Une courte introduction à prolog

PP

25 mars 2001

Ceci n'est pas un cours de prolog, c'est juste une très courte introduction à prolog. On essaie de montrer les points essentiels du langage. On commence par une approche naïve du langage vu comme une base de données intelligentes. On voit ensuite d'autres aspects plus techniques (cut, listes, ...) qui doivent être maîtrisés pour pouvoir réellement programmer en prolog. On va alors bien au-delà du simple aspect base de données de prolog pour atteindre les capacités d'un système expert reposant sur la logique des prédicats et finalement, celles d'un langage de programmation universel.

Toutes les manipulations se font en gprolog.

1 Quelques exemples simples en prolog

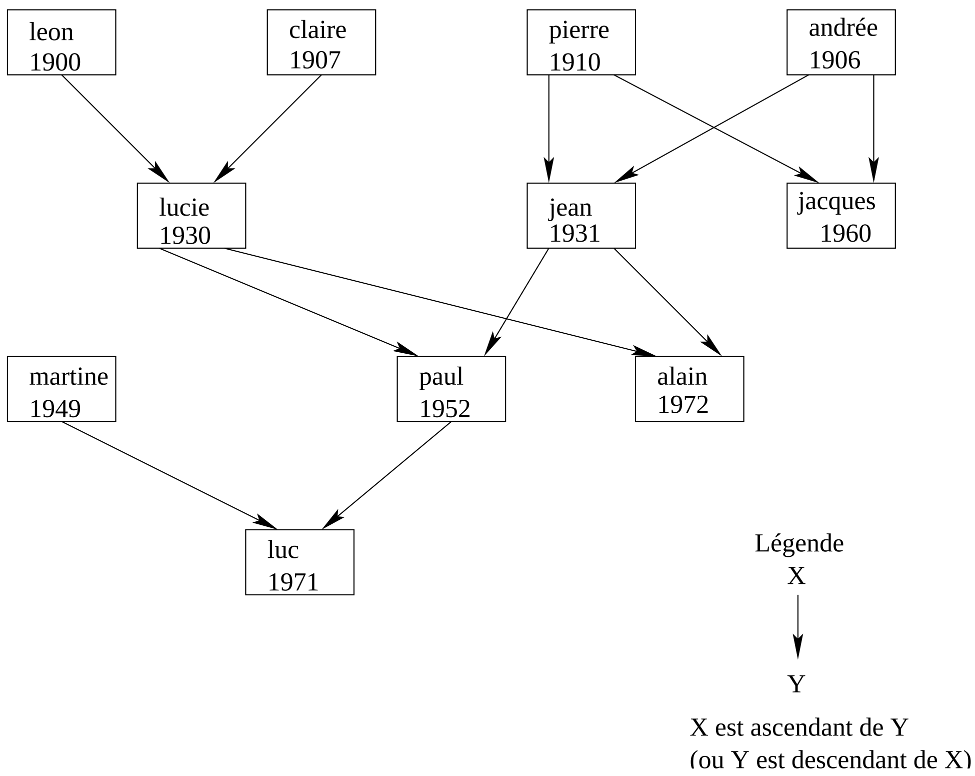
1.1 Une histoire de famille

On veut représenter la hiérarchie familiale indiquée à la figure 1.1.

Pour cela, on doit tout d'abord déclarer l'existence et le sexe des protagonistes :

```
masculin(jean).
masculin(paul).
masculin(alain).
masculin(pierre).
masculin(leon).
masculin(luc).
masculin(jacques).

feminin(lucie).
feminin(claire).
feminin(andree).
feminin(martine).
```



On peut alors demander si untel est un homme par une interrogation du genre : `masculin(roger).`

Celle-ci produira la réponse `no` car roger n'est pas connu dans la base de connaissances. Par contre, si on demande `masculin(roger).`, la réponse sera `yes`.

On peut aussi demander quelles sont les personnes de sexe féminin déclarées dans la base de connaissance : `femme(X).`

La réponse sera alors :

```
X = lucie ?

X = claire ?

X = andree ?

X = martine

yes
```

Les variables sont des identificateurs dont la première lettre est une majuscule (comme `Xjfshdjfhsdjkhfs`).

Ensuite, on peut expliciter les liens de filiations :

```
estPere(jean,paul).
estPere(jean,alain).
estPere(pierre,jean).
estPere(leon,lucie).
estPere(paul,luc).
estPere(pierre,jacques).

estMere(claire,lucie).
estMere(andree,jean).
estMere(andree,jacques).
estMere(lucie,paul).
estMere(martine,luc).
estMere(lucie,alain).
```

Par convention, le premier prédicat indique que jean est le père de paul.

On peut alors demander si paul est le père de jean : `estPere(paul,jean).` qui provoque la réponse `no`. On peut aussi demander quels sont les enfants de andrée : `estMere(andree,X).` qui provoque la réponse :

```
X = jean ?

X = jacques

yes
```

On peut aussi composer plusieurs prédicats dans une interrogation, par exemple en demandant quels sont les personnes qui sont père d'une fille : `estPere(X,Y), feminin(Y).` qui provoque la réponse : `X = leon, Y = lucie`.

On peut ensuite définir la notion de grand-père en indiquant que X est le grand-père de Y si X est le père d'un certain Z lui-même père ou mère de Y :

```
estGrandPere(X,Y):- estPere(X,Z), estPere(Z,Y).
estGrandPere(X,Y):- estPere(X,Z), estMere(Z,Y).
```

On peut alors demander si jean est le grand-père de alain par `estGrandPere(jean,alain).` qui provoque la réponse `no`, puis demander qui est le grand-père de luc : `estGrandPere(X,luc).` qui produit `X = jean`, puis de qui jean est le grand-père : `estGrandPere(jean,X).` qui produit `X = luc`, puis qui est le grand-père de qui : `estGrandPere(X,Y).` qui produit 5 réponses.

1.2 Simulation d'un circuit logique

On souhaite simuler un circuit logique binaire.

```
circuit(A,B,C,S) :- et(A,B,X), ou(B,C,Y), exor(X,Y,S).

et(1,1,1).
et(0,1,0).
et(1,0,0).
et(0,0,0).

ou(1,1,1).
ou(0,1,1).
ou(1,0,1).
ou(0,0,0).

exor(1,1,0).
exor(0,1,1).
exor(1,0,1).
exor(0,0,0).
```

– quelle est la valeur de la sortie du circuit pour des valeurs d'entrée données ?

```
| ?- circuit(1,1,1,X).

X = 0 ?

no
```

– quelles valeurs faut-il en entrée pour avoir 1 en sortie ?

```
| ?- circuit(X,Y,Z,1)
X = 0
Y = 1
Z = 1 ?

X = 0
Y = 1
Z = 0 ?
```

```
X = 1
Y = 0
Z = 1 ?

X = 0
Y = 0
Z = 1 ?

no
```

- quelles valeurs faut-il en entrée quand A=C pour avoir 1 en sortie ?

```
| ?- circuit(X,Y,X,1).

X = 0
Y = 1 ?

X = 1
Y = 0 ?

no
```

- quelles valeurs faut-il en entrée quand A=C pour avoir 1 en sortie si on ne s'intéresse pas à B ?

```
| ?- circuit(X,_,X,1).

X = 0 ?

X = 1 ?

no
```

1.3 Un peu d'arithmétique

On souhaite écrire des prédicats permettant de résoudre des problèmes d'arithmétique entière tels que : combien font 3 et 2 ? quels couples de nombres entiers ont leur somme qui fait 6 ? que faut-il ajouter à 5 pour obtenir 8 ? mais aussi des systèmes d'équations comme

$$\begin{cases} a+b+c=10 \\ a+2+c=8 \end{cases}$$

pour les entiers compris entre 0 et 10.

```
successeur(0,1).
successeur(1,2).
successeur(2,3).
successeur(3,4).
successeur(4,5).
successeur(5,6).
successeur(6,7).
successeur(7,8).
successeur(8,9).
successeur(9,10).

somme(X,1,Z) :- successeur(X,Z).
somme(X,Y,Z) :-
    successeur(Y1,Y),
    somme(X,Y1,Z1),
    successeur(Z1,Z).

somme3(A,B,C,D) :-
    somme(A,B,X),
    somme(X,C,D).
```

On ne considère que les entiers compris entre 0 et 10. Si on veut en prendre d'autres en considération, il suffit d'ajouter des prédicats `successeur`.

Le prédicat `somme(X,Y,Z)` signifie que `Z` est la somme de `X` et `Y`.

Le prédicat `somme3(A,B,C,D)` signifie que `D` est la somme de `A`, `B` et `C`.

– combien font 3 et 2 ?

```
| ?- somme(3,2,X).

X = 5 ?

no
```

- quels couples de nombres entiers ont leur somme qui fait 6 ?

```
| ?- somme(X,Y,6).

X = 5
Y = 1 ?

X = 4
Y = 2 ?

X = 3
Y = 3 ?

X = 2
Y = 4 ?

X = 1
Y = 5 ?

X = 0
Y = 6 ?

no
```

- que faut-il ajouter à 5 pour obtenir 8 ?

```
| ?- somme(5,X,8).

X = 3 ?

no
```

- résoudre le système d'équations plus haut :

```
| ?- somme3(A,B,C,10),somme3(A,2,C,8).

A = 0
B = 4
C = 6 ?
```

```
A = 1
B = 4
C = 5 ?

A = 2
B = 4
C = 4 ?

A = 3
B = 4
C = 3 ?

A = 4
B = 4
C = 2 ?

A = 5
B = 4
C = 1 ?

(10 ms) no
```

2 Les listes

Dans cette section, on étudie différents mécanismes plus subtils qui permettent de tirer toute la puissance du moteur d'inférences prolog. On s'intéresse aux traitements de listes et on montre l'utilité et l'effet du cut sur des exemples.

2.1 Les listes

Une liste est une suite ordonnées d'éléments, chaque élément pouvant être une valeur (nombre, symbole), ou une liste. On note une liste à l'aide d'un `[` en début de liste et d'un `]` en fin de liste. Par exemple, `[2,1,3]` est la liste composée des nombres 2, 1 et 3 ; une `,` sépare chaque élément du suivant. `[2,[5,3],6,1]` est une liste dont le deuxième élément est une liste de deux nombres.

Très souvent, on traite une liste en considérant son premier (ou ses premiers) élément puis le reste. Pour cela, la notation `[X | L]` est très utile : `X` est le premier élément de la liste (que ce soit un nombre, un symbole ou même une liste) et `L` est la liste composée des autres éléments. Ainsi, l'unification de `[2,4,3,1]` avec `[X | L]` effectue la liaison `X = 2` et `L = [4,3,1]`. Si les trois premiers éléments d'une liste nous intéressent, on peut donc écrire `[A, B, C | L]`. On peut aussi utiliser le caractère `_` pour indiquer qu'un élément de la liste ne doit pas être lié. Attention, dans la notation `[X | L]`, `L` est une liste ; aussi, la liste `[1,2,3]` peut s'écrire aussi `[1|[2,3]]`, mais pas `[1|2,3]`.

Une liste qui ne contient pas d'élément se nomme la « liste vide »et se note `[]`.

2.2 Membre

Il s'agit d'écrire un prédicat qui détermine si une valeur appartient à une liste. Si le traitement de listes est l'apanage du langage Lisp, on montre ici que prolog, grâce à son asopect déclaratif, permet d'exprimer simplement des traitements qui nécessiteraient plusieurs fonctions spécialisées en Lisp.

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

– une valeur est-elle dans une liste ?

  ```
  | ?- membre(1,[2,1,3]).

  true ?

  no
  | ?- membre(1,[2,5,3]).

  no
  ```

  On se borne ici à la fonction à laquelle nous sommes habitués en Lisp ;

– quelles valeurs sont contenues dans une liste ?

  ```
  | ?- membre(X,[2,1,3]).

  X = 2 ?

  X = 1 ?

  X = 3 ?

  no
  ```

  Avec le même programme prolog, on effectue ce nouveau type de traitement. En Lisp, il faut une autre fonction et on doit appliquer l'une ou l'autre explicitement ;

– quelle liste contient une certaine valeur ?

```
| ?- membre(1,[2,X,3]).

X = 1 ?

no
| ?- membre(1,X).

X = [1|_] ?

X = [_,1|_] ?

X = [_,_,1|_] ?

X = [_,_,_,1|_] ?

X = [_,_,_,_,1|_] ?

X = [_,_,_,_,_,1|_] ?

X = [_,_,_,_,_,_,1|_] ?

X = [_,_,_,_,_,_,_,1|_] ?

X = [_,_,_,_,_,_,_,_,1|_] ?

yes
| ?- membre(2,[3,X,Y,Z,5]).

X = 2 ?

Y = 2 ?

Z = 2 ?

no
```

Là, c'est autrement plus fort que ce que l'on a l'habitude de faire en Lisp !

2.3 Concaténation

Écrire un prédicat qui prend trois listes : la troisème est la concaténation des deux premières. On peut l'écrire très simplement comme suit :

```
concat(L,[],L).
concat([],L,L).
concat([X|Y],L,[X|L12]) :-
    concat(Y,L,L12).
```

Mis à part les deux cas triviaux où l'une des deux listes est vide auquel cas la troisième est égale à la liste non vide, le cas général s'exprime en disant que la concaténation de deux listes est égale au premier élément de la première liste suivi de la concaténation du reste de la première liste avec la deuxième.

– utilisation standard : concaténation de deux listes :

  ```
  | ?- concat([1,2],[3],L).

  L = [1,2,3]

  yes
  ```

– quelle liste faut-il concaténer à une liste pour en obtenir une troisième ? Il suffit de le demander :

  ```
  | ?- concat([1,2],L,[1,2,3,4,5]).

  L = [3,4,5]

  yes
  | ?- concat(L,[2,3],[1,2,3]).

  L = [1] ?

  no
  ```

– quelles listes permettent d'obtenir une troisème ?

  ```
  | ?- concat(L,H,[1,2,3,4,5]).

  H = []
  ```

```
L = [1,2,3,4,5] ?

H = [1,2,3,4,5]
L = [] ?

H = []
L = [1,2,3,4,5] ?

H = [2,3,4,5]
L = [1] ?

H = []
L = [1,2,3,4,5] ?

H = [3,4,5]
L = [1,2] ?

H = []
L = [1,2,3,4,5] ?

H = [4,5]
L = [1,2,3] ?

H = []
L = [1,2,3,4,5] ?

H = [5]
L = [1,2,3,4] ?

H = []
L = [1,2,3,4,5] ?

H = []
L = [1,2,3,4,5] ?

no
```

2.4 Renverse

Écrire un prédicat qui renverse les éléments d'une liste.

```
renverse([],[]).
renverse([X|L],M) :- renverse(L,N), concat(N,[X],M).
```

pas génial :

```
| ?- renverse([1,2,3],L).

L = [3,2,1]

yes
| ?- renverse(L,[1,2,3]).

L = [3,2,1] ?

Prolog interruption (h for help) ?
execution aborted
```

2.5 Préfixe

Écrire un prédicat qui détermine si une liste est préfixe d'une autre.

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

– utilisation standard : une liste est-elle préfixe d'une autre ?

```
| ?- prefixe([1,2],[1,2,3,4]).

yes
| ?- prefixe([1,2],[1]).

no
| ?- prefixe([1,2],[1,2]).

yes
| ?- prefixe([2,3],[1,2,3,4]).
```

```
no
```

– quelles sont les listes préfixes d'une certaine liste ?

```
| ?- prefixe(L,[1,2,3,4]).

L = [] ?

L = [1] ?

L = [1,2] ?

L = [1,2,3] ?

L = [1,2,3,4] ?

no
```

– quelles sont les listes dont une certaine liste est préfixe ?

```
| ?- prefixe([1,2,3,4],L).

L = [1,2,3,4|_]

yes
```

2.6 Sous-liste

Écrire un prédicat qui détermine si une liste est sous-liste d'une autre.

```
sousListe([],[_]).
sousListe(_,[]) :- fail.
sousListe(L,M) :- prefixe(L,M).
sousListe(L,[_|M]) :- sousListe(L,M).
```

On remarquera l'utilisation du prédicat prédéfini `fail` qui est tout le temps faux et provoque donc toujours un échec.

– utilisation standard : une liste est-elle sous-liste d'une autre ?

```
| ?- sousListe([1,2],[2,3]).

no
| ?- sousListe([1,2],[1,2,3]).

true ?

no
| ?- sousListe([1,2],[2,3,1,2,1,1,2,3]).

true ?

no
```

- quelles sont les listes qui ont une certaine liste comme sous-liste ?

```
| ?- sousListe([1,2],L).

L = [1,2|_] ?

L = [_,1,2|_] ?

L = [_,_,1,2|_] ?

L = [_,_,_,1,2|_] ?

yes
```

Note : on arrête l'énumération en tapant retour-chariot et on la poursuit en tapant `;`.

- quelles sont les listes sous-liste d'une liste donnée ?

```
| ?- sousListe(L,[1,2,3]).

L = [] ?

L = [1] ?

L = [1,2] ?
```

```
L = [1,2,3] ?

L = [] ?

L = [2] ?

L = [2,3] ?

L = [] ?

L = [] ?

L = [3] ?

L = [] ?

no
```

Dans le dernier exemple, les répétitions de la liste nulle devraient être évitées. On discutera de ce point plus loin (voir section 3).

2.7 Longueur

Écrire un prédicat qui donne la longueur d'une liste.

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

– utilisation standard : calculer la longueur d'une liste donnée :

```
| ?- longueur([1,2,3],X).

X = 3

yes
```

– une liste est-elle d'une certaine longueur ?

```
| ?- longueur([1,2,3],4).
```

```
no
| ?- longueur([1,2,3],3).

yes
```

– quelles sont les listes d'une longeur donnée ?

```
| ?- longueur(L,3).

L = [_,_,_] ?

yes
```

– quelles sont les listes d'une longueur quelconque ?

```
| ?- longueur(L,X).

L = []
X = 0 ?

L = [_]
X = 1 ?

L = [_,_]
X = 2 ?

L = [_,_,_]
X = 3 ?

L = [_,_,_,_]
X = 4 ?

L = [_,_,_,_,_]
X = 5 ?

L = [_,_,_,_,_,_]
X = 6 ?

L = [_,_,_,_,_,_,_]
```

```
X = 7 ?

yes
```

2.8 n^e

Écrire un prédicat qui donne le n^e élément d'une liste.

```
nieme([],_,_) :- fail.
nieme([X|_],1,X).
nieme([_|L],N,Y) :- N1 is N-1, nieme(L,N1,Y).
```

Dans la première règle, le prédicat prédéfini `fail` indique que le n^e élément d'une liste vide n'existe pas.

– quel est le n^e élément d'une liste ?

```
| ?- nieme([1,2,3],2,X).

X = 2 ?

no
```

– le n^e élément d'une liste a-t-il une valeur donnée ?

```
| ?- nieme([1,2,3],2,3).

no
| ?- nieme([1,2,3],2,2).

true ?

no
```

– quelles sont les listes dont le n^e élément a une certaine valeur ?

```
| ?- nieme(L,2,2).

L = [_,2|_] ?

yes
```

Ici, si on demande une autre solution , prolog se plante. On verra à la section 3 comment résoudre ce problème.

2.9 Minimum, maximum et somme des éléments d'une liste

Écrire des prédicats qui donnent respectivement le minimum, le maximul et la somme des éléments d'une liste.

```
minimum([X],X).
minimum([X|L],X) :- minimum(L,X1), X =< X1, !.
minimum([X|L],X1) :- minimum(L,X1), X > X1.
```

- quelle est le minimum d'une liste ?

```
| ?- minimum([3,2,4,1,6],X).

X = 1 ?

no
```

- le minimum est-il une certaine valeur ?

```
| ?- minimum([3,2,4,1,6],4).

no
```

- liste dont le minimum est donné ?

```
| ?- minimum(L,4).

L = [4] ?

yes
```

Si on demande d'autres solutions, il y a plantage [1].

Quant on sait écrire le minimum, c'est bien entendu un jeu d'enfant d'écrire le maximum. Pour la somme des éléments d'une liste, on écrira :

```
somme([],0).
somme([X|L],S) :- somme(L,Y), S is X+Y.
```

1. le prédicat prédéfini qui fait la même chose, `min_list`, possède exactement le même comportement

3 Le cut

Il est temps d'introduire un prédicat prédéfini très important en prolog, le cut. Celui-ci effectue un élagage de l'arbre de résolution de prolog. Son utilisation va nous permettre de résoudre des problèmes rencontrés jusqu'alors.

Il arrive que l'on ne souhaite trouver qu'une seule solution et non pas toutes les solutions résultant d'une interrogation. Il arrive aussi que l'on sache qu'il n'existe qu'une seule solution à une interrogation. Dans ces deux cas, chercher d'autres solutions après avoir trouvé la première est une perte de temps. Par ailleurs, dans certains cas, cela peut même entraîner un plantage (voir l'exemple nieme ci-dessous).

3.1 Principe de fonctionnement du cut

L'action du cut est la suivante :

1. ! est toujours vrai ;
2. lors d'un backtracking, si prolog arrive à un nœud dont la liste des sous-buts commence par !, alors prolog remonte jusqu'au nœud correspondant au but précédent le but qui a déclenché l'appel de la règle contenant le !.

Cela s'explique bien sur quelques exemples simples :

```
p(a).
p(b).
p(c).

p2(aa).
p2(bb).
p2(cc).

r(X) :- p(X).

r1(X) :- !, p(X).
r2(X) :- p(X), !.

q1(X,Y) :- r(X), !, p2(Y).

q2(X,Y) :- r(X), p2(Y), !.

q3(X,Y) :- !, r(X), p2(Y).
```

L'interrogation `r(X).` fournit les 3 réponses `X = a ; X = b ; X = c`.

L'interrogation `r1(X).` fournit les 3 réponses `X = a ; X = b ; X = c`.

L'interrogation `r2(X).` fournit la réponse `X = a`.

L'interrogation `p(X),r1(Y).` fournit 9 réponses :
$(X, Y) \in \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c)\}$.

L'interrogation `p(X),r2(Y).` fournit 3 réponses : $(X, Y) \in \{(a, a), (b, a), (c, a)\}$.

L'interrogation `q1(X,Y).` fournit les 3 réponses
`X = a, Y = aa ; X = a, Y = bb ; X = a, Y = cc`.

L'interrogation `q2(X,Y).` fournit la réponse `X = a, Y = aa`.

L'interrogation `q3(X,Y).` fournit les 9 réponses formés de tous les éléments du produit cartésien des ensembles $\{a, b, c\} \times \{aa, bb, cc\}$.

Toutes ces interrogations renvoient `yes`.

3.2 membre1

Le prédicat `membre` écrit plus haut n'est pas intéressant si on veut l'utiliser pour tester la présence d'une valeur dans une liste. En effet, une fois que l'on a trouvé la valeur recherchée, ce n'est pas la peine de continuer et de la trouver plusieurs fois de surcroît. Il y a des cas où cela est même mauvais. On peut alors écrire très facilement cette nouvelle version avec un cut comme suit :

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

La première règle signifie : quand on a trouvé la valeur recherchée, on arrête là. On peut comparer les deux prédicats `membre` et `membre1` :

```
| ?- membre(3,[1,2,3,4,3,2,3,5]).

true ?

true ?

true ?

no
| ?- membre1(3,[1,2,3,4,3,2,3,5]).

yes
```

Notons qu'en plus, `membre1` répond bien `yes` alors que `membre` répond `no` parce que la valeur recherchée n'est pas en dernière position dans la liste.

3.3 Retrait de la première occurence d'une valeur dans une liste

Écrire un prédicat qui retire la première occurence d'une valeur donnée dans une liste.

```
retire1(_,[],[]).
retire1(X,[X|L],L) :- !.
retire1(X,[Y|L],[Y|M]) :- retire1(X,L,M).
```

Le cut est indispensable ici pour ne retirer que la première occurence de la liste.

3.4 Retour sur sousListe

Plus haut (voir section 2.6), on avait constaté que le prédicat sousListe donne parfois des réponses peut intéressantes. Cela va mieux en mettant un cut dans la première règle :

```
sousListe([],[_]) :- !.
sousListe(_,[]) :- fail.
sousListe(L,M) :- prefixe(L,M).
sousListe(L,[_|M]) :- sousListe(L,M).
```

On obtient alors :

```
| ?- sousListe(L,[1,2,3]).

L = [] ?

L = [1] ?

L = [1,2] ?

L = [1,2,3] ?

L = [] ?

L = [2] ?
```

```
L = [2,3] ?

L = []

yes
```

mais ce n'est pas encore parfait...

3.5 Retour sur nieme

Comme on l'a vu plus haut (section 2.8), le prédicat `nieme` tel qu'il a été écrit plus haut n'est pas parfait. On peut améliorer les choses en y ajoutant un cut comme suit :

```
nieme(_,N,_) :- N =< 0, !, fail.
nieme([],_,_) :- fail.
nieme([X|_],1,X) :- !.
nieme([_|L],N,Y) :- N1 is N-1, nieme(L,N1,Y).
```

En effet, quand on a trouvé le n^e élément, ce n'est plus la peine de le chercher. Dès lors, l'interrogation :

```
| ?- nieme(L,2,2).

L = [_,2|_] ?

yes
```

ne provoque plus de plantage si on demande une autre solution.

3.6 retireDoublon

Écrire un prédicat qui retire les doublons d'une liste et la transforme donc en ensemble.

```
retireDoublon([],[]).
retireDoublon([X|L],M) :-
    membre(X,L),
    retireDoublon(L,M), !.
retireDoublon([X|L],[X|M]) :-
    retireDoublon(L,M).
```

Attention à l'utilisation du cut.

- ```
  | ?- retireDoublon([1,2,3],L).

  L = [1,2,3]

  yes
  | ?- retireDoublon([1,2,1,3,2,1,3],L).

  L = [2,1,3]

  yes
  ```

3.7 Union, intersection et complémentaire de listes

Écrire des prédicats qui donnent respectivement l'union, l'intersection et le complémentaire d'une liste dans une autre.

```
union([],[],[]).
union(L,[],M) :- retireDoublon(L,M).
union(L1,[X|L2],L12) :- membre(X,L1), union(L1,L2,L12), !.
union(L1,[X|L2],[X|L12]) :- union(L1,L2,L12).
```

ne fonctionne que si les deux premiers arguments ont une valeur. On aura par exemple :

```
| ?- union([1,2],[56,98,2,71,1,3,4],L).

L = [56,98,71,3,4,1,2]

(10 ms) yes
| ?- union([1,2],[2,3,4],L).

L = [3,4,1,2]

yes
```

Pour l'intersection, elle peut s'écrire :
```

```
inter([],[],[]).
inter(_,[],[]).
inter(L1,[X|L2],[X|L12]) :- membre(X,L1), inter(L1,L2,L12), !.
inter(L1,[_|L2],L12) :- inter(L1,L2,L12).
```

On aura par exemple :

```
| ?- inter([1,3,2],[4,5,2,98,7],L).

L = [2]

yes
| ?- inter([1,3,2],[4,5,2,98,3,7],L).

L = [2,3]

yes
```

Pour le complémentaire d'un ensemble dans un autre, on peut écrire :

```
comp([],_,[]).
comp([X|L1],L2,L12) :- membre(X,L2), comp(L1,L2,L12), !.
comp([X|L1],L2,[X|L12]) :- comp(L1,L2,L12).
```

On aura alors par exemple :

```
| ?- comp([1,2,3],[3,4,1],L).

L = [2]

yes
| ?- comp([1,2,3],[3,4,11],L).

L = [1,2]

yes
```

3.8 Égalité de deux ensembles

Remarque : soit l'interrogation :

```
| ?- union([1,2],[2,3,4],[3,4,1,2]).

yes
| ?- union([1,2],[2,3,4],[3,4,2,1]).

no
```

La première réponse est normale, pas la deuxième puisque nous avons affaire à des ensembles dans lesquels l'ordre des éléments n'importe pas.

Il faut donc définir un prédicat qui teste si deux ensembles sont égaux. Cela s'écrit très facilement si on remarque que le complémentaire d'un ensemble dans un ensemble qui contient les mêmes éléments que lui est vide, d'où :

```
ensemblesEgaux([],[]).
ensemblesEgaux(L1,L2) :- comp(L1,L2,[]).
```

4 Toujours plus loin

On indique en vrac des techniques et quelques prédicats prédéfinis dans `gprolog` qui sont indispensables pour écrire des programmes complets. Puis, on reprend des exemples que l'on peut maintenant traiter.

4.1 Prédicats sur la nature d'un objet

- `var(X)` est vrai si `X` possède une valeur (est instanciée) ;
- `nonvar(X)` est vrai si `X` n'est pas instanciée ;
- `atom(X)` est vrai si `X` est un symbole (un « atome ») ;
- `integer(X)` est vrai si `X` est un entier (ou une variable instanciée avec une valeur entière) ;
- `float(X)` est vrai si `X` est un flottant (ou une variable instanciée avec une valeur flottante) ;
- `number(X)` est vrai si `integer(X)` ou `float(X)` est vrai ;
- `atomic(X)` est vrai si `X` est un symbole ou un nombre, c'est-à-dire, si `X` n'est pas une liste ;
- `list(X)` est vrai si `X` est une liste.

4.2 Expressions arithmétiques

Le prédicat `N is expr` rend vrai si la valeur de la variable `N` est égale à l'expression arithmétique `expr` si `N` est instanciée, affecte la valeur de `expr` à `N` si celle-ci n'est pas instanciée, et rend alors vrai.

L'expression `expr` s'exprime avec des variables, des constantes et les opérations et fonctions habituelles. Si elles contient des variables, celles-ci doivent impérativement être instanciées.

On peut tester la valeur de deux expressions arithmétiques à l'aide des opérateurs suivants :

- `E1 =:= E2` est vrai si la valeur de `E1` est égale à la valeur de `E2` ;
- `E1 =\= E2` est vrai si la valeur de `E1` est différente à la valeur de `E2` ;
- `E1 < E2` est vrai si la valeur de `E1` est inférieure strictement à la valeur de `E2` ;
- `E1 =< E2` est vrai si la valeur de `E1` est inférieure ou égale à la valeur de `E2` ;
- `E1 > E2` est vrai si la valeur de `E1` est supérieure strictement à la valeur de `E2` ;
- `E1 >= E2` est vrai si la valeur de `E1` est supérieure ou égale à la valeur de `E2`.

Plus généralement, on peut tester la valeur de deux termes (des variables instanciées ou des symboles) à l'aide des opérateurs suivants :

- `T1 == T2` est vrai si `T1` est égal à `T2` ;
- `T1 \== T2` est vrai si `T1` est différent de `T2` ;
- `T1 @< T2` est vrai si `T1` est inférieur strictement à `T2` : si ce sont des symboles, l'ordre alphabétique est utilisé ; si ce sont des variables instanciées, c'est l'ordre d'instanciation qui est utilisé : la plus anciennement instanciée est inférieure à l'autre ;
- `T1 @=< T2` est vrai si `T1` est inférieure ou égale à `T2` ;
- `T1 @> T2` est vrai si `T1` est supérieure strictement à `T2` ;
- `T1 @>= T2` est vrai si `T1` est supérieure ou égale à `T2`.

4.3 Membre récursif

Écrire un prédicat `membre2` qui est vrai si une valeur appartient à une liste ou à une de ses sous-listes ceci récursivement. Ainsi, avec le prédicat `membre` écrit plus haut, on a :

```
membre(3,[1,[2,3],4]).
```

qui donne `no`.

Pour écrire ce prédicat, on a besoin d'introduire quelques prédicats prédéfinis sur les objets :

```
membre2(X,[X|_]).
membre2(X,[Y|_]) :- list(Y), membre2(X,Y).
membre2(X,[_|L]) :- membre2(X,L).
```

4.4 Renverse récursif

De la même façon, on peut redéfinir le prédicat `renverse` pour qu'il agisse de manière récursive également :

```
metAuBout(L,[],L).
metAuBout([],L,[L]).
metAuBout([X|Y],L,[X|L12]) :-
    metAuBout(Y,L,L12).

renverse2([],[]).
renverse2([X|L],M) :- atomic(X), renverse2(L,N), concat(N,[X],M).
renverse2([X|L],M) :- list(X), renverse2(L,N), renverse2(X,Y), metAuBout(N,Y,M).
```

On a alors par exemple :

```
renverse2([[1,2,3],4,5,[6,[7,8]]],L).

L = [[[8,7],6],5,4,[3,2,1]] ?
```

On remarque par contre que si la variable est en première position, le prédicat ne fonctionne pas. Aussi, on peut ajouter le prédicat suivant :

```
renverse3(L1,L2) :- var(L1), renverse2(L2,L1), !.
renverse3(L1,L2) :- var(L2), renverse2(L1,L2), !.
```

qui donne bien le résultat attendu.

4.5 Mise à plat d'une liste

Écrire un prédicat qui est vrai si le deuxième argument est identique au premier si ce n'est que les éléments des sous-listes du premier ont été mis simplement dans la liste résultante. Par exemple :

```
miseAPlat([1,[2,[3]],[4,5,[6,7]]],L).

L = [1,2,3,4,5,6,7] ?
```

et, bien sûr,

```
miseAPlat([1,2,3,4,5,6,7],L).

L = [1,2,3,4,5,6,7] ?
```

```
miseAPlat([],[]).
miseAPlat([X|L],[X|M]) :- atomic(X), miseAPlat(L,M).
miseAPlat([X|L],M) :-
  list(X),
  miseAPlat(X,X1),
  miseAPlat(L,N),
  concat(X1,N,M), !.
```

4.6 Les tableaux

Il n'y a pas explicitement de tableaux en `gprolog`. Par contre, on peut facilement faire comme si grâce aux listes. Une liste contient les éléments du tableau. Il faut alors disposer de prédicats qui accèdent à ses éléments comme s'il s'agissait d'un tableau, en fournissant l'indice ou les indices de l'élément à accéder. Pour lire la valeur d'un élément de tableau, on utilise alors le prédicat **`nieme`** défini plus haut. Pour écrire la valeur d'un élément du tableau, on écrit un nouveau prédicat qui le fait, toujours en utilisant une liste pour contenir le tableau.

```
affecteNieme([],_,_,[]).
affecteNieme(T,0,_,T).
affecteNieme([_|T],1,V,[V|T2]) :- affecteNieme(T,0,_,T2), !.
affecteNieme([X|T],I,V,[X|T2]) :-
  I1 is I-1,
  affecteNieme(T,I1,V,T2).
```

4.7 Affichage d'un message à l'écran

Le prédicat `write` affiche un message à l'écran. Il s'utilise très simplement en lui passant en argument la valeur à afficher. Ce peut être un symbole, un nombre, une chaîne de caractères ou une variable. Dans ce dernier cas, la variable doit impérativement être instanciée. Par exemple, dans :

```
| ?- membre(X,[1,2,3]), write('La valeur est '), wwrite(X).
La valeur est 1

X = 1 ?
```

```
La valeur est 2

X = 2 ?
La valeur est 3

X = 3 ?
```

on voit qu'avant l'interrogation pour chacune des réponses, les `write` ont affiché un message contenant la valeur de `X`.

5 Encore quelques exemples

5.1 Tri par insertion

Écrire un prédicat qui effectue un tri par insertion d'une liste.

```
tri([],[]).
tri(L,[Y|M]) :- minimum(L,Y), retire1(Y,L,N), tri(N,M), !.
```

que l'on utilise comme suit :

```
| ?- tri([5,8,1,432,3,5,8,1,18,23,7],X).

X = [1,1,3,5,5,7,8,8,18,23,432]
```

5.2 Tri rapide

Écrire un prédicat qui effectue un tri d'une liste par l'algorithme du tri rapide.

```
split([],_,[],[]).
split([X|L],P,[X|I],S) :- X < P, !, split(L,P,I,S).
split([X|L],P,I,[X|S]) :- X >= P, split(L,P,I,S).

concat3([],[],[],[]).
concat3([],L1,L2,M) :- concat(L1,L2,M).
concat3([X|L1],L2,L3,[X|M]) :- concat3(L1,L2,L3,M).

qsort([],[]).
qsort([X|L],M) :-
```

```
    split(L,X,LI,LS),
    qsort(LI,LIT),
    qsort(LS,LST),
    concat3(LIT,[X],LST,M), !.
```

que l'on utilise comme suit :

```
| ?- qsort([5,8,1,432,3,5,8,1,18,23,7],X).

X = [1,1,3,5,5,7,8,8,18,23,432]
```

5.3 Le crible d'Ératosthène

```
listeVide(0,[]).
listeVide(N,[1|L]) :-
  N1 is N-1,
  listeVide(N1,L), !.

barreMultiples(N,MD,_,L,L) :- MD > N, !.
barreMultiples(N,MD,M,L,L2) :-
  affecteNieme(L,MD,0,L3),
  M1 is M+MD,
  barreMultiples(N,M1,M,L3,L2).

cc(N,M,C,C) :-
  N < M*M, !.
cc(N,M,L,C) :-
  MD is 2*M,
  barreMultiples(N,MD,M,L,L2),
  M1 is M+1,
  cc(N,M1,L2,C).

calculeCrible(N,C) :-
  listeVide(N,L),
  cc(N, 2, L, C), !.
```

L'appel du prédicat `calculeCrible(30,L).` renverra dans `L` le crible pour les entiers compris entre 1 et 30.

On peut ensuite écrire un prédicat qui affiche tous les nombres premiers jusqu'à une valeur donnée :

```
premier(N) :-
  calculeCrible(N,C),
  nieme(C,N,1).

afficheSiPremier(N) :-
  premier(N),
  write(N),
  write(' '), !.
afficheSiPremier(_).

tousLesPremiers(N,I) :-
  I =< N,
  afficheSiPremier(I),
  I1 is I+1,
  tousLesPremiers(N,I1).

afficheLesPremiers(N) :-
  tousLesPremiers(N,1).
```

5.4 Calcul formel

On peut définir un ensemble de prédicats permettant de calculer la dérivée d'une fonction. Une fonction est représentée sous forme préfixée par une liste. Ainsi, la fonction $f(x) = 2e^{3x} - x^3 + 3$ se représente alors par la liste prolog : `[+,[-,[*,2,[e,[*,3,x]]],[^,x,3]],3]`. On souhaite définir le prédicat `derive(l,v,d)` qui fournit dans `d` la dérivée de la fonction représentée par la liste `l` par rapport à la variable `v`. On rappelle les règles de dérivation suivante :

- la dérivée d'une fonction vide est nulle ;
- la dérivée d'une constante est nulle ;
- la dérivée d'une somme est la somme des dérivées de ses termes ;
- la dérivée d'une soustraction est la soustractions des dérivées de ses termes ;
- la dérivée du produit d'une constante par une fonction est le produit de cette constante par la dérivée de cette fonction ;
- la dérivée d'un produit de fonctions uv est $uv' + u'v$;
- la dérivée du quotient de deux fonctions $\frac{u}{v}$ est $\frac{uv'-u'v}{uv}$;

– la dérivée de l'exponentionnelle d'une fonction est le produit de la dérivée de cette fonction par l'exponentionnelle de la fonction ;
– la dérivée de x^y, si y est un nombre, est yx^{y-1}.

On exprime alors très facilement ces règles sous la forme :

```
derivee(F,_,0) :- F == [], !.
derivee(X,X,1) :- !.
derivee(Y,X,0) :- atomic(Y), X \== Y, !.
derivee([+,L,M],X,[+,LD,MD]) :-
    derivee(L,X,LD),
    derivee(M,X,MD), !.
derivee([-,L,M],X,[-,LD,MD]) :-
    derivee(L,X,LD),
    derivee(M,X,MD), !.
derivee([*,U,V],X,[+,[*,U,DV],[*,V,DU]]) :-
    derivee(U,X,DU),
    derivee(V,X,DV), !.
derivee([/,U,V],X,[/,[-,[*,U,DV],[*,V,DU]],[*,U,V]]) :-
    derivee(U,X,DU),
    derivee(V,X,DV), !.
derivee([^,X,Y],X,[*,Y,[^,X,Z]]) :-
    number(Y), Z is Y-1, !.
derivee([e,F],X,[*,FD,[e,F]]) :-
    derivee(F,X,FD), !.
```

que l'on peut alors utiliser pour calculer la dérivée de la fonction indiquée plus haut : `derivee([+,[-,[*,2,[e,[*,3,x]]],[^,x,3]],3],x,D).` provoque la réponse :

```
D = [+,[-,[+,[*,2,[*,[/,[-,[*,3,1],[*,x,0]],[*,3,x]],[e,[/,3,x]]]],
  [*,[e,[/,3,x]],0]],[*,3,[^,x,2]]],0]
```

qui représente bien la dérivée de la fonction $f(x)$ par rapport à x.

Bien entendu, cette dérivée n'est pas sous forme simplifiée : pourquoi le serait-elle ? et comment pourrait-elle l'être puisque l'on n'a pas demandé à ce qu'elle le soit et l'on n'a pas expliquée comment on simplifie une expression algébrique. C'est là un excellent sujet de réflexion qui, arrivé là où nous en sommes, ne devrait plus poser de problème.