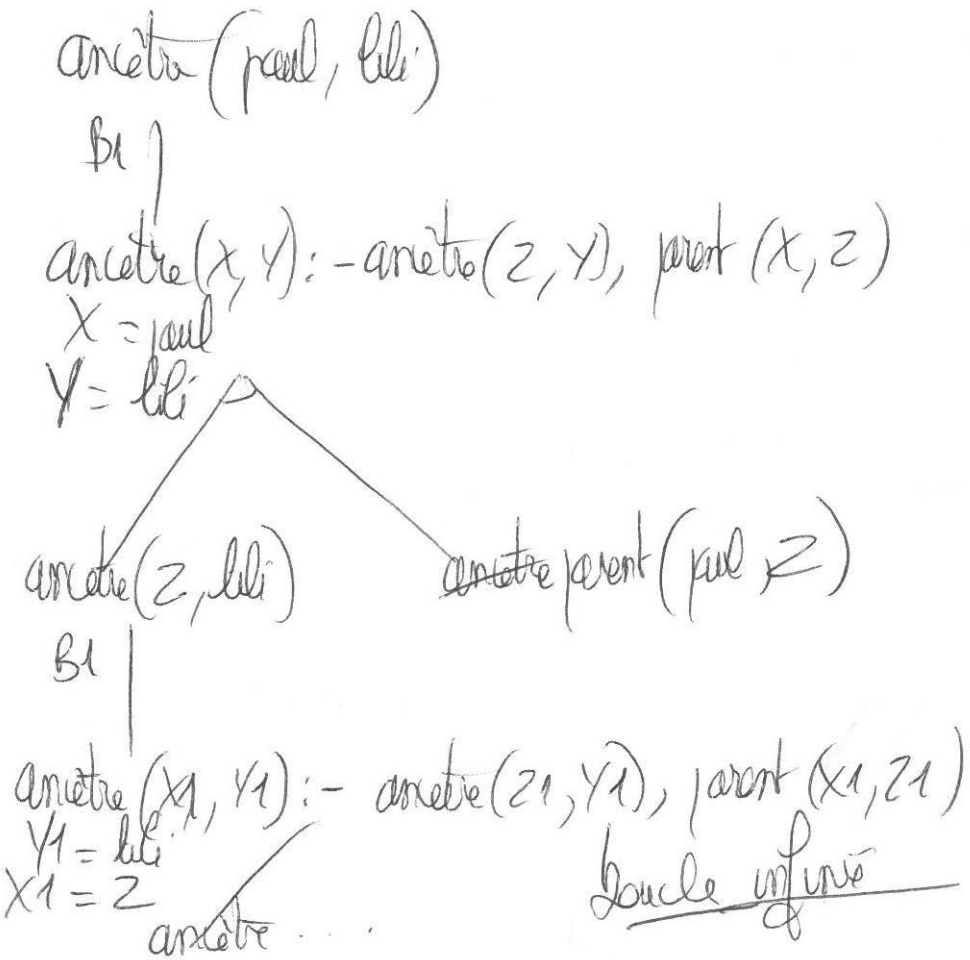
la requête "ancêtre(paul, lili)." réussit

- Si maintenant, on échange l'ordre des littéraux dans la clause B1, on a la clause ancêtre(X,Y):-ancêtre(Z,Y), parent(X,Z), l'arbre de recherche a une profondeur infinie, la stratégie de résolution ne permet pas d'aboutir

Les limites de la résolution en Prolog : V1 la requête réussit

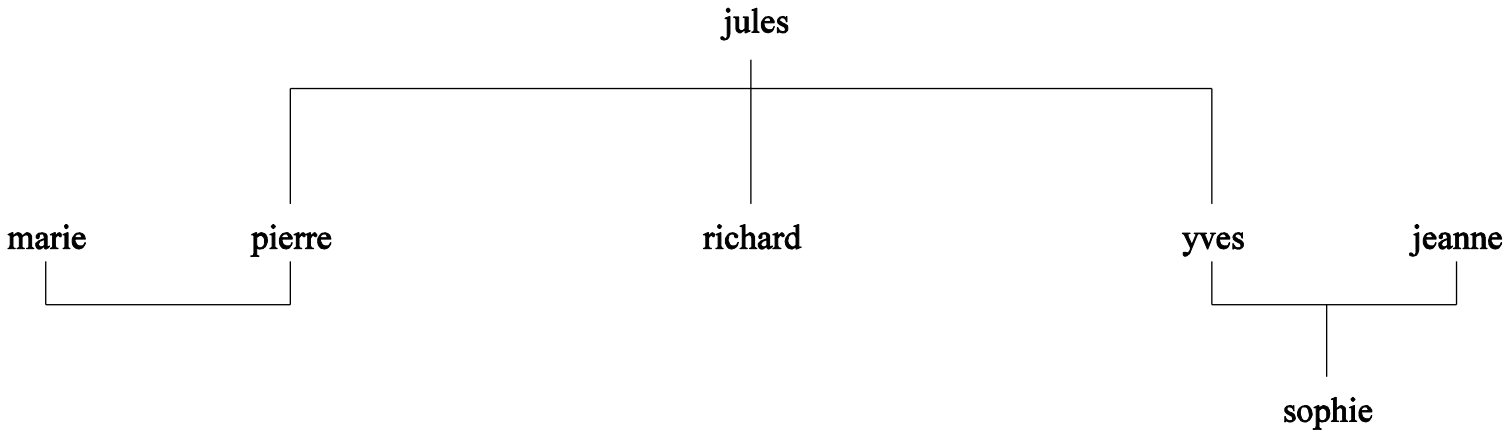


En échangeant ordre des littéraux dans B1.



EXEMPLE

UNE FAMILLE (inspirée de Sterling et Shapiro)



Décrire la famille à l'aide des prédicats pere, mere, homme, femme.
Ecrire des règles pour fils, fille, grand-pere, grand-mere.
Interroger cette base de données

L'environnement SWI-PROLOG

- L'utilisateur a la main ?-
- Pour rentrer des clauses il suffit de les entrer une à une par ***assert***

?- **assert**(pere(jules, pierre)).

?- **assert**(parent(X,Y):- pere(X,Y)).

?- **assert**(parent(X,Y):- mere(X,Y)).

?- **asserta**(Clause) : ajout d'une clause en tête d'un paquet

?- **assertz**(Clause) : ajout d'une clause en fin de paquet

- **retract**(Clause) La première clause unifiée avec Clause est retirée de l'espace de travail

L'environnement SWI-PROLOG

- **clause**(Tête, corps) : recherche d'une clause dans l'espace de travail dont la tête s'unifie avec tête, si le corps de la clause est vide, Corps s'unifie avec l'atome true

  - Exemple :

```
lgpaire([]).
lgIpaire([_]).
lgIpaire([_|L]):-lgpaire(L).
lgpaire([_|L]):-lgIpaire(L).
```

```
1 ?- clause(lgIpaire([_]),X).
X = true .
```

```
2 ?- clause(lgIpaire([1]),X).
X = true ;
X = lgpaire([]).
```

L'environnement SWI-PROLOG

- Pour visualiser l'espace de travail, on lance la requête

?- **listing**.

- Pour visualiser un paquet de clauses , on lance la requête **listing**(<paquet>)
- Pour lancer une requête,

?- pere(X,Y).

?-parent(X,pierre).

- Pour chaque solution, l'interpréteur affiche les valeurs prises par les variables ou YES. Pour voir les solutions suivantes, faire « ; » et non « return »

L'environnement SWI-PROLOG

- Pour quitter SWI-Prolog, on lance la requête halt.

- Pour charger un fichier dans l'espace de travail,
  - **consult**(nom-de-fichier).

    nom-de-fichier se termine obligatoirement par .pl

    exemple : consult(‘ c:\famille.pl ’).
  - Ou double-cliquer sur le fichier .pl sous windows

- Pour appeler l'éditeur de clauses, **edit**(fichier) ou edit(<paquet>)
  - exemple : edit(‘ c:\famille.pl ’).

L’environnement SWI-Prolog

- La mise au point : le mode TRACE
  - **trace**
  - **notrace**
- La trace sélective
  - **spy** <paquet>
  - **nospy** <paquet>
- Appel : on entre dans une clause définissant le paquet
- Sortie : le but courant est résolu
- Retour : on essaie une autre clause du paquet

Exemple de trace

trace, ancêtre(paul, lili).

Call: (7) ancêtre(paul, lili) ? creep

Call: (8) parent(paul, _G5213) ? creep

Exit: (8) parent(paul, jacques) ? creep

Call: (8) ancêtre(jacques, lili) ? creep

Call: (9) parent(jacques, _G5213) ? creep

Exit: (9) parent(jacques, lili) ? creep

Call: (9) ancêtre(lili, lili) ? creep

Call: (10) parent(lili, _G5213) ? creep

Fail: (10) parent(lili, _G5213) ? creep

Redo: (9) ancêtre(lili, lili) ? creep

Call: (10) parent(lili, lili) ? creep

Fail: (10) parent(lili, lili) ? creep

Fail: (9) ancêtre(lili, lili) ? creep

Redo: (8) ancêtre(jacques, lili) ? creep

Call: (9) parent(jacques, lili) ? creep

Exit: (9) parent(jacques, lili) ? creep

Exit: (8) ancêtre(jacques, lili) ? creep

Exit: (7) ancêtre(paul, lili) ? creep

Quelques prédicats prédéfinis de PROLOG - Les vérifications de type

- **var(X)** réussit si X est une variable non instanciée
- **nonvar(X)** X est une variable instanciée, ou un terme autre qu ’une variable
- **atom(X)** X est un atome (constante qui n’est pas un nombre)
- **number(X)** X est un nombre : entier, réel
- **integer(X)**
- **float(X)**

Quelques prédicats prédéfinis les expressions numériques (1/2)

- X = 2+3.

  X = 2+3. Rien ne demande à Prolog d'évaluer « 2+3 ».

- X is 2+3.

  X=5. « is » demande à Prolog d'évaluer « 2+3 ».

  Attention : lorsqu'une expression numérique est évaluée, toutes les variables doivent être instanciées

- X is 2/3.

  X=0, 6666666666666666.

- X is 2 div 3.

  X=0.

Pas de notation standardisée pour les calculs arithmétiques ➔/ et div dépendant des implémentations de Prolog.

Quelques prédicats prédéfinis les expressions numériques (2/2)

- opérateurs **+ - ; * / ^ mod**
- fonctions mathématiques **sin**, **cos**, **exp**, **log**
- comparateurs

= :=	=
= \ =	<>
=<	<=
>=	>=

- Identification (=unification)
  - X=Y
    X=Y.
    res(F,3)=res(F,4).
    False.
    res(X,3)=res(F,3).
    X=F.
  - X is 2.
    X=2.
  - Y =:=2.
    ERROR: =:=/2: Arguments are not sufficiently instantiated

Listes

- Liste composée des éléments a, b et c : [a,b,c]
- Liste vide : []
- Une liste peut contenir des éléments qui sont eux-mêmes des listes [a, [b,c]].
- Une liste non vide est représentée par [X|Y] où X est le premier élément de la liste et Y la liste privée de son premier élément
  - Exemple :

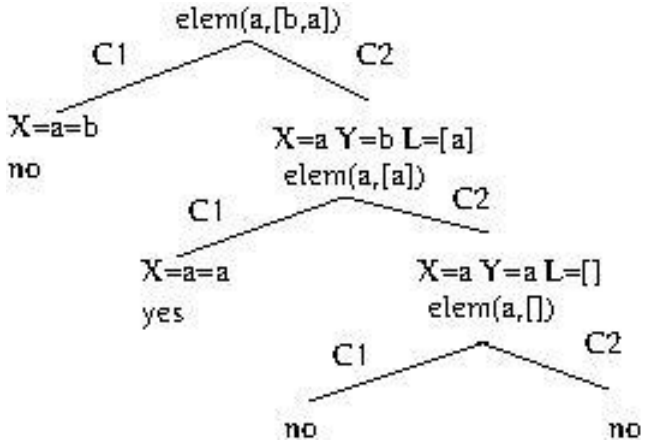
  si [X|Y]=[a,b,c] on a X=a et Y=[b,c]
  si [X|Y]=[a] on a X=a et Y=[]
  si [X,Y|L]=[a,b] on a X=a, Y=b et L=[].

Le traitement des listes

- Le parcours d'une liste
  - Exemple : « éléments de »

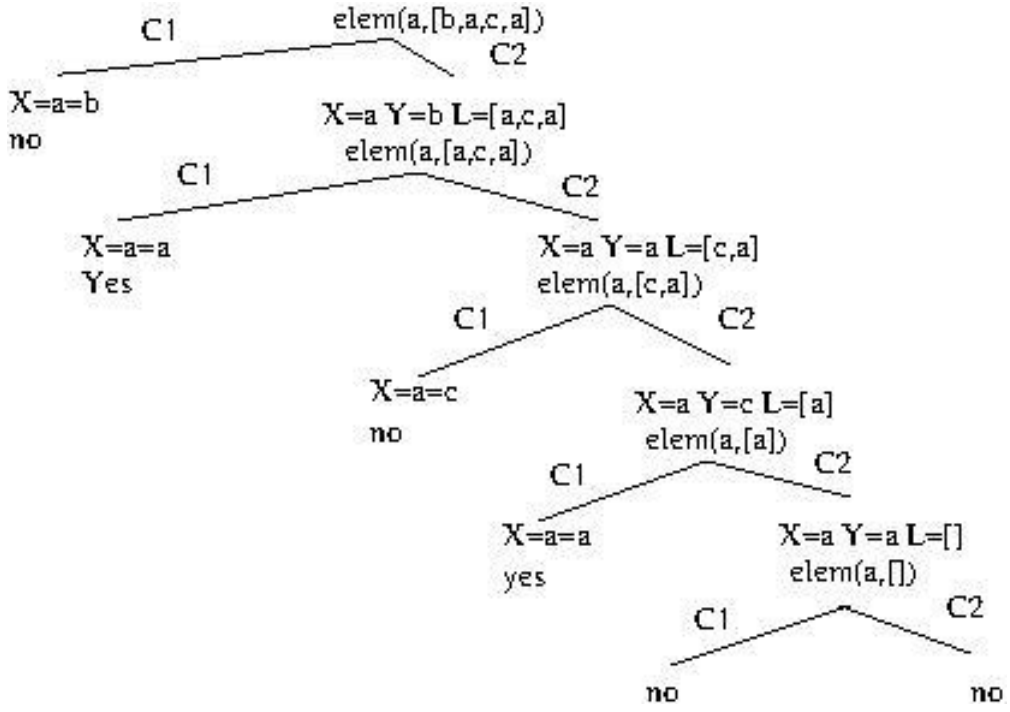
/*C1*/ elem(X, [X|L]).
/* C2*/ elem(X, [Y|L]):-elem(X,L).

elem(a,[b,a])?



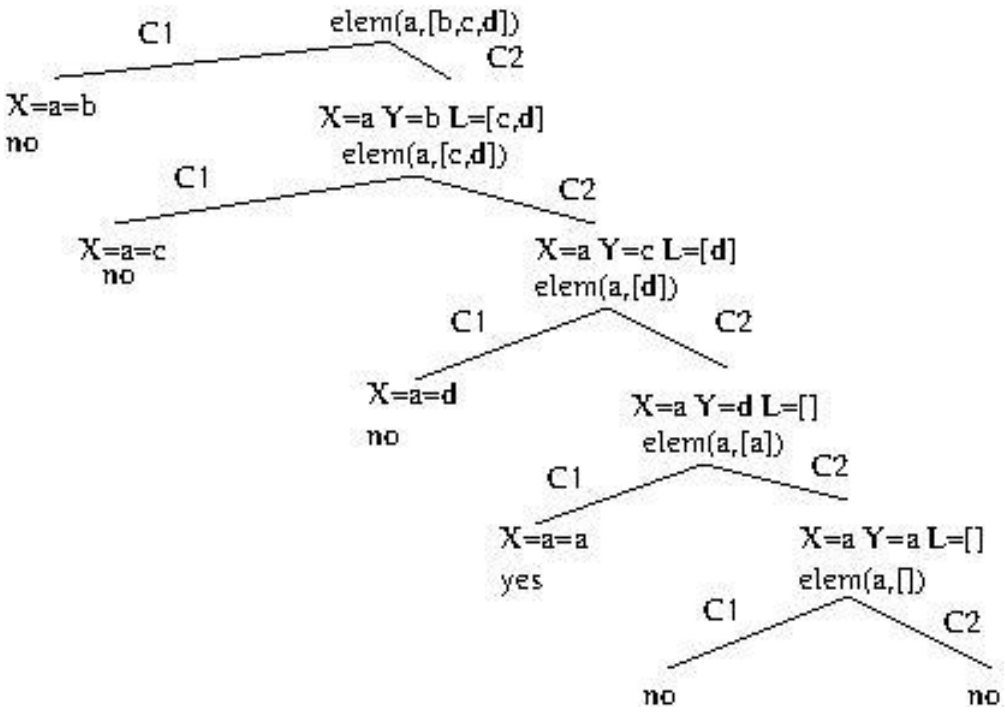
Le traitement des listes

- elem(a,[b,a,c,a])?



Le traitement des listes

- elem(a,[b,c,d])?



Concaténation de 2 listes

concat(f,g) ⇐ si vide(f) alors g
sinon
si tête(x,f) et corps(z,f)
alors concatg(x, concat(z,g))

```
/*C1*/ concat([X|Y],L,[X|R]):-concat(Y,L,R).
/*C2*/ concat([],L,L).
```

Inversion d’une liste

```
/*C1*/ reverse([],T,T).
/*C2*/reverse([X|Y],T,R):-reverse(Y,[X|T],R).
```

La coupure : contrôler la résolution

- Soient c_1, c_2...c_m les choix en attente; cm étant le plus récent, b1, ...,Bn? les buts à effacer.

- Soit la clause p:-f_1, f_2, ..., f_i, **!**, f_{i+1},..., f_k avec p et b1 unifiables

- La liste des choix en attente est maintenant c_1, c_2, ..c_m, c(b1) où c(b1) est la mémorisation du choix en instance pour l'effacement du but b1 et la suite des buts devient f_1,...f_i, !, f_{i+1}, f_k, b_2, ...b_n?

- Supposons que f_1, ...f_i soient effacés , la suite des choix en attente est alors c_1, c_2, ...c_m, c(b_1)..., c(f_i) et la suite des buts est !,f_{i+1},...,f_k, b_2,...,b_n?

La coupure

- Le prédicat prédéfini "**!**" s'efface car toujours évalué à vrai. L'effacement de la coupure "**!**" va supprimer tous les choix en attente jusqu'à la tête de la clause contenant la coupure

- Les choix en attente sont alors : c_1, c_2..., c_m.

- La suite des buts est : f_{i+1}, ..., f_k, b_2,...b_n?

La coupure

1. Soit p, la tête d'une clause p:-f1, f2,...fi, **!**, fi+1,...fk. Si le but Bi et p s'unifient alors la coupure présente dans cette clause élimine la sélection des clauses dont la tête peut s'unifier avec Bi, situées au dessous
2. La coupure n'affecte jamais les retours en arrière sur les buts qui sont situés à sa droite
3. La coupure élimine toutes les solutions distinctes qui pourraient apparaître pour la conjonction de buts qui sont situés à sa gauche

- Exemple d'utilisation de la coupure

```
/*C1*/ elem(X,[X|L]):-!.
/*C2*/elem(X,[Y|L]):-elem(X,L).
```

Exemple : Le traitement du OU exclusif

$(A1 \vee A2) \Rightarrow B$

$B \vee \neg(A1 \vee A2)$

$B \vee [(\neg A1) \wedge (\neg A2)]$

$[B \vee (\neg A1)] \wedge [B \vee (\neg A2)]$

C'est à dire deux clauses de Horn :

$B \vee (\neg A1)$

$B \vee (\neg A2)$

En utilisant PROLOG, on obtient :

/*C1*/ B :-A1.

/*C2*/ B :- A2.

Si on veut exprimer que les littéraux A1 et A2 s'excluent, on utilise alors la coupure.

/*C11*/vrai(vrai).

/*C12*/faux(faux).

/*C13*/ouE(A1, A2) :- vrai(A1), faux(A2),!.

/*C14*/ouE(A1, A2) :- faux(A1), vrai(A2),!.

Le traitement de la négation

- Le prédicat prédéfini **fail** s'efface systématiquement et provoque l'échec de la résolution en cours

- **not** est défini par :

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

  not(X).

  Si X s'efface alors not(X) échoue sinon not(X) réussit

- Exemples :

  ":-not(not(11 is 5+6))." réussit tandis que ":-not(X is 3+4)." échoue

  Il s'agit d'une "négation" dont l'interprétation est : tout ce qui est inconnu (non déductible) est considéré par Prolog comme faux.

  C'est l'hypothèse du Monde clos.

Le jeu du taquin : exploration d'un arbre d'états

- On veut programmer en Prolog le jeu du taquin
- On définit quatre opérateurs H,B,G,D qui permettent respectivement de déplacer une tuile vers le haut, vers le bas, vers la gauche et vers la droite
- on interdit les séquences des déplacements H B H, B H B, G D G, D G D.
- on veut programmer une stratégie d'exploration des états du type "best-first" et trouver la meilleure solution