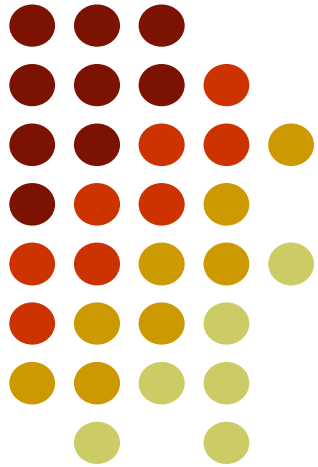
Le langage Prolog



Cours n°3

Les listes (suite)

Notion de coupure

Résolution d’un type de problèmes de logique : les intégrammes

Partie 1
-
Parcours de liste
Construction de liste

Rappel : notations sur les listes

- La notation `[X|Y]` représente une liste non-vide dont la **tête** (le 1er élément) est X et la **queue** (le reste de la liste) est Y.
  - Cela constitue la base de l'utilisation des listes dans les programmes Prolog.
  - **|** symbole spécial décomposant la liste en Tête et Queue : [Tête | Queue] correspond à [Car | Cdr] (cf. Scheme)
  - [a, b, c] ≡ [a | [b | [c | [ ]]]]
  - La tête est un élément et la queue est une liste :

```
?- [a, b] = [X|Y].
X = a, Y = [b].
```

- La notation [ ] représente la liste vide.

Parcours d'une liste

- Cas de base :

`parcours([],v).` v = valeur associée à la liste vide

- Cas général :

Parcours([T|Q],R):- parcours(Q,RI), R is f(RI,T).

R est calculé à partir de RI et de T

Exemple : somme des éléments d'une liste

```
somme([], 0).
somme([T|Q], S) :- somme(Q, SQ), S is T + SQ.
```

Exercice

Ecrire le prédicat nbpairs/2 qui prend en premier argument une liste d'entiers et produit en second argument le nombre d’entiers pairs de la liste.

Exemple d'utilisation:

```
?- nbpairs([5,4,1,12,3],N).
N = 2.
```

Solution de l'exercice

```
nbpairs([], 0).
nbpairs([T|Q], N) :- T mod 2 =:= 0, nbpairs(Q, N1), N is N1+1.
nbpairs([T|Q], N) :- T mod 2 =:= 1, nbpairs(Q, N).
```

Construction d'une liste (1)

- Nous avons vu comment parcourir une liste.
- Voyons comment en construire une.
  - Examinons le problème suivant : L une liste contenant un nombre pair d'éléments.
    - Par exemple : L = [0,1,2,3,4,5]
  - Cherchons à construire une nouvelle liste W en prenant les éléments de rang impair dans L.
    - Dans notre exemple cela donnerait : W = [0,2,4]
  - Soit elements_rang_impair ce prédicat:

    elements_rang_impair([0,1,2,3,4,5], X).

    X = [0,2,4]

Construction d'une liste (2)

- Relation de récurrence pour l'énumération des éléments de rang impair :

  elements_rang_impair([X1, X2 | Y]) :- elements-rang-impair(Y).

- Règle d'arrêt : il faut s'arrêter si la liste vide :

  elements_rang_impair([]).

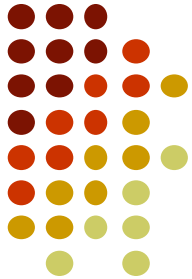
- On modifie le programme de façon à construire la nouvelle liste à partir du parcours de l'ancienne. D'où le programme suivant:

  (R1) elements_rang_impair([], []).

  (R2) elements_rang_impair([X1, X2|Y], [X1|L]) :- elements_rang_impair(Y, L).

- Interprétation de ces deux règles :
  - R1 : prendre un élément sur deux de la liste vide donne la liste vide
  - R2 : prendre un élément sur deux dans la liste [X1, X2 |Y], donne la liste [X1|L], si prendre un élément sur deux dans la liste Y donne la liste L.

Construction d'une liste (3)



- Ce programme est incomplet :

  Elements_rang_impair([], []).

  Elements_rang_impair([X, _|Y], [X|L]) :- elements_rang_impair(Y, L).

- Exemples d'exécution :

  ?- elements_rang_impair([a,b,c,d],L).

  L = [a, c]

  ?- elements_rang_impair([a,b,c],L).

  false

- Pourquoi cet échec ? :

  ⇒ Le cas des listes à nombre impair d'éléments n'est pas traité !

Construction d'une liste (4)

- D'où le programme final suivant :

```
Elements_rang_impair([], []).
Elements_rang_impair([X], [X]).
Elements_rang_impair([X, _|Y], [X|L]) :- elements_rang_impair(Y, L).
```

- Règle introduite : si une liste est formée d'un seul élément X, la liste résultat est cette liste [X].
- Exemples d'exécution :

```
?- elements_rang_impair([a,b,c,d],L).
L = [a, c]
?- elements_rang_impair([a,b,c],L).
L = [a, c]
```

- Remarque : les éléments dans la liste résultat sont rangés dans le même ordre que dans la liste initiale.

Exercice

Ecrire le prédicat listepairs/2 qui prend en premier argument une liste d'entiers et produit en second argument la liste des entiers pairs qui apparaissent dans la première liste.

Exemple d'utilisation:

```
?- listepairs([5,4,1,12,3],L).
L = [4, 12].
```

Solution de l'exercice

```
listepairs([], []).
listepairs([T|Q], [T|L]):- T mod 2 =:= 0, listepairs(Q, L).
listepairs([T|Q], L):- T mod 2 =:= 1, listepairs(Q, L).
```

Partie 2
-
Accumulateur

Construction d'une liste au moyen d'une liste auxiliaire appelée « accumulateur » (1)

Principe : la liste est construite pendant les appels récursifs.

Il s'agit d'introduire un argument supplémentaire correspondant à une liste « accumulatrice » :

- à l'appel initial, la liste accumulatrice vaut la liste vide: []
- à chaque appel récursif on ajoute des éléments à la liste construite
- en fin de parcours, la liste construite (« accumulée ») constitue le résultat du calcul.

Construction d’une liste au moyen d’une liste auxiliaire appelée « accumulateur »(2)

Problème : Ecrire un prédicat inverser qui inverse l’ordre des éléments d’une liste donnée.

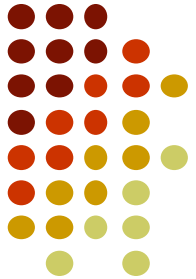
- Soit D une liste donnée, R est la liste D inversée.
- Nous allons introduire une liste auxiliaire pour y ranger les éléments énumérés au fur et à mesure de leur parcours.
- Le prédicat inverser aura donc 3 arguments:

  (1) la liste à inverser,

  (2) la liste auxiliaire,

  (3) la liste inversée

Construction d'une liste au moyen d'une liste auxiliaire « accumulateur » (3)



- On obtient les règles suivantes :

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

  - Les 2 premiers arguments sont consultés, le troisième est élaboré
  - Au lancement, la liste auxiliaire est vide : `inverser([a,b,c],[],R).`
  - L'élément qui apparaît en tête de la liste est retiré et placé en tête de la liste auxiliaire. Il y a donc inversion de l'ordre des éléments
  - Cas de base: après le parcours de la liste donnée, L et R sont identiques.
- Interprétation de ces deux règles :
  - Inverser la liste vide, à partir de la liste auxiliaire L, a pour résultat la liste L.
  - Inverser la liste [X|Y], à partir de la liste auxiliaire L, donne la liste résultat R, si inverser la liste Y, à partir de la liste auxiliaire [X|L], donne cette même liste R.

Construction d'une liste au moyen d'une liste auxiliaire « accumulateur » (4)

- Si l'on souhaite disposer d'un prédicat inverser/2 qui ne mentionne que les listes donnée et résultat, on peut introduire le prédicat suivant :

```
inverser(L,R) :- inverser(L,[],R).

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

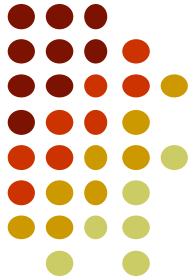
- Exécution :

```
?- inverser([a,b,c], [], R).
R = [c, b, a]

?- inverser([a,b,c],R).
R = [c, b, a]
```

Partie 3
-
Prédicats prédéfinis sur les listes

Quelques prédicats prédéfinis sur les listes



- Dans la documentation Prolog :
  - les arguments consultés sont précédés d'un **+**
  - les arguments élaborés sont précédés d'un **-**
  - les arguments qui peuvent être soit consultés, soit élaborés (suivant le sens de l'unification) sont précédées d'un **?**

  Exemple : `numlist(+Min, +Max, -List).`

```
?- numlist(2,8,L).
L = [2, 3, 4, 5, 6, 7, 8]
```

Prédicats simplifiant la manipulation des listes (1)

is_list(+Terme) réussit si Terme est une liste

```
?- is_list([a,b,c]).
true.
```

length/2 permet d'obtenir la longueur d'une liste :

```
?- length([a,b,c],L).
L = 3.
?- length(L,3).
L = [_3648, _3654, _3660].
```

```
?- length([a,b,c],4).
false.
```

member(?Element, ?Liste) réussit si Element s'unifie avec un élément de Liste. Prédicat utilisé principalement pour énumérer les membres de la liste :

```
?- member(E,[a,b,c]).
E = a ;
E = b ;
E = c.
```

```
?- member(d,[a,b,c]).
false
?- member(b,[a,b,c]).
true
```

memberchk(?Element, ?Liste) permet de vérifier si Element appartient à Liste :

```
?- memberchk(d,[a,b,c]).
false.
?- memberchk(b,[a,b,c]).
true.
```

```
?- memberchk(X,[a,b,c]).
X = a.
```

Prédicats simplifiant la manipulation des listes (2)

last(?Liste, ?Last) permet de sélectionner le dernier élément d'une liste.

```
?- last([a,b,c],X).
X = c.
```

nth0(?I, ?Liste, ?Elem) permet de sélectionner Elem l'élément d'indice I de Liste, le premier élément ayant l'indice 0 :

```
?- nth0(3,[a,b,c,d,e],E).
E = d.
```

nth0 permet aussi d'obtenir le ou les indice(s) d'un élément donné dans une liste :

```
?- nth0(I,[a,b,c,b,d,e],b).
I = 1 ;
I = 3
```

nth1(?I, ?Liste, ?Elem) fonctionne exactement comme nth0, mais l'indice du premier élément de la liste est 1 :

```
?- nth1(3,[a,b,c,d,e],E).
E = c.
```

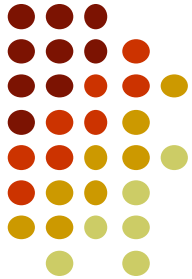
```
?- nth1(I,[a,b,c,b,d,e],b).
I = 2 ;
I = 4
```

select(?Elem,?Liste,?Reste) fonctionne comme member mais en plus unifie Reste avec Liste privée de Elem (permet de sélectionner un élément et le supprimer de la liste) :

```
?- select(b,[a,b,c,d,b,a],L).
L = [a, c, d, b, a] ;
L = [a, b, c, d, a] ;
false.
```

```
?- select(E,[a,b,c],R).
E = a, R = [b, c] ;
E = b, R = [a, c] ;
E = c, R = [a, b].
```

Prédicats simplifiant la manipulation des listes (3)



flatten(+Liste1, -Liste2) applanit Liste1 et unifie le résultat dans Liste2 :

```
?- flatten([a,[[],b],[[c],d,e]],L).
L = [a, b, c, d, e].
```

reverse(+Liste1, -Liste2) inverse l'ordre des éléments de Liste1 et unifie le résultat dans Liste2 :

```
?- reverse([a, b, c, d, e],L).
L = [e, d, c, b, a].
```

sumlist(+Liste, -Somme) unifie Somme avec la somme de tous les éléments de Liste (numériques):

```
?- sumlist([3.4,12,17.2,3],S).
S = 35.6
```

permutation(?Liste1, ?Liste2) vrai si Liste2 est une permutation de Liste1 ; permet également d'énumérer toutes les permutations possibles de Liste1 ou Liste2.

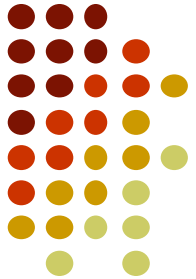
```
?- permutation([a,b,c],P).
P = [a, b, c] ;
P = [a, c, b] ;
P = [b, a, c] ;
P = [b, c, a] ;
P = [c, a, b] ;
P = [c, b, a] ;
false.
```

```
?- permutation([a,b,c],[c,a,b]).
true
```

msort(+Liste, -Triee) trie les éléments de Liste pour créer Triée :

```
?- msort([12,3,-7,9,-3,17,11,10,3,9,6,18],L),writeln(L).
[-7,-3,3,3,6,9,9,10,11,12,17,18]
L = [-7, -3, 3, 3, 6, 9, 9, 10, 11|...].
```

Concaténation : prédicat append



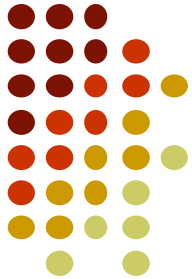
- **append** est le prédicat prédéfini pour la concaténation de listes : append(?List1, ?List2, ?List3).

```
?- append([a,b,c],[d,e],L).
L = [a, b, c, d, e]
```

- **append** est complètement symétrique et peut être utilisé pour d'autres opérations :
  - Trouver tous les découpages d'une liste en 2 sous-listes:

```
?- append(L1,L2,[a,b,c,d]).
L1 = [], L2 = [a, b, c, d] ;
L1 = [a], L2 = [b, c, d] ;
L1 = [a, b], L2 = [c, d] ;
L1 = [a, b, c], L2 = [d] ;
L1 = [a, b, c, d], L2 = []
```

append : autres utilisations possibles



- Trouver le dernier élément d'une liste :

```
?- append(_,[X],[a,b,c,d]).
X = d
```

- Découper une liste :

```
?- append(L2,L3,[b,c,a,d,e]), append(L1,[a],L2).
L2 = [b, c, a]
L3 = [d, e]
L1 = [b, c]
```

Conversions entre atomes chaines et listes

atom_chars(?Atom, ?Liste) unifie un atome avec la liste des atomes caractères qui le composent :

```
?- atom_chars(bonjour,L).        ?- atom_chars(X,[b, o, n, j, o, u, r]).
L = [b, o, n, j, o, u, r].       X = bonjour.
```

string_chars(?String, ?Liste) unifie une chaine avec la liste des atomes caractères qui le composent :

```
?- string_chars("hello",L).      ?- string_chars(C,[h, e, l, l, o]).
L = [h, e, l, l, o].             C = "hello".
```

atom_string(?Atom, ?String) unifie un atome avec la chaine des caractères qui le composent :

```
?- atom_string(foo, Z).          ?- atom_string(A,"hello world!").
Z = "foo".                       A = 'hello world!'.
```

number_string(?Number, ?String) unifie un nombre avec la chaine des caractères qui le composent :

```
?- number_string(3.14159265,S).  ?- number_string(N,"3.14159265").
S = "3.14159265".                N = 3.14159265.
```

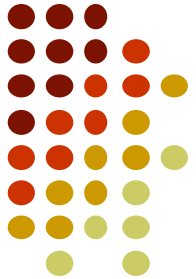
string_codes(?String, ?Codes) unifie une chaine avec la liste des codes des caractères de la chaine:

```
?- string_codes("hello",L).      ?- string_codes(S,[104, 101, 108, 108, 111]).
L = [104, 101, 108, 108, 111].   S = "hello".
```

atomics_to_string(+List, -String) convertit une liste d'éléments atomiques en une chaine :

```
?- atomics_to_string([hello, ' ', "world ", 33],S).
S = "hello world 33".
```

Prédicats sur les chaines



string_length(+String, -Long) unifie une chaine sa longueur :

```
?- string_length("bonjour !",Lg).
Lg = 9.
```

string_chars(?String, ?Chars) unifie une chaine avec la liste caractères qui le composent :

```
?- string_chars("hello",L).
L = [h, e, l, l, o].
```

```
?- string_chars(C,[h, e, l, l, o]).
C = "hello".
```

string_concat(?String1, ?String2, ?String3)

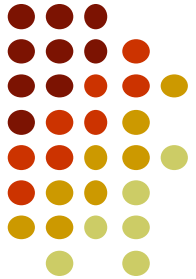
```
?- string_concat(A,B,"Bonjour").
A = "", B = "Bonjour" ;
A = "B", B = "onjour" ;
A = "Bo", B = "njour" ;
A = "Bon", B = "jour" ;
A = "Bonj", B = "our" ;
A = "Bonjo", B = "ur" ;
A = "Bonjou", B = "r" ;
A = "Bonjour", B = "".
```

sub_string(+String, ?Before, ?Length, ?After, ?SubString)

```
?- sub_string("Hello world !",3,A,2,B).
A = 8,
B = "lo world".
```

```
?- sub_string("Hello world !",0,5,_,B).
B = "Hello".
```

Quelques prédicats prédéfinis sur les ensembles (listes sans doublons)



is_set(+Terme).
    réussit si Terme est un ensemble c'est-à-dire une liste ne comportant pas de doublons
list_to_set(+List, -Set).
    transforme List en un ensemble Set (suppression des doublons)
subset(+Subset, +Set).
    Subset $\subset$ Set
intersection(+Set1, +Set2, -Set3).
    Set3 = Set1 $\cap$ Set2
union(+Set1, +Set2, -Set3).
    Set3 = Set1 $\cup$ Set2
subtract(+Set, +Delete, -Rest).
    Rest = Set - Delete
merge_set(+Set1, +Set2, -Set3).
    Set3 = fusion de Set1 et Set2
    (Set1, Set2 et Set3 sont des ensembles triés)
sort(+List, -SetTrié).
    transforme List en un ensemble trié SetTrié

maplist : application d'un prédicat à tous les éléments d'une liste

Les prédicats **maplist** appliquent un prédicat sur tous les membres d'une liste ou jusqu'à ce que le prédicat échoue.

maplist(+Pred,+List). Pred est d'arité 1

```
?- maplist(atomic,[a,12,31.1,hello,"world"]).
true.
maplist( <(3), [15,8,5,6,9,4] ).
true
```

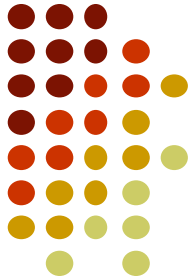
maplist(+Pred, ?List1, ?List2). Pred est d'arité 2

```
?- maplist(atom_chars,[il,fait,beau],L).
L = [[i, l], [f, a, i, t], [b, e, a, u]].
```

maplist(+Pred, ?List1, ?List2, ?List3). Pred est d'arité 3

```
?- maplist(string_concat,["il","est","bon"],["et","elle","ne"],L).
L = ["ilet", "estelle", "bonne"].
?- maplist(numlist,[1,2,3],[4,6,8],L).
L = [[1, 2, 3, 4], [2, 3, 4, 5, 6], [3, 4, 5, 6, 7, 8]].
```

Liste des solutions d'un prédicat



- Les prédicats findall/3, bagof/3 et setof/3 permettent de faire la liste des solutions d'un prédicat.
- findall(+Motif, +But, -Liste).
  - Trouve toutes les solutions de But et en fait une Liste selon le modèle Motif
- bagof(+Motif, +But, -Liste).
  - Trouve toutes les solutions de But (par groupes), échoue si But n'a pas de solution
- setof(+Motif, +But, -Liste).
  - Trouve toutes les solutions de But (par groupes), mais supprime les doublons ; échoue si But n'a pas de solution

findall, bagof, setof : exemples (1)

- Soit le programme suivant :

```
?- findall(C, foo(A,B,C), L).
L = [c, f, d, f, f, e, g]

?- bagof(C, foo(A,B,C), L).
A = a, B = b, L = [c, f, d, f] ;
A = b, B = c, L = [f, e] ;
A = c, B = c, L = [g]

?- setof(C, foo(A,B,C), L).
A = a, B = b, L = [c, d, f] ;
A = b, B = c, L = [e, f] ;
A = c, B = c, L = [g]

?- findall([A,B,C], foo(A,B,C),L).
L = [[a,b,c],[a,b,f],[a,b,d],[a,b,f],[b,c,f],[b,c,e],[c,c,g]]
```

les variables A et B sont unifiées. On a 3 solutions: une pour chaque couple (A,B) disctinct.

L est ordonnée

```
foo(a, b, c).
foo(a, b, f).
foo(a, b, d).
foo(a, b, f).
foo(b, c, f).
foo(b, c, e).
foo(c, c, g).
```

findall, bagof, setof : exemples (2)

?- bagof(C, A^foo(A,B,C), L).

B = b, L = [c, f, d, f] ;

B = c, L = [f, e, g] ;

?- setof(C, A^foo(A,B,C), L).

B = b, L = [c, d, f] ;

B = c, L = [e, f, g] ;

A^ signifie “ne pas unifier A”

seule la variable B est unifiée, la variable A restant libre. On a 2 solutions: une par valeur de B différente.

```
foo(a, b, c).
foo(a, b, f).
foo(a, b, d).
foo(a, b, f).
foo(b, c, f).
foo(b, c, e).
foo(c, c, g).
```

?- bagof(C, A^B^foo(A,B,C), L).

L = [c, f, d, f, f, e, g]

?- setof(C, A^B^foo(A,B,C), L).

L = [c, d, e, f, g]

les variables A et B ne sont pas unifiées. On n'a alors plus qu'une solution. (~ findall)

Partie 4
-
Coupure

Notion de coupure (1)

- La recherche systématique de toutes les solutions possibles est une caractéristique importante de Prolog.
- Mais elle constitue également son talon d'Achille car elle conduit parfois à une explosion combinatoire qu'il est nécessaire de couper si l'on veut voir son programme se terminer.

Notion de coupure (2)

- Différents noms utilisés : **coupure**, **cut** ou **coupe choix**
- Introduit un contrôle du programmeur sur l'exécution de ses programmes Prolog :
  - permet d'élaguer des branches de l'arbre de recherche
  - rend les programmes plus simples et plus efficaces
- Notation: **!** (ou / dans le Prolog de Marseille)
- Le coupe choix est un prédicat prédéfini qui réussit toujours.

Notion de coupure (3)

- Le coupe choix permet de signifier à Prolog qu'on ne désire pas conserver les points de choix en attente
- Les autres solutions possibles ne sont donc pas examinées

Coupure dans une règle

- Le coupe-choix ne conserve que les unifications faites avant la suppression des points de choix.

```
a(1).
a(2).
b(1).
b(2).
p(X):- a(X), b(X).
```

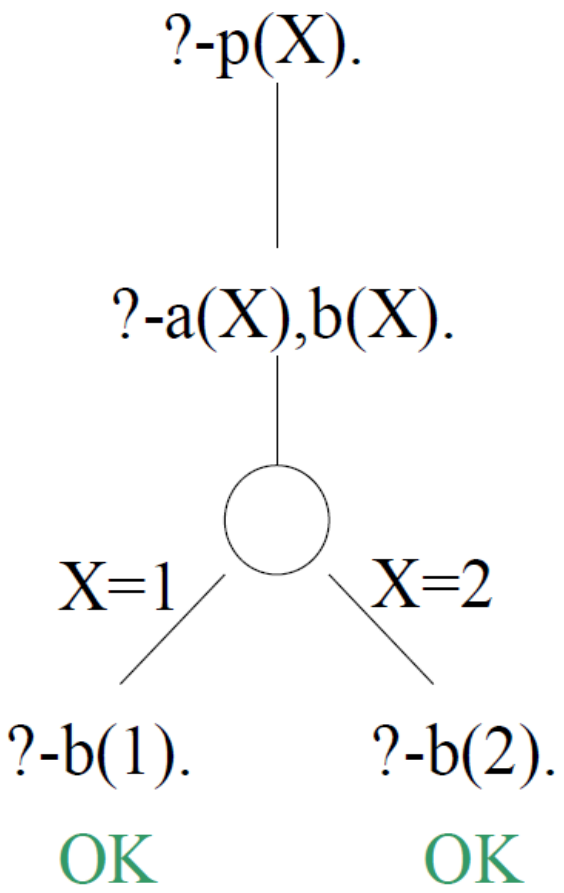
```
?- p(X).
X =1 ;
X=2.
```

```
a(1).
a(2).
b(1).
b(2).
p(X):- a(X), !, b(X).
```

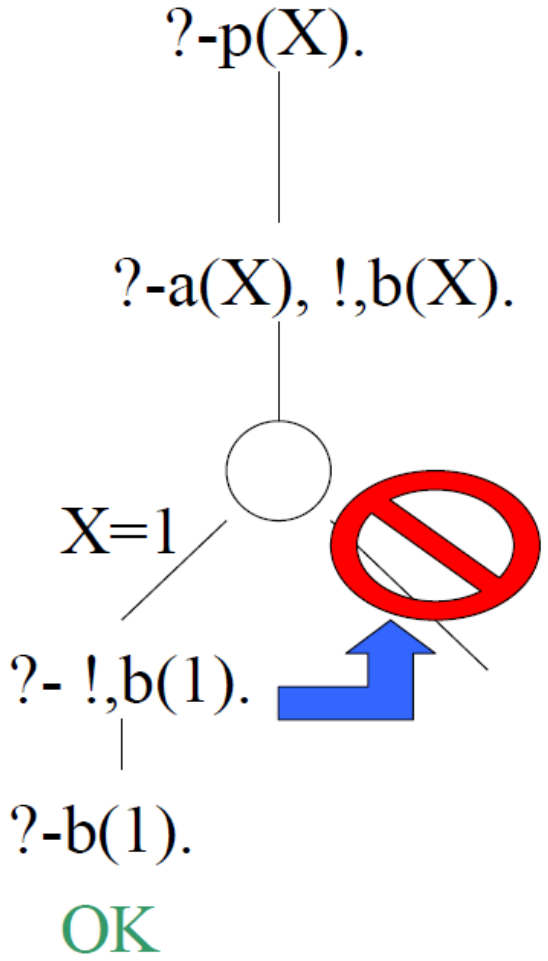
```
?- p(X).
X =1.
```

Arbre de Recherche

Sans Coupure



Avec Coupure



Notion de coupure (4)

- Le coupe-choix permet :
  - d'éliminer des points de choix
  - d'éliminer des tests conditionnels que l'on sait inutiles
    ⇒ plus d'efficacité lors de l'exécution du programme
- Quand Prolog démontre un coupe choix, il ignore toutes les règles du même prédicat, qui le suivent.
  - Un coupe choix élague toutes les solutions pour la conjonction de buts qui apparaissent à sa gauche dans la règle
  - En revanche il n'affecte nullement la conjonction de buts qui se trouve à sa droite

Coupure dans une règle

```
a(1).
a(2).
b(1).
b(2).
q(X,Y):- a(X), b(Y).
```

```
?- q(X,Y).
X = Y, Y = 1 ;
X = 1, Y = 2 ;
X = 2, Y = 1 ;
X = Y, Y = 2.
```

```
a(1).
a(2).
b(1).
b(2).
q(X,Y):- a(X), !, b(Y).
```

```
?- q(X,Y).
X = Y, Y = 1 ;
X = 1, Y = 2.
```

Exemples d'utilisation

- Pour calculer min(a,b) :

```
min(A, B, A) :- A=<B.
min(A, B, B) :- A >B.
```

```
min(A, B, A) :- A =< B, !.
min(A, B, B).
```

- Exprimer des clauses exclusives :

```
humain(X) :- femme(X).
humain(X) :- homme(X).
```

```
humain(X) :- femme(X), !.
humain(X) :- homme(X), !.
```

Danger de la coupure

- On peut perdre des solutions :

achete(helene, journal).

achete(corinne, timbres).

achete(X, pain).

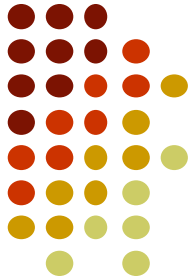
achete(helene, journal) :- !.

achete(corinne, timbres) :- !.

achete(X, pain).

- l'interrogation: achete*(helene, N).*
- Sans coupure : N=journal et N=pain
- Avec coupure : N=journal

Exemple de simplification obtenue avec le coupe-choix



Conjecture de Syracuse (TD2)

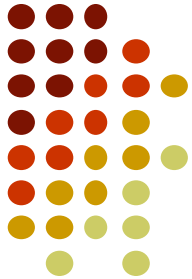
Prenez un entier positif. S'il est pair, divisez-le par 2 ; s'il est impair, multipliez-le par 3 et ajoutez lui 1. Immanquablement, on aboutit à la valeur 1. Par exemple, à partir de l'entier 7, on obtient la suite suivante : 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Quel que soit l'entier de départ, il semble que l'on aboutisse toujours à 1. Cette conjecture (dite « conjecture de Syracuse ») attend toujours une preuve ! Ecrire le prédicat syracuse/2 qui détermine la longueur de la suite correspondant à un entier donné. Par exemple, avec la valeur 7, ce prédicat retournera 17.

```
?- syracuse(7,L)

L=17.
```

```
syracuse(1,1).
syracuse(N,L):- N mod 2 =:= 1, N \== 1, N1 is N*3+1, syracuse(N1,L1), L is L1+1.
syracuse(N,L):- N mod 2 =:= 0, N1 is N//2, syracuse(N1,L1), L is L1+1.
```

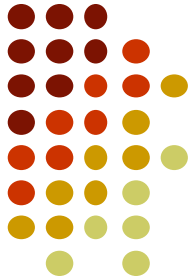
Exemple de simplification obtenue avec le coupe-choix



```
syracuse(1,1).
syracuse(N,L):- N mod 2 =:= 1, N \== 1, N1 is N*3+1, syracuse(N1,L1), L is L1+1.
syracuse(N,L):- N mod 2 =:= 0, N1 is N//2, syracuse(N1,L1), L is L1+1.
```

```
syracuse(1,1):- !.
syracuse(N,L):- N mod 2 =:= 1, !, N1 is N*3+1, syracuse(N1,L1), L is L1+1.
syracuse(N,L):- N1 is N//2, syracuse(N1,L1), L is L1+1.
```

Autre exemple d'utilisation du coupe-choix



- Pour calculer la racine carrée entière d'un nombre, on utilise un générateur de nombres entiers *entier(k)*. L'utilisation du coupe-choix est alors indispensable après le test K*K>N car la génération des entiers n'a pas de fin.

entier(0).

entier(N1) :- entier(N), N1 is N+1.

*racine(N, R) :- entier(K), K*K >N, ! , R is K-1.*

Notion de coupure (fin)

- 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

Le prédicat fail

- Le prédicat *fail/0* est un prédicat qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure.
- Prolog effectue alors un back track (retour arrière jusqu'au dernier point de choix)
- Toutes les instanciations de variable faites dans la branche sont annulées

Le prédicat fail

- L’utilisation de *fail* peut permettre de réaliser des itérations.

  Exemple :

  but1:- between(1,10,X), write(X), nl, fail.

  but1.

Combinaison cut-fail

- Enchaînement :- …, !, fail.
- Le cut a supprimé tous les points de choix
- Le fail retourne au dernier point de choix
- Leur combinaison termine donc la démonstration
- Permet d'exprimer une exception à une règle :

Tout oiseau qui n’est pas une autruche vole :

```
vole(autruche) :- !, fail.
vole(X) :- oiseau(X).
```

Définition de la négation

- La négation se définit par l'enchainement cut-fail.
- *not* est définit par les deux clauses suivantes:

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

  not(_).
- 1ère ligne : si X est prouvé, cut, fail : arrêt et false
- 2ème ligne: sinon, true (c-à-d qu'il est vrai que X n'est pas prouvé)

Partie 5

-

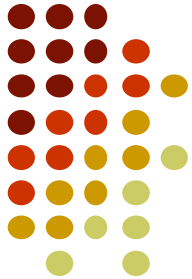
Résolution d’un type de problèmes de logique

Les intégrammes

Les intégrammes

- Les **intégrammes** appelés parfois logigrammes sont un type de casse-tête logique.
- On donne un certain nombre d'indices, desquels il faudra déduire l'intégralité des relations entre tous les éléments.

Exemple d’intégramme simple



Max, Eric et Luc habitent chacun une maison différente. Ils possèdent chacun un animal domestique distinct.

On sait que :

- Max a un chat.
- Eric n’habite pas en pavillon.
- Luc habite un studio, il n’a pas le cheval.

La problématique à résoudre est :

- Qui habite le château et qui a le poisson ?

Question : comment représenter le problème en Prolog ?

Modélisation : les faits

```
% les maisons :
maison(studio).
maison(pavillon).
maison(château).
% les animaux :
animal(chat).
animal(cheval).
animal(poisson).
```

Modélisation : les règles

```
% le prédicat habitation représente la relation
% entre une personne, sa maison et son animal :
% Max a un chat :
habitation(max,M,chat) :- maison(M).
% Eric n'habite pas en pavillon :
habitation(eric,M,A) :- maison(M), M\==pavillon, animal(A).
% Luc habite un studio, il n'a pas le cheval :
habitation(luc,studio,A):-animal(A),A\==cheval.
```

Résolution du problème : prédicat résoudre

Astuce du jour :

```
% le prédicat resoudre décrit l'ensemble du problème à résoudre
% avec ses 4 éléments inconnus  et affiche la solution :
resoudre :-
    habitation(max,MM,chat),
    habitation(eric,ME,AE),
    habitation(luc,studio,AL),
    is_set([MM,ME,studio]),
    is_set([chat,AE,AL]),
    write(max),write(' '),write(MM),write(' '),writeln(chat),
    write(eric),write(' '),write(ME),write(' '),writeln(AE),
    write(luc),write(' '),write(studio),write(' '),writeln(AL).
```

Interrogeons Prolog

```
?- resoudre.
max pavillon chat
eric chateau cheval
luc studio poisson
true ;
false.
```

On souhaite maintenant résoudre l'intégramme suivant

Dans une rue 3 maisons voisines sont de couleurs différentes : rouge, bleue et verte. Des personnes de nationalités différentes vivent dans ces maisons et elles ont chacune un animal de compagnie différent.

Les données du problème sont :

- l'anglais vit dans la maison rouge.
- le jaguar est l'animal de l'espagnol.
- le japonais vit à droite de la maison du possesseur de l'escargot.
- le possesseur de l'escargot vit à gauche de la maison bleue.

La problématique à résoudre est :

- Qui possède le serpent ?

Modélisation du problème

- On va représenter la solution par une liste de 3 termes, chaque terme aura la forme suivante : `maison(couleur,nationalité,animal)`
- Le programme Prolog ne contient qu'un prédicat :

```
serpent(N, Rue) :- …
```

  - `N` représente la nationalité du possesseur du serpent
  - `Rue` représente la solution complète, c'est-à-dire la liste de 3 termes `maison(couleur,nationalité,animal)`

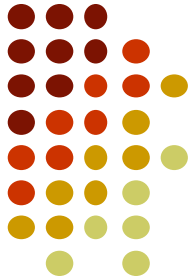
Solution

```
serpent(N, Rue) :-
    % Rue est représentée par une liste de 3 maisons :
    length(Rue,3),
    % il y a une maison rouge, une maison bleue et une maison verte :
    member(maison(rouge,_,_),Rue),
    member(maison(bleue,_,_),Rue),
    member(maison(verte,_,_),Rue),
    % l'anglais vit dans la maison rouge :
    member(maison(rouge,anglais,_),Rue),
    % le jaguar est l'animal de l'espanol :
    member(maison(_,espagnol,jaguar),Rue),
    % le japonais vit à droite de la maison du possesseur de l'escargot :
    nextto(maison(_,_,escargot),maison(_,japonais,_),Rue),
    % le possesseur de l'escargot vit à gauche de la maison bleue :
    nextto(maison(_,_,escargot),maison(bleue,_,_),Rue),
    % le troisième animal est un serpent :
    member(maison(_,N,serpent),Rue).
```

Interrogeons Prolog

```
?- serpent(N,Rue).
N = japonais,
Rue = [maison(rouge, anglais, escargot),
       maison(bleue, japonais, serpent),
       maison(verte, espagnol, jaguar)]
```

Utilisons cette modélisation pour le problème précédent



```
resoudre2(S):-
    % il y a 3 habitations: studio, chateau, pavillon (non ordonnés)
    S = [habitation(_,studio,_), habitation(_,chateau,_),
         habitation(_,pavillon,_)],
    % il y a un poisson
    member(habitation(_,_,poisson),S),
    % il y a un cheval
    member(habitation(_,_,cheval),S),
    % Max a un chat
    member(habitation(max,_,chat),S),
    % Eric n'habite pas en pavillon :
    member(habitation(eric,HE,_),S), HE \== pavillon,
    % Luc habite un studio, il n'a pas le cheval :
    member(habitation(luc,studio,AL),S), AL \== cheval.
```

Interrogeons Prolog

```
?- resoudre2(S).
S = [habitation(luc, studio, poisson),
     habitation(eric, chateau, cheval),
     habitation(max, pavillon, chat)]
```