

6. Contrôle de la résolution

(VIDEO-PPT 3.4)

6.1. Possibilités existantes

(emprunts à [R. Janvier])

Prolog offre un niveau d'expression très puissant : description de la solution d'un problème sous forme de relations et de contraintes. L'inconvénient est que l'utilisateur ne possède pas le contrôle de la résolution comme dans un langage impératif (C, Java) par exemple. Les seuls opérateurs de contrôle sont le choix : alternatives offertes par les clauses d'un même prédicat ; et le « et » séquentiel entre les buts de la partie droite d'une clause ou d'une question.

Il existe donc dans ce cadre différentes mises en œuvre de « choix » :

a) exprimer le choix dans la tête de clause

- *Exemple* : pour varier dans la représentation des données, en travaillant sur la représentation d'entiers par un arbre de successeurs d'un nombre avec le prédicat `pair` :

```
pair(0) .  
pair(succ(succ(N)) :- pair(N) .
```

b) exprimer le choix dans les contraintes de clause

- *Exemple* : en revenant aux listes

```
membre(X, [X|Q]) .  
membre(X, [Y|Q]) :- X\=Y, membre(X, Q) .
```

c) exprimer le choix dans les buts de la queue de clause

- *Exemple* : quand une liste encode un graphe

```
chemin(X,Y) :- arc(X,Y) .  
chemin(X,Y) :- arc(X,Z), chemin(Z,Y) .
```

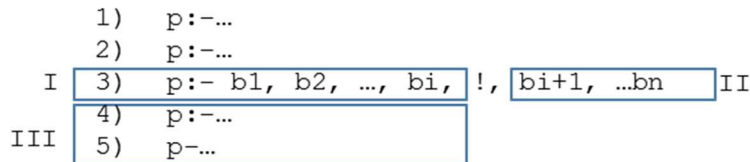
Ces techniques permettent donc d'obtenir l'exclusivité recherchée. Néanmoins, elles ne permettent pas de capturer toutes les situations où il devient inutile de continuer à parcourir l'arbre de recherche de la résolvante.

C'est pourquoi un outil de contrôle de la résolution (ou coupe-choix) a été introduit. Un tel outil n'a rien de logique, il s'utilise dans la vision opérationnelle de Prolog. S'il améliore l'efficacité de certains programmes, il engendre *a contrario* une perte d'expressivité des prédicats qui ne sont plus réversibles (les arguments ont alors un sens en entrée ou en sortie).

6.2. Coupe-choix (ou coupure)

Définition : Le pseudo-prédicat prédéfini **!** s'efface toujours (est toujours vrai) et demande à l'interpréteur de renoncer à tout choix laissé en suspens de la tête de clause courante incluse au **!** et ceux-là uniquement.

➤ Une vue du coupe-choix



Question : $|- p.$

- Prolog essaie de résoudre p avec les règles 1 et 2
- Exécution de la règle 3 :
 - Tant que **!** n'est pas effacé
 - Exécution normale du bloc I (avec éventuellement des retours internes)
 - Un retour jusqu'à la tête de clause déclenche l'exécution du bloc III
 - Si **!** s'efface :
 - Tous les choix laissés en suspens sur I sont coupés, y compris ceux sur la tête de clause, (le bloc III n'est donc pas exécuté)
 - Le bloc II s'exécute normalement sans possibilité de retour avant **!**

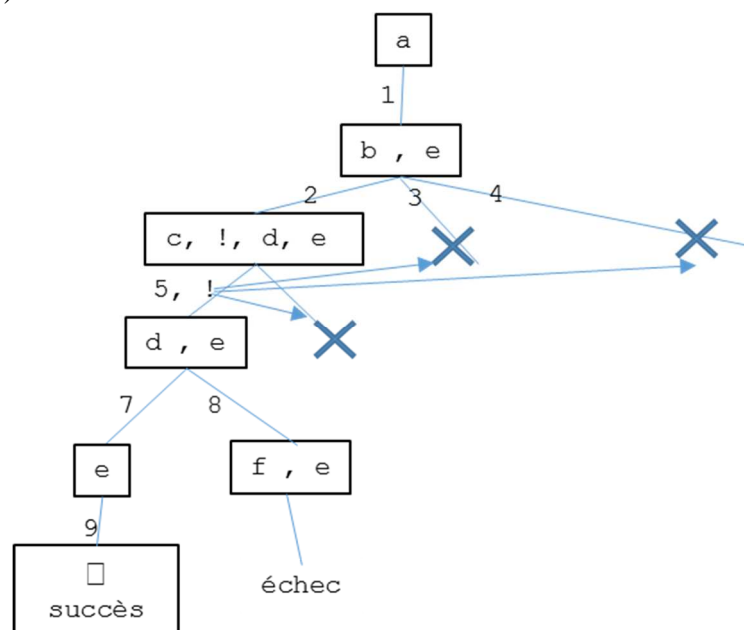
Le coupe-choix est un mécanisme d'anti-retour, une fois franchi le **!** il n'est plus possible de revenir en arrière (backtrack) avant celui-ci.

➤ Autre exemple (synthétique)

Base de connaissances :

1. $a :- b, e.$
2. $b :- c, !, d.$
3. $b :- c.$
4. $b :- e.$
5. $c.$
6. $c :- f.$
7. $d.$
8. $d :- f.$
9. e

Question : $|- a.$



Il convient de remarquer que l'effacement du coupe-choix provoque l'élagage des choix pendants de même niveau (3 et 4) mais aussi de celui du premier sous-but rencontré (6) !

➤ Application : Tri par permutations

Le principe du tri par permutations consiste à générer toutes les listes possibles à partir des éléments d'une liste *L* donnée. Parmi ces listes, l'une est triée.

D'où le programme :

```
1. triPermut(L, Lpermut) :- permut(L, Lpermut) , croissant(Lpermut), ! .
```

Le coupe-choix indique à Prolog que dès qu'il a trouvé une solution, il peut s'arrêter (car elle est unique).

Les permutations d'une liste s'obtiennent en concaténant chaque élément *x* de la liste avec chaque permutation de la liste moins *x*.

Permutations de [2, 1, 3] :

```

2, --- 1, 3
   --- 3, 1
1, --- 2, 3
   --- 3, 2
3, --- 2, 1
   --- 1, 2

```

Pour réaliser les permutations d'une liste *L*, il faut énumérer les éléments *x* de *L* et obtenir *L* privée de *x*. Le prédicat `enlever` réalise cette décomposition :

```

1. enlever(X, [X|Q], Q).
2. enlever(X, [Y|Q], [Y|Q1]) :- enlever(X, Q, Q1).

```

D'où :

```

1. permut([], []).
2. permut(L, [X|L2]) :- enlever(X, L, L1) , permut(L1, L2).

```

Pour terminer, le prédicat `croissant` vérifie si une liste est croissante.

```

1. croissant([X]).
2. croissant([X, Y|Q]) :- X =< Y , croissant (Q).

```

L'intérêt de ce tri n'est pas son efficacité mais sa construction algorithmique basée sur la méthode de programmation par essais successifs (ou générer-tester), qui sera abordée dans la suite du polycopié.


➤ Application : Négation par échec

La négation par l'échec découle de l'hypothèse du monde fermé, c'est-à-dire que ce qui n'est pas démontrable doit pouvoir être considéré faux. Le prédicat `non(p)` est donc vrai si `p` n'est pas démontrable. Ainsi avec la base de la famille Kennedy déjà introduite, le but `non(pere(john, x))` ne cherche pas tous les `x` dont `john` n'est pas le père, mais constate que `john` a un enfant et donc répond faux.

Soit `fail` le pseudo-prédicat qui est toujours faux (*remarque* : pour définir `fail` il suffit de **ne pas** le définir dans la base de connaissances). On peut alors écrire le méta-prédicat `non/1` de la façon suivante :


1. `non(P) :- P, !, fail.`
2. `non(P).`


Le méta-prédicat `non` prend en paramètre un prédicat `P` qui est placé en position d'évaluation en partie droite de la règle 1.

 **Attention** : Si le terme utilisé en prédicat `P` contient des variables libres, les résultats peuvent être « surprenants ». Ainsi si l'on cherche à savoir qui est à la fois « un homme » et « non riche », en raison de la sémantique opérationnelle du coupe-choix, seule une séquence de buts fonctionnera, la commutativité de la conjonction sera perdue ! :

1. `homme(daniel).`
 2. `homme(cresus).`
 3. `riche(cresus).`
- ```
?- homme(X), non(riche(X)).
X = daniel

?- non(riche(X)), homme(X).
no
```

 *Le prédicat `non/1` est clairement différent de la négation logique (puisque la conjonction logique ne tient pas). Il devra être utilisé en dernier recours car il empêche la réversibilité.*

 *Pour cette raison Gprolog ne propose pas de prédicat prédéfini `not(P)` mais utilise la notation `\+(P)` avec `P` unifiable avec une tête de clause pour marquer que ce méta-prédicat n'est pas la négation dans tous les cas d'appel.*

## ➤ Application : Boucles d'interactions

L'interaction avec l'utilisateur est clairement un schéma d'exécution orienté dans lequel le coupe-choix sera précieux.

- Schéma d'une boucle mue par le succès :

```
boucle :- traitement, !, boucle.
boucle.
```

Tant que `traitement` réussit, la boucle est relancée.

Par exemple, avec le prédicat `traitement` défini ci-dessous, `boucle` lit un entier et affiche son carré jusqu'à la lecture d'un 0 (`traitement` échoue dans ce cas et provoque l'arrêt de la boucle).

```
traitement :- read_integer(X) , carre(X).
carre(0) :- ! , write(fin) , fail.
carre(X) :- X2 is X*X , write(X2), nl.
```

- Schéma d'une boucle mue par échec :

```
1. boucle :- traitement , !.
2. boucle :- boucle.
```

Tant que le `traitement` échoue, la boucle est relancée par la règle 2.

### MiniTP :

- 1) *Ecrire le prédicat `nonUnifiable (X, Y)` qui est vrai si `x` et `y` ne sont pas unifiables.*
- 2) *Vous avez ci-dessous deux versions d'un prédicat `mini`, expliquer pourquoi le résultat de la question `mini(2,3,3)` est correct avec `min1` (répond no) et incorrect avec `min2` (répond yes)*

```
min1(X,Y,X) :- X=<Y .
min1(X,Y,Y) :- X>Y .

min2(X,Y,X) :- X=<Y, ! .
min2(X,Y,Y) .
```