Comment Prolog répond t-il aux questions ? (Exemples)

```
parent(marc,ines).
parent(marc,fred).
parent(marc,luc).
parent(luc,anne).
parent(luc,ines).
parent(ines,paul).
```

Il y a quatre possibilités de définir le prédicat `descendant(X,Z)` qui exprime le fait que Z est descendant de X.

Version 1:

```
desc1(X,Z) :-
     parent(X,Z).
desc1(X,Z) :-
     parent(X,Y),
     desc1(Y,Z).
```

Version 2:

```
% On change l'ordre des clauses de version 1
desc2(X,Z) :-
     parent(X,Y),
     desc2(Y,Z).
desc2(X,Z) :-
     parent(X,Z).
```

Version 3:

```
% On change l'ordre des deux buts dans
% la deuxième clause de la version 1
desc3(X,Z) :-
     parent(X,Z).
desc3(X,Z) :-
     desc3(Y,Z),
     parent(X,Y).
```

Version 4:

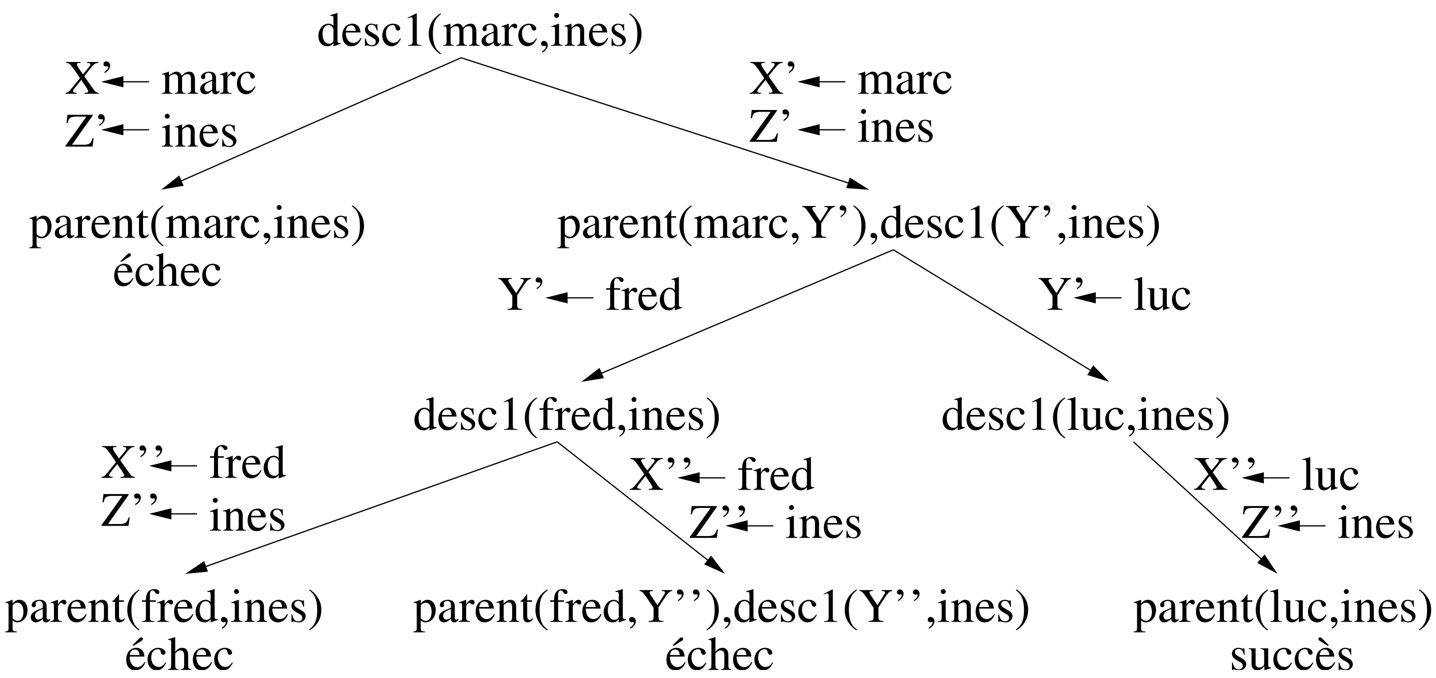
```
% On change l'ordre des clauses dans version 3
desc4(X,Z) :-
     desc4(Y,Z),
     parent(X,Y).
desc4(X,Z) :-
     parent(X,Z).
```

Toutes ces versions ont la même sémantique déclarative mais une sémantique opérationnelle différente.

La sémantique opérationnelle dépend de l'ordre des clauses et de l'ordre des buts à l'intérieur des clauses.

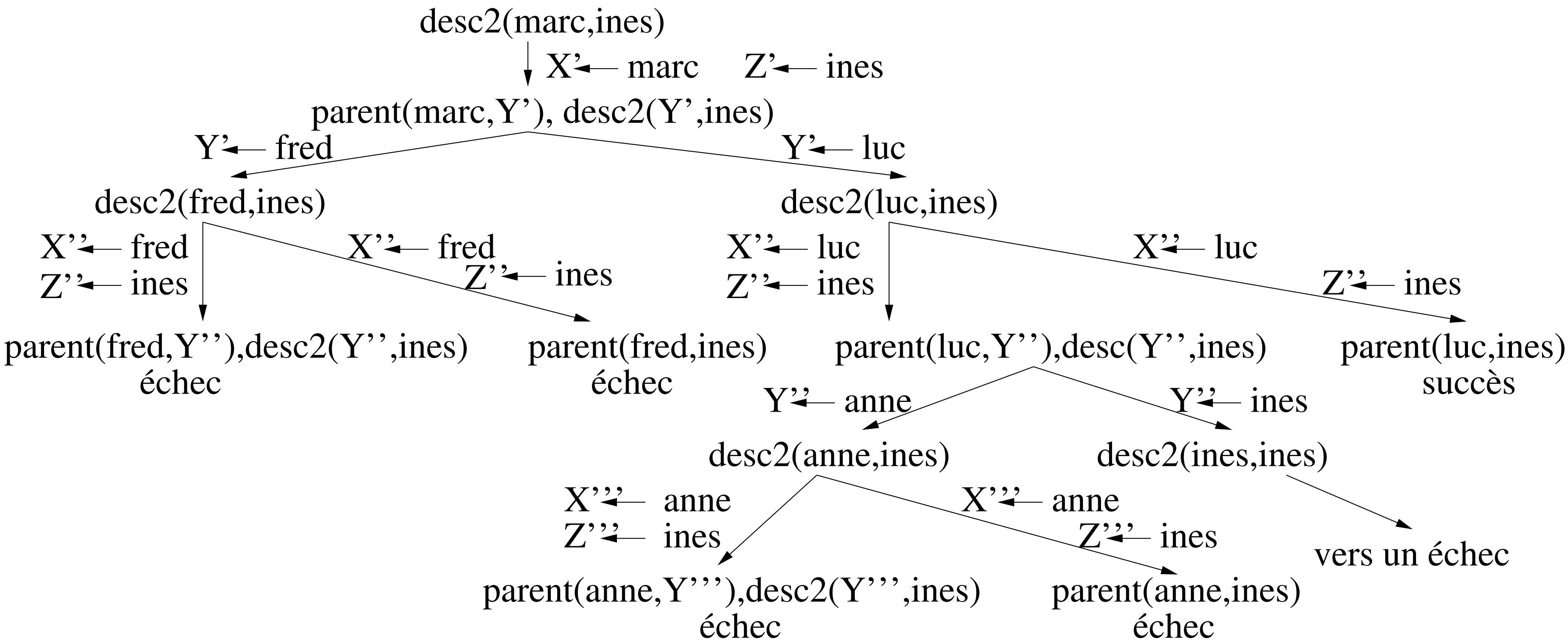
Version 1

```
desc1(X,Z) :- parent(X,Z).
desc1(X,Z) :- parent(X,Y), desc1(Y,Z).
```



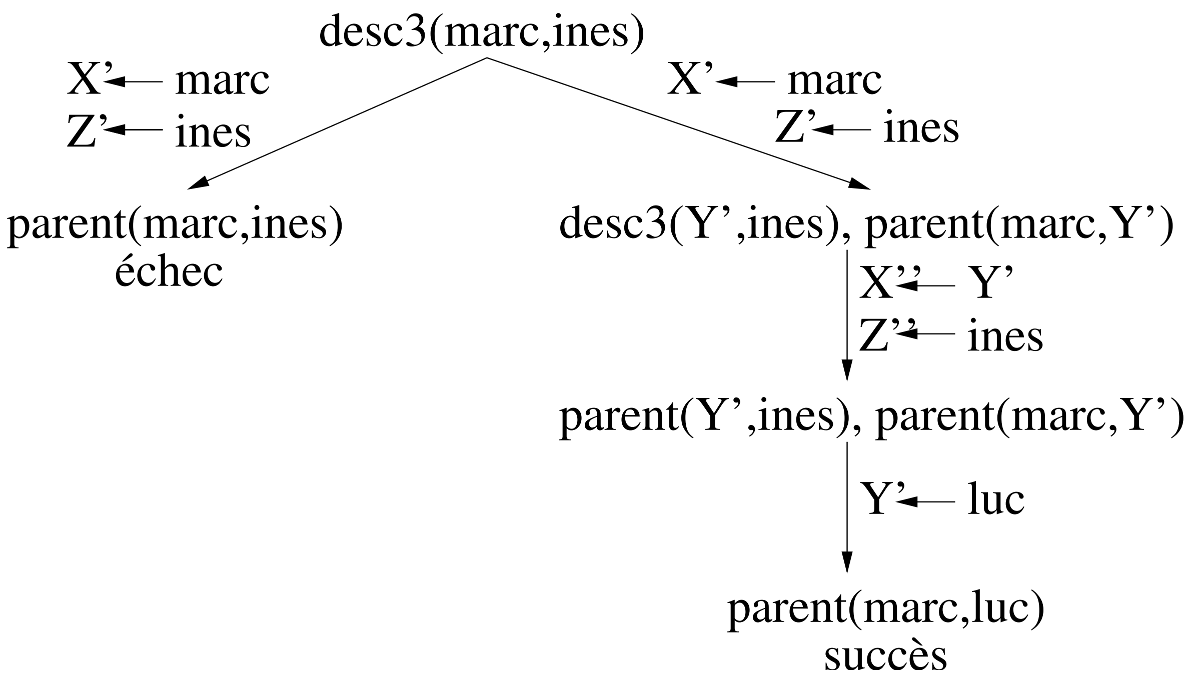
Version 2

```
desc2(X,Z) :- parent(X,Y), desc2(Y,Z).
desc2(X,Z) :- parent(X,Z).
```



Version 3

```
desc3(X,Z) :- parent(X,Z).
desc3(X,Z) :- desc3(Y,Z), parent(X,Y).
```



Version 4

```
desc4(X,Z) :- desc4(Y,Z), parent(X,Y).
desc4(X,Z) :- parent(X,Z).
```

desc4(marc,ines)

X' ← marc
Z' ← ines

desc4(Y',ines),parent(marc,Y')

X'' ← Y'
Z'' ← ines

desc4(Y'',ines),parent(Y',Y''),parent(marc,Y')

X''' ← Y''
Z''' ← ines

desc4(Y''',ines),parent(Y'',Y'''),parent(Y',Y''),parent(marc,Y')

etc.

Arithmétique

Plusieurs opérateurs sont prédéfinis pour des opérations arithmétiques simples.

- `+` l'addition
- `-` la soustraction
- `*` la multiplication
- `/` la division
- `**` la puissance
- `//` la division entière
- `mod` le reste de la division entière

Arithmétique

L'expression `3 + 4` est considérée en Prolog comme une structure d'arité 2 avec symbole fonctionnel + et arguments 3 et 4.

Si on pose la question `?- X = 3 + 4` Prolog répond `X = 3 + 4` !!

Il faut forcer l'évaluation de l'expression avec l'opérateur prédéfini `is`.

Exemples:

```
1 ?- Y is 4 + 3.

Y = 7

Yes
2 ?- X is Y + 1.
ERROR:Arguments are not sufficiently instantiated
```

```
3 ?- Y = 2, X is Y + Y.

Y = 2
X = 4

Yes
```

Arithmétique

Ces exemples montrent que

- La valeur d'une variable est perdue quand on pose une nouvelle question
- Pour pouvoir évaluer une expression avec `is` toutes les variables qui y apparaissent doivent être instanciées.

Il y a des opérateurs pour comparer deux valeurs.

- `X > Y`
- `X < Y`
- `X >= Y`
- `X =< Y`

- `X =:= Y` Les valeurs de `X` et `Y` sont égaux
- `X =\= Y` Les valeurs de `X` et `Y` sont différents

Pour pouvoir évaluer ces expressions les valeurs de `X` et `Y` doivent être possible à calculées. Ces expressions ne causent pas une instanciation des variables !

Attention: L'opérateur = est **différent** de `=:=`

Exemples

```
1 ?- 1 + 3 =:= 3 + 1.
Yes
2 ?- 1 + 3 = 3 + 1.
No
3 ?- 1 + A =:= B + 2.
ERROR:Arguments are not sufficiently instantiated
4 ?- 1 + A = B + 2.
A = 2
B = 1
Yes
5 ?- X = 3 + 2, X < 7.
X = 3+2
Yes
6 ?- Y = 3, X is 3+Y, X + Y < 9.
No
```

Calcul du maximum

```
% Relation max(X,Y,Max) ou Max est le maximum
% des deux entiers X et Y

max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y > X.
```

Listes

- une structure de donnée simple
- utilisée très souvent dans la programmation non numérique
- une séquence d'un certain nombre d'éléments
- par exemple une séquence de `paul,pierre,marc,marie`.
- En Prolog: `[paul, pierre, marc, marie]`
- représentée en interne par un arbre.

Représentation des listes

Une liste est

- soit la liste vide: l'atome `[]`
- soit une structure à deux arguments:
  - le premier élément (la **tête**)
  - le reste de la liste (la **queue**)
- En général, la tête peut être tout terme Prolog, la queue doit être une liste.
- Par exemple, pour `[paul, pierre, marc, marie]` la tête est `paul` et la queue `[pierre, marc, marie]`.
- La tête et la queue sont combinées par un symbole fonctionnel spécial: `.( Tete, Queue)`

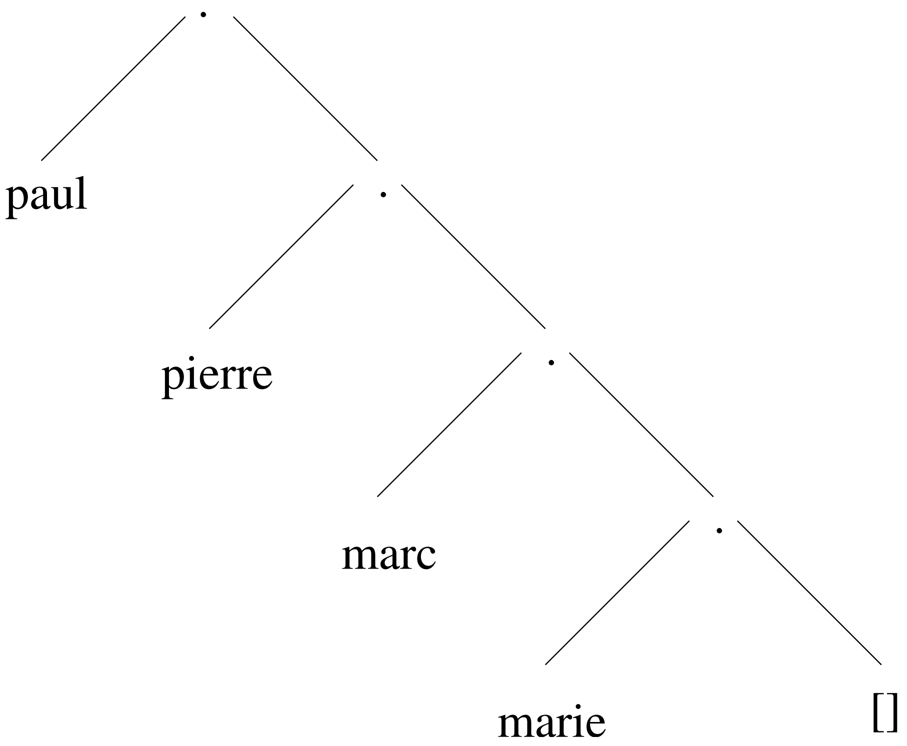
- Puisque la queue doit être une liste, elle est soit vide soit elle a ses propres tête et queue.

- L'exemple s'écrit
  `.( paul, .( pierre, .( marc, .(marie, []))))`

Listes

La liste `[paul, pierre, marc, marie]` s'écrit

comme



- Prolog donne les listes automatiquement dans la forme `[ ...... ]`

Listes

- On peut utiliser la notation `[Tete | Queue]` au lieu de `.( Tete, Queue)`

- On peut énumérer plusieurs éléments avant le `|`.
  Par exemple:

```
[a,b,c] = [a | [b, c]] = [a, b | [c]] =
[a, b, c | []] = [a | [b | [c]]]
```

Opérations sur les listes

Les listes sont des structures. On peut définir des prédicats qui manipulent ces structures.

Appartenance

- On définit la relation d'appartenance comme `membre(X,L)` où X est un objet et L une liste.
- Le but `membre(X,L)` est vrai si X est un élément de L.
- `membre(b,[a, b, c])` est vrai.
- `membre(b,[a, [b, c]])` n'est pas vrai.
- `membre([b, c],[a, [b, c]])` est vrai.

- On peut observer, que
  X est un membre de L, si
  - X est la tête de L, ou
  - X est membre de la queue de L

- En Prolog:

```
membre( X, [X | Queue]).
membre( X, [Tete | Queue]) :-
     membre( X, Queue).
```

Concaténation

- On définit la relation: `concat( L1, L2, L3)` pour concaténer la liste L1 avec la liste L2 pour obtenir L3.

- `concat( [a,b], [c,d], [a,b,c,d])` est vrai

- `concat( [a,b], [c,d], [a,b,a,c,d])` est faux

- Pour définir `concat( L1, L2, L3)` il faut considérer deux cas qui dépendent du premier argument L1
  - Si L1 est vide, alors L2 et L3 doivent être les mêmes.
  - Si L1 est non-vide, alors elle a une tête et une queue. L1 est de la forme `[X | L]`. Dans ce cas concaténer L1 avec L2 donne une liste `[X|L3]` où L3 est la concaténation de L avec L2.

- Ça donne:

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

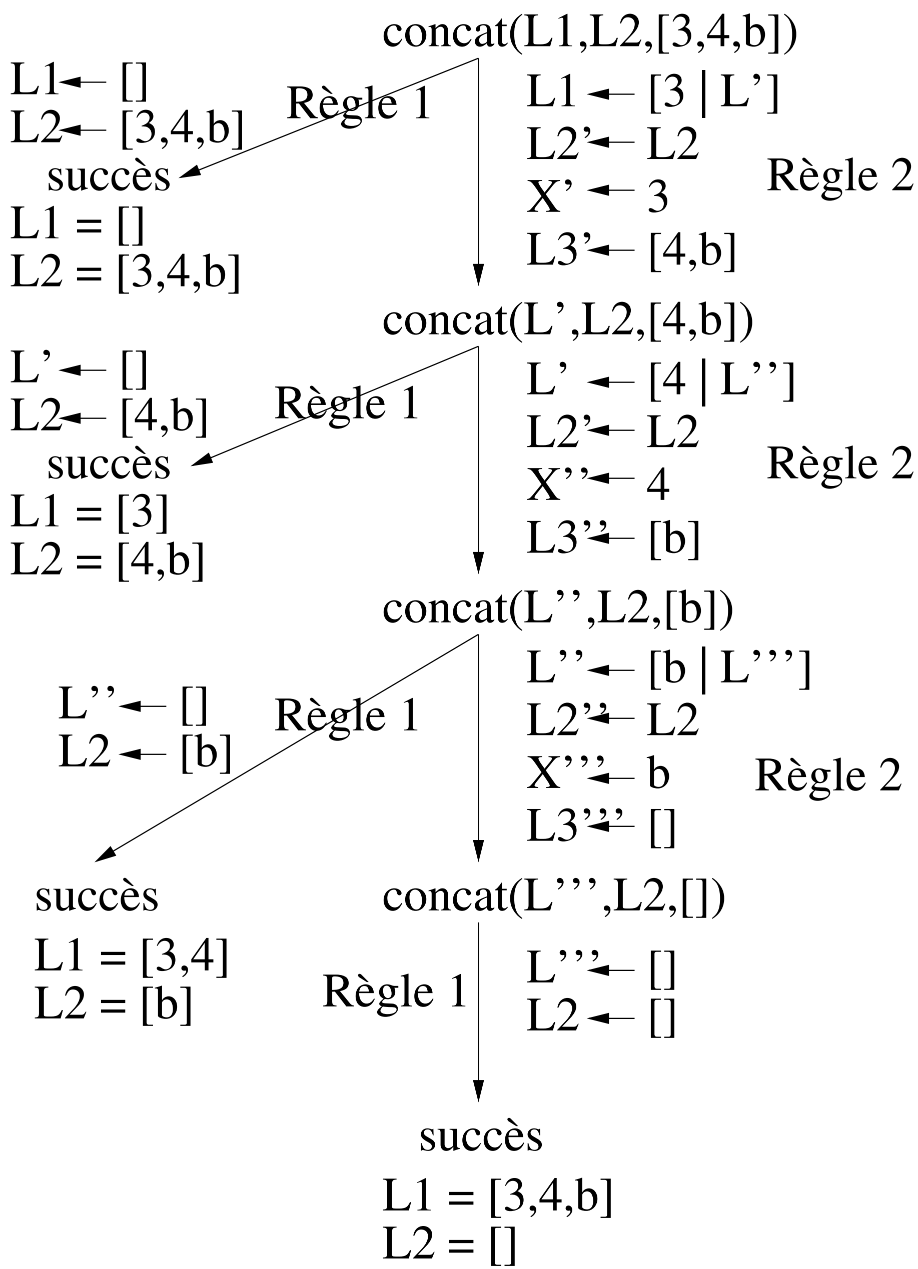
Exemples d'utilisation de concat

```
2 ?- concat([a,b,c],[1,2,3],[a,b,c,1,2,3]).

Yes
3 ?- concat([a,b,c],[1,2,3],L).

L = [a, b, c, 1, 2, 3]

Yes
4 ?- concat([a,[b,c],[[]]],[a,[]],L).

L = [a, [b, c], [[]], a, []]
Yes
```

```
5 ?- concat(L1,L2,[3,4,b]).
L1 = []
L2 = [3, 4, b] ;

L1 = [3]
L2 = [4, b] ;

L1 = [3, 4]
L2 = [b] ;

L1 = [3, 4, b]
L2 = [] ;
No
```

Exemple d'arbre de preuve

```
concat([],L,L). % Règle 1
concat([X|L],L2,[X|L3]):-concat(L,L2,L3).% Règle 2
```


concat(L1,L2,[3,4,b])
L1← []
L2← [3,4,b]
succès
L1 = []
L2 = [3,4,b]
Règle 1
L1 ← [3 | L']
L2' ← L2
X' ← 3
L3' ← [4,b]
Règle 2
concat(L',L2,[4,b])
L' ← []
L2 ← [4,b]
succès
L1 = [3]
L2 = [4,b]
Règle 1
L' ← [4 | L'']
L2' ← L2
X'' ← 4
L3'' ← [b]
Règle 2
concat(L'',L2,[b])
L'' ← []
L2 ← [b]
Règle 1
L'' ← [b | L''']
L2'' ← L2
X''' ← b
L3''' ← []
Règle 2
succès
L1 = [3,4]
L2 = [b]
concat(L''',L2,[])
Règle 1
L''' ← []
L2 ← []
succès
L1 = [3,4,b]
L2 = []

Exemple d'utilisation de concat (Suite)

```
6 ?- concat(Avant,[mai | Apres],
[ja,fe,ma,av,mai,jui,jul,ao,se,oc,no,de]).

Avant = [ja, fe, ma, av]
Apres = [jui, jul, ao, se, oc, no, de]

Yes
7 ?- concat(_,[Mois1,mai,Mois2 | _],
[ja,fe,ma,av,mai,jui,jul,ao,se,oc,no,de]).

Mois1 = av
Mois2 = jui

Yes
8 ?- L1 = [a,b,e,f,g,g,g,h,a],
|     concat(L2,[g,g,g | _],L1).
```

```
L1 = [a, b, e, f, g, g, g, h, a]
L2 = [a, b, e, f]

Yes
```

Appartenance en utilisant concat

On peut définir la relation `membre` aussi avec `concat`.

```
membre1( X, L) :-
   concat( L1, [X | L2], L).
```

ou

```
membre1( X, L) :-
   concat( _, [X | _], L).
```

Ajouter un élément

```
ajouter( X, L, [X | L]).
```

On peut directement utiliser `[X | L]`.

Exemple “traduction”

```
traduitmot(he,il).
traduitmot(she,elle).
traduitmot(i,je).

traduitmot(eats,mange).
traduitmot(eat,mange).

traduitmot(bread,pain).

traduitphrase([],[]).
traduitphrase([X | Reste], [Y | ResReste]) :-
     traduitmot( X, Y),
     traduitphrase( Reste, ResReste).
```

Exemple "traduction" (Suite)

```
2 ?- traduitphrase([i,eat,bread],L).

L = [je, mange, pain]

Yes
3 ?- traduitphrase(L,[il,mange,pain]).

L = [he, eats, bread] ;

L = [he, eat, bread] ;

No
```

```
4 ?- traduitphrase(L,L1).

L = []
L1 = [] ;

L = [he]
L1 = [il] ;

L = [he, he]
L1 = [il, il] ;
```

Enlever un élément

- Enlever l'élément X d'une liste L peut être réalisé par un prédicat `enlever(X,L,L1)` où L1 est la liste L sans l'élément X

- Il y a deux cas:
  - Si X est la tête de la liste, alors le résultat est la queue de la liste.
  - Si X est dans la queue, alors il est enlevé de la queue.

- En Prolog:

```
enlever( X, [X | Queue], Queue).

enlever( X, [Y | Queue], [Y | Queue1]) :-
    enlever(X, Queue, Queue1).
```

- Si X apparaît plusieurs fois dans la liste, des retours en arrière successives donnent toutes les possibilités.

- `enlever` échoue si X n'appartient pas à la liste.

Exemples d'utilisation pour enlever

```
2 ?- enlever( a, [a, b, b, a, b], L).
L = [b, b, a, b] ;

L = [a, b, b, b] ;

No
3 ?- enlever( X, [a, c, c], L).
X = a
L = [c, c] ;

X = c
L = [a, c] ;

X = c
L = [a, c] ;
No
```

```
4 ?- enlever( a, L, [c, d, e]).
L = [a, c, d, e] ;

L = [c, a, d, e] ;

L = [c, d, a, e] ;

L = [c, d, e, a] ;

No
```

Insérer un élément / membre

- Avec enlever on peut définir `inserer(X,L,L1)` qui insère l'élément X quelque part dans la liste L en donnant L1.

```
inserer(X,L,L1) :-
    enlever(X,L1,L).
```

- On peut aussi définir la relation membre en utilisant `enlever`. Un élément X est membre d'une liste si on peut l'enlever.

```
membre2( X, L) :-
    enlever( X, L, _).
```

Sousliste

- On définit un prédicat `sousliste( S, L)` qui est vrai si S est une sousliste (contiguë) de L.

- Exemples: `sousliste( [a,b], [c,a,b,d])` est vrai et `sousliste( [a,b], [c,a,d,b])` est faux.

- S est une sousliste de L, si
  - L peut être décomposée dans deux listes L1 et L2, et
  - L2 peut être décomposée dans deux listes, S et L3.

- En Prolog:

```
sousliste( S, L) :-
    concat( L1, L2, L),
    concat( S, L3, L2).
```

Exemples d'utilisation

```
2 ?- sousliste([a,b],[b,a,b,c]).
Yes
3 ?- sousliste(X, [a, b]).

X = [] ;

X = [a] ;

X = [a, b] ;

X = [] ;

X = [b] ;

X = [] ;
No
```

```
4 ?- sousliste([a,b],L).

L = [a, b|_G324] ;

L = [_G320, a, b|_G330] ;

L = [_G320, _G326, a, b|_G336] ;

L = [_G320, _G326, _G332, a, b|_G342]
```

Permutations

- Nous définissons un prédicat `permute/2` avec deux arguments.
- Le premier argument est une liste donnée L1 et le deuxième une variable L2, dans laquelle par retours en arrière successifs il y aura comme solutions toutes les listes qui sont des permutations de L.
- Il y a deux cas (dépendant de L1):
  - Si L1 est vide, L2 l'est aussi.
  - Si L1 n'est pas vide, elle a la forme `[X | L]` et une permutation s'obtient d'abord en permutant L (ce qui donne L3) est en suite en insérant X quelque part en L3.
- en PROLOG:

```
permute([],[]).
permute([ X | L], L2) :-
   permute( L, L3),
   inserer( X, L3, L2).
```

Exemples d'utilisation

```
2 ?- permute([a,b,c],L).

L = [a, b, c] ;

L = [b, a, c] ;

L = [b, c, a] ;

L = [a, c, b] ;

L = [c, a, b] ;

L = [c, b, a] ;
No
```

```
3 ?- permute(L,[a,b,c]).
L = [a, b, c] ;

L = [b, a, c] ;

L = [c, a, b] ;

L = [a, c, b] ;

L = [b, c, a] ;

L = [c, b, a] ;
ne s'arrête pas.
```

Longueur d'une liste

- Nous définissons le prédicat `longueur(Liste,N)` qui étant donné une liste `Liste` instancie N avec sa longueur.

- Il y a deux cas:
  - Si `Liste` est vide, sa longueur est 0
  - Si `Liste` n'est pas vide, elle a la forme `[Tete | Queue]` et sa longueur est la longueur de la liste `Queue` plus 1.

- En Prolog:

```
longueur( [], 0).
longueur( [_ | Queue], N) :-
   longueur( Queue, N1),
   N is 1 + N1.
```

```
2 ?- longueur([a,b,c],N).
N = 3
Yes
3 ?- longueur(L,3).
L = [_G281, _G284, _G287] ;
s'arrête pas....
```

- L'ordre des deux buts dans la deuxième clause de longueur ne peut pas être changé.

Longueur d'une liste (Autres versions)

```
longueur1( [], 0).
longueur1( [_ | Queue], N) :-
   longueur1( Queue, N1),
   N = 1 + N1.

longueur2( [], 0).
longueur2( [_ | Queue], N) :-
   N = 1 + N1,
   longueur2( Queue, N1).

longueur3( [], 0).
longueur3( [_ | Queue], 1 + N1) :-
   longueur2( Queue, N1).
```

```
3 ?- longueur1([a,b,c],N).
N = 1+ (1+ (1+0))
Yes
4 ?- longueur2([a,b,c],N).
N = 1+ (1+ (1+0))
Yes
5 ?- longueur3([a,b,c],N).
N = 1+ (1+ (1+0))
6 ?- longueur1(L,2).
ne s'arrête pas....
```