Prolog 2019-2020

Les prédicats procéduraux comme **is** et = **:=** sont interdits sauf mention explicite du contraire.

Semaine 1

Premier exemple

```
% fait : socrate est un homme

homme(socrate).


% règle : si une personne est un homme,
%           alors elle est mortelle

mortel(X) :-
    homme(X).
```

Généalogie

```
homme(albert).
homme(jean).
homme(paul).
homme(bertrand).
homme(louis).
homme(benoit).


femme(germaine).
femme(christiane).
femme(simone).
femme(marie).
femme(sophie).



% nommer les prédicats arg1_arg2_..._argn donc
% NE PAS avoir un prédicat pere(arg1,arg2)

pere_enfant(albert,jean).
pere_enfant(jean,paul).
pere_enfant(paul,bertrand).
pere_enfant(paul,sophie).
```

```
pere_enfant(jean,simone).
pere_enfant(louis,benoit).


mere_enfant(germaine,jean).
mere_enfant(christiane,simone).
mere_enfant(christiane,paul).
mere_enfant(simone,benoit).
mere_enfant(marie,bertrand).
mere_enfant(marie,sophie).


humain(X) :-
    homme(X).

humain(X) :-
    femme(X).


parent_enfant(Parent,Enfant) :-
    pere_enfant(Parent,Enfant).

parent_enfant(Parent,Enfant) :-
    mere_enfant(Parent,Enfant).


fils_parent(F,P) :-
    parent_enfant(P,F),
    homme(F).

fille_parent(F,P) :-
    parent_enfant(P,F),
    femme(F).


grandPere_petitEnfant(GP,PE) :-
    pere_enfant(GP,X),
    parent_enfant(X,PE).


grandMere_petitEnfant(GM,PE) :-
    mere_enfant(GM,X),
    parent_enfant(X,PE).


frere_frereOuSoeur(X,Y) :-
    homme(X),
    parent_enfant(Parent,X),
    parent_enfant(Parent,Y),
    dif(X,Y).
```

Exercice

```
personne_aime(marie,vin).

voleur(pierre).

personne_aime(pierre,Y) :-
    personne_aime(Y,vin).

personne_vole(X,Y) :-
    voleur(X),
    personne_aime(X,Y).
```

Graphes

```
:- use_module(library(clpfd)).


origine_destination_cout(a,b,2).
origine_destination_cout(a,g,6).
origine_destination_cout(b,e,2).
origine_destination_cout(b,c,7).
origine_destination_cout(g,e,1).
origine_destination_cout(g,h,4).
origine_destination_cout(e,f,2).
origine_destination_cout(f,c,3).
origine_destination_cout(f,h,2).
origine_destination_cout(c,d,3).
origine_destination_cout(h,d,2).


/*
chemin(X,Y) :-
    origine_destination_cout(X,Y,_).

chemin(X,Y) :-
    origine_destination_cout(X,Z,_),
    origine_destination_cout(Z,Y,_).

chemin(X,Y) :-
    origine_destination_cout(X,Z,_),
    origine_destination_cout(Z,W,_),
    origine_destination_cout(W,Y,_).
*/


% Définition récursive d'un chemin

% Cas de base : exactement un arc

origine_destination__chemin(S1,S2) :-
    origine_destination_cout(S1,S2,_).
```

```
% Cas général : un chemin de longueur au moins 2 est constitué
% d'un arc suivi d'un chemin

origine_destination__chemin(S1,S2) :-
    origine_destination_cout(S1,S,_),
    origine_destination__chemin(S,S2).



% Définition récursive d'un chemin

% Cas de base : exactement un arc

origine_destination_cout__chemin(S1,S2,C) :-
    origine_destination_cout(S1,S2,C).

% Cas général : un chemin de longueur au moins 2 est constitué
% d'un arc suivi d'un chemin

origine_destination_cout__chemin(S1,S2,C) :-
    origine_destination_cout(S1,S,CoutArc),
    origine_destination_cout__chemin(S,S2,CoutChemin),
    C #= CoutArc+CoutChemin.
```

Semaine 2

```
:- use_module(library(clpfd)).

% https://www.swi-prolog.org/man/clpfd.html

% longueur d'une liste

% cas de base

liste_longueur([],0).


% cas général

liste_longueur([_X|L],N) :-   % <-
    liste_longueur(L,N1),
    N #= N1+1.

/*
 lecture déclarative :
 SI L a pour longueur N1
   ET que N = N1+1

 ALORS
   la liste L précédée de l'élément _X a pour longueur N
*/

% power of prolog

% une liste est soit :
%   - une liste vide notée [] ;
%   - contituée d'un élément X et d'un reste (ou queue) R.
%     Cette liste est notée [X|R].


% longueur d'une liste

% cas de base

liste_longueur([],0).

/*
 lecture déclarative :

 La liste vide a pour longueur 0.
*/

% cas général

liste_longueur([_X|L],N) :-   % <-
    liste_longueur(L,N1),
```

```
    N #= N1+1.


/*
 lecture déclarative :
 SI L a pour longueur N1
   ET que N = N1+1

 ALORS
   la liste L précédée de l'élément _X a pour longueur N
*/

% power of prolog


% pas de gestion explicite de la mémoire/pas d'espace à réserver
% ?- L=[a,b,c].

% pas de typage/tous les éléments peuvent être de type différent
% atome ou variable
% ?- L=[a,1,3.14,X].


% suite de Fibonacci

% f(0)=0,f(1)=1,f(n)=f(n-1)+f(n-2)

entier_fibonacci(0,0).

entier_fibonacci(1,1).

entier_fibonacci(N,Fib) :-
    N #>= 2,
    N1 #= N-1,
    entier_fibonacci(N1,Fib1),
    N2 #= N-2,
    entier_fibonacci(N2,Fib2),
    Fib #= Fib1+Fib2.


% plus grand diviseur commun

% version avec soustraction

entier_entier_pgcd(N,N,N).

entier_entier_pgcd(N1,N2,N) :-
    N1 #< N2,
    NN2 #= N2-N1,
    entier_entier_pgcd(N1,NN2,N).
```

```
entier_entier_pgcd(N1,N2,N) :-
    N1 #> N2,
    NN1 #= N1-N2,
    entier_entier_pgcd(NN1,N2,N).



% TD sur les listes

% appartenance   member

membre_liste(X,[X|_L]).

membre_liste(X,[_Y|L]) :-
    membre_liste(X,L).


% nieme   nth0 nth1 https://www.swi-prolog.org/pldoc/man?predicate=nth0/3

indice_liste_element__1(1,[X|_L],X).

indice_liste_element__1(N,[_X|L],X) :-
    N #> 1,
    N1 #= N-1,
    indice_liste_element__1(N1,L,X).



dernier_liste(X,[X]).

dernier_liste(X,[_Y|L]) :-
    dernier_liste(X,L).

/*
  SI X est le dernier élément de L

  ALORS X est le dernier élément de toute liste
        composée d'un élément suivi de L
 */

avantDernier_liste(X,[X,_Y]).          % liste à exactement 2 éléments

avantDernier_liste(X,[_X1,X2,X3|L]) :-
    avantDernier_liste(X,[X2,X3|L]).


liste_inverse([],[]).

liste_inverse([X|L1],L2) :-
    liste_inverse_acc([X|L1],L2,[]).
```

```
liste_inverse_acc([],Acc,Acc).

liste_inverse_acc([X|L1],L2,Acc) :-
    liste_inverse_acc(L1,L2,[X|Acc]).


palindrome(L) :-
    liste_inverse(L,L).

% palindrome([k,a,y,a,k]).


liste_compressee([],[]).

liste_compressee([X],[X]).

liste_compressee([X,X|L],NL) :-
    liste_compressee([X|L],NL).

liste_compressee([X,Y|L],[X|NL]) :-
    dif(X,Y),
    liste_compressee([Y|L],NL).

/*
  lecture déclarative :

*
  SI  est différent de Y
    ET que Y suivi de L a pour iste compressee NL

  ALORS X suivi de (Y suivi de L) a pour liste compressée
        X suivi de NL
 */




entier_listeDiviseurs(N,L) :-
    findall(
            Div,
            (between(1,N,Div),N #= Div*_),
            L
           ).

premier(N) :-
    entier_listeDiviseurs(N,[1,N]).



entier_racine(N,R) :-
    R in 0..N,
```

```
R^2 #=< N,
N #< (R+1)^2.
```

Semaine 3

```
:- use_module(library(clpfd)).

% Constraint Logic Programming Finite Domain
% gère des ENTIERS
%
% https://swish.swi-prolog.org/p/FzJzpXIF.pl#tabbed-tab-0RS


% cas de base (fait) : 0!=1

entier_factorielle(0,1).

% cas général (règle) : N!=N*(N-1)!

entier_factorielle(N,Fact) :- % <-
    N #> 0,
    N1 #= N-1,
    entier_factorielle(N1,Fact1),
    Fact #= N*Fact1.

% https://www.metalevel.at/prolog/reading

/*
 lecture déclarative :
 SI N>0
   ET N1 = N-1
   ET factorielle(N1)=Fact1
   ET Fact=N*Fact1

 ALORS
   factorielle(N)=Fact
 */

/*
  lecture procédurale :
  Pour que factorielle(N)=Fact,
  il suffit que N>0
   ET N1 = N-1
   ET factorielle(N1)=Fact1
   ET Fact=N*Fact1
 */


:- use_module(library(clpfd)).


% Notation [X|L]
%
% !!! ATTENTION !!!
%
```

```
% | n'est pas symétrique
%
% X est FORCEMENT exactement UN élément
% L est une liste quelconque (y compris vide)


% longueur d'une liste

% PREDEFINI length

% cas de base

liste_longueur([],0).

/*
 lecture déclarative :

 La liste vide a pour longueur 0.
*/

% cas général

liste_longueur([_X|L],N) :-   % <-
    N #> 0,
    N #= N1+1,
    liste_longueur(L,N1).


% ATTENTION, pas de fait/règle
% pour un énoncé faux

% membre_liste(X,[]) :- false.

membre_liste(X,[X|_L]).

membre_liste(X,[_Y|L]) :-
    membre_liste(X,L).



% nieme   nth0 nth1 https://www.swi-prolog.org/pldoc/man?predicate=nth0/3

indice_liste_element__1(1,[X|_L],X).

indice_liste_element__1(N,[_X|L],X) :-
    N #> 1,
    N1 #= N-1,
    indice_liste_element__1(N1,L,X).
```

```
% 99 problems n 1

dernier_liste(X,[X]).

dernier_liste(X,[_Y|L]) :-
    dernier_liste(X,L).


% 99 problems n 2

avantDernier_liste(X1,[X1,_X2]).

avantDernier_liste(X,[_Y|L]) :-
    avantDernier_liste(X,L).


% 99 problems n 3

% voir plus haut
% indice_liste_element__1


% 99 problems n 4

% voir plus haut
% liste_longueur


% 99 problems n 5

% Ecrire un prédicat liste_inverse(L,LInverse)

% Commencer par écrire un prédicat
% liste_liste_concatenation(L1,L2,Concatenation)
%
% ?- liste_liste_concatenation([a,b],[c,d,e],L).
% L = [a,b,c,d,e]

% potentiellement 4 cas
% L1 vide      L2 vide
% L1 non vide  L2 vide
% L1 vide      L2 non vide
% L1 non vide  L2 non vide

/*
liste_liste_concatenation([],[],[]).

liste_liste_concatenation([X|L1],[],[X|L1]).

liste_liste_concatenation([],[X|L2],[X|L2]).
```

```
liste_liste_concatenation([X|L1],[Y|L2],[X|L]) :- % <-
    liste_liste_concatenation(L1,[Y|L2],L).
*/

% lecture déclarative
%
% SI
%   L1 concaténée avec [Y|L2] donne L
% ALORS
%   [X|L1] concaténée avec [Y|L2] donne [X|L]

liste_liste_concatenation([],L2,L2).

liste_liste_concatenation([X|L1],L2,[X|L3]) :-
    liste_liste_concatenation(L1,L2,L3).


% ?- liste_inverse([a,b,c],LInverse).
% LInverse = [c,b,a].

% cas de base (fait) : liste vide

liste_inverse__naif([],[]).

% lecture déclarative
% l'inverse de la liste vide est la liste vide

liste_inverse__naif([X|L1],L2) :- % <-
    liste_liste_concatenation(NL1,[X],L2),   % X est le dernier élément de L2
    liste_inverse(L1,NL1).                   % NL1 est l'inverse de L1

% SI
%      NL1 concaténée avec [X] donne L2
%   ET NL1 est l'inverse de L1
%
% ALORS
%   L2 est l'inverse de [X|L1]


liste_inverse([],[]).

% acc = accumulateur

liste_inverse([X|L1],L2) :-
    liste_inverse_acc([X|L1],L2,[]).

liste_inverse_acc([],Acc,Acc).

liste_inverse_acc([X|L1],L2,Acc) :-
    liste_inverse_acc(L1,L2,[X|Acc]).
```

```
entier_factorielle__rec(0,1).

entier_factorielle__rec(N,Fact) :-
    N #>0,
    N1 #= N-1,
    entier_factorielle__rec(N1,Fact1),
    Fact #= N*Fact1.


entier_factorielle(0,1).

% accumulateur

entier_factorielle(N,Fact) :-
    N #> 0,
    entier_factorielle_acc(N,Fact,1).


entier_factorielle_acc(0,Acc,Acc).

entier_factorielle_acc(N,Fact,Acc) :-
    N #> 0,
    N1 #= N-1,
    NAcc #= N*Acc,
    entier_factorielle_acc(N1,Fact,NAcc).
```

Semaine 4

```
:- use_module(library(clpfd)).

% Constraint Logic Programming Finite Domain

liste_inverse([],[]).

% acc = accumulateur

liste_inverse([X|L1],L2) :-
    liste_inverse_acc([X|L1],L2,[]).


liste_inverse_acc([],Acc,Acc).

liste_inverse_acc([X|L1],L2,Acc) :-
    liste_inverse_acc(L1,L2,[X|Acc]).


% palindrome(L) est vrai si L représente un palindrome

% ?- palindrome([n,o,n]).
% true

% ?- palindrome([o,u,i]).
% false

palindrome(L) :-
    liste_inverse(L,L).



borneInf_borneSup_liste(Sup,Sup,[Sup]).

borneInf_borneSup_liste(Inf,Sup,[Inf|L]) :-
    Inf #< Sup,
    Inf1 #= Inf+1,
    borneInf_borneSup_liste(Inf1,Sup,L).


% findall(Element,(Buts,...),L)

% ATTENTION, findall à 3 arguments


borneInf_borneSup_liste__findall(Inf,Sup,L) :-
    findall(
            N,
            between(Inf,Sup,N),
            L
           ).
```

```
% PREDICAT PREDEFINI numlist


% entier_listeDiviseurs

% ?- entier_listeDiviseurs(7,L).
% L=[1,7]

% ?- entier_listeDiviseurs(10,L).
% L=[1,2,5,10]

entier_listeDiviseurs(N,L) :-
    findall(
            Div,
            (between(1,N,Div),N mod Div #= 0), % N #= Div*_
            L
           ).


% premier(N) est vrai si N est premier

% ATTENTION, 1 n'est ni premier, ni composé

premier(N) :-
    entier_listeDiviseurs(N,[1,N]). % entier_listeDiviseurs(N,[_,_]).


% ?- between(1,inf,N),premier(N).


borneSup_listePremiers__findall(BorneSup,L) :-
    findall(
            Premier,
            (between(1,BorneSup,Premier),premier(Premier)),
            L
           ).


borneSup_listePremiers__include(BorneSup,L) :-
    borneInf_borneSup_liste(1,BorneSup,ListeEntiers),
    include(premier,ListeEntiers,L).


pair(N) :-
    N #= 2*_.

% include
% exclude
% partition
```

Semaine 5

```
:- use_module(library(clpfd)).


entier_carre(N,Carre) :-
    Carre #= N*N.

% cas de base : 0 élément

liste_somme([],0).

% ces général : au moins un élément

liste_somme([N|L],Somme) :-
    Somme #= N+Somme1,
    liste_somme(L,Somme1).


% cas de base : 0 élément

liste_produit([],1).

% ces général : au moins un élément

liste_produit([N|L],Produit) :-
    Produit #= N*Produit1,
    liste_produit(L,Produit1).



% cas de base : 0 élément

liste_listeTriee__fusion([],[]).

% cas de base : 1 élément

liste_listeTriee__fusion([N],[N]).

% cas général : au moins 2 éléments

liste_listeTriee__fusion([N1,N2|L],LTriee) :-
    liste_listeIndicesImpairs_listeIndicePairs([N1,N2|L],LImpairs,LPairs),
    liste_listeTriee__fusion(LImpairs,LImpairsTries),
    liste_listeTriee__fusion(LPairs,LPairsTries),
    liste_liste_listeFusion(LImpairsTries,LPairsTries,LTriee).


liste_listeIndicesImpairs_listeIndicePairs([],[],[]).

liste_listeIndicesImpairs_listeIndicePairs([N],[N],[]).
```

```
liste_listeIndicesImpairs_listeIndicePairs([N1,N2|L],[N1|L1],[N2|L2]) :-
    liste_listeIndicesImpairs_listeIndicePairs(L,L1,L2).


% liste_liste_listeFusion(LPairsTries,LImpairsTries,LTriee)

% Les deux premiers arguments doivent être des listes triées

liste_liste_listeFusion([],LPairsTries,LPairsTries).

liste_liste_listeFusion(LImpairsTries,[],LImpairsTries).

liste_liste_listeFusion([N1|L1],[N2|L2],[N1|LTriee]) :-
    N1 #=< N2,
    liste_liste_listeFusion(L1,[N2|L2],LTriee).

liste_liste_listeFusion([N1|L1],[N2|L2],[N2|LTriee]) :-
    N1 #> N2,
    liste_liste_listeFusion([N1|L1],L2,LTriee).


liste_liste__quicksort([],[]).

liste_liste__quicksort([Pivot|L],NL) :-
    partition(#>(Pivot),L,Petits,Grands),
    liste_liste__quicksort(Petits,PetitsTries),
    liste_liste__quicksort(Grands,GrandsTries),
    append(PetitsTries,[Pivot|GrandsTries],NL).
```

Semaine 6

```
:- use_module(library(clpfd)).

%
% distinguer autant que possible les contraintes
% de la recherche d'une (ou plusieurs) solution(s)

solution(L) :-
    L=[S1,S2,S3,S4,S5,S6,S7,S8],

    % domaine ?
    L ins 1..3,

    % S1 #\= S2, S1 #\= S3, S1 #\= S4

    maplist(#\=(S1),[S2,S3,S4,S5]),
    maplist(#\=(S2),[S3,S4,S6]),
    maplist(#\=(S5),[S7,S8]),
    maplist(#\=(S6),[S7,S8]),
    maplist(#\=(S7),[S8]).

% label(L) L DOIT être une liste = labeling([],L)

% attention, le choix par défaut (donner les valeurs aux variables) dans l'ordre de L
% est souvent TRES mauvais

% deux options importantes pour labeling
% ff : first fail (chercher une valeur pour la variable dont le domaine résiduel est
% le plus petit)
% ffc : first fail involved in most constraints (variable de plus haut degré
% dans le graphe des contraintes)

?- solution(L),label(L).
?- solution(L),labeling([ffc],L).
```

Mémoïsation explicite avec asserta/assertz

Attention ! Ceci ne marche pas sur SWISH.

```
:- use_module(library(clpfd)).

:- dynamic entier_fib__assert/2.


entier_fib__assert(0,0).

entier_fib__assert(1,1).

entier_fib__assert(N,R) :-
    N #>= 2,
    N1 #= N-1,
```

```
    entier_fib__assert(N1,R1),
    N2 #= N-2,
    entier_fib__assert(N2,R2),
    R #= R1+R2,
    asserta(entier_fib__assert(N,R) :- !).

/* utiliser asserta pour un ajout au début et assertz pour un ajout à la fin */
```

Mémoïsation implicite avec tabling

```
:- table entier_fib__table/2.


entier_fib__table(0,0).

entier_fib__table(1,1).

entier_fib__table(N,R) :-
    N #>= 2,
    N1 #= N-1,
    entier_fib__table(N1,R1),
    N2 #= N-2,
    entier_fib__table(N2,R2),
    R #= R1+R2.
```

Semaine 7

Cryptarithme

```
:- use_module(library(clpfd)).

% https://www.crazy-stuff.net/fr/divertissements/enigmes/cinq-maisons-einstein

solve(L,Poisson) :-

     L = [Nationalite,Couleur,Animal,Boisson,Sport],

    Nationalite = [Anglais,Suedois,Danois,Allemand,Norvegien],
    Couleur     = [Rouge,Blanc,Vert,Jaune,Bleu],
    Animal      = [Chien,Oiseau,Chat,Cheval,Poisson],
    Boisson     = [The,Cafe,Lait,Biere,Eau],
    Sport       = [Football,Volley,Baseball,Tennis,Hockey],

    Nationalite ins 1..5,
    Couleur     ins 1..5,
    Animal      ins 1..5,
    Boisson     ins 1..5,
    Sport       ins 1..5,

    maplist(all_distinct,[Nationalite,Couleur,Animal,Boisson,Sport]),

    Anglais #= Rouge,
    Suedois #= Chien,
    Danois #= The,
    Vert #= Blanc-1,
    Vert #= Cafe,
    Football #= Oiseau,
    Jaune #= Baseball,
    Lait #= 3,
    Norvegien #= 1,
    abs(Volley-Chat) #= 1,
    abs(Cheval-Baseball) #= 1,
    Tennis #= Biere,
    Allemand #= Hockey,
    abs(Norvegien-Bleu) #= 1,
    abs(Volley-Eau) #= 1.
```

Cryptarithme

```
:- use_module(library(clpfd)).

% cryptarithme

% ?- probleme([S,E,N,D]+[M,O,R,E]=[M,O,N,E,Y],Variables).

probleme(Somme=Total,Variables) :-
    somme_liste(Somme,Liste), % variables de la somme
    append(Liste,Total,Vs),   % toutes les variables avec multiplicité
    sort(Vs,Variables),       % supprimer les doublons

    Variables ins 0..9,
    all_distinct(Variables),

    contrainte(Somme=Total),

    premierNonNul(Somme),
    premierNonNul(Total).

% ?- somme_liste([S,E,N,D]+[M,O,R,E],[S,E,N,D,M,O,R,E]).

somme_liste(L,L) :- is_list(L).

somme_liste(AutresTermes+L,NL) :-
    somme_liste(AutresTermes,NNL),
    append(NNL,L,NL).


% ?- contrainte([S,E,N,D]+[M,O,R,E]=[M,O,N,E,Y]).

contrainte(Somme=Total) :-
    somme_nombre(Somme,NombreSomme),
    somme_nombre(Total,NombreTotal),
    NombreSomme  #= NombreTotal.

% ?- liste_nombre([S,E,N,D],S*10^3+E*10^2+N*10^1+D*10^0)

liste_nombre([N],N).

liste_nombre([N1,N2|L],N) :-
    append(LL,[Dernier],[N1,N2|L]),
    liste_nombre(LL,NLL),
    N #= 10*NLL+Dernier.


somme_nombre(L,N) :-
    is_list(L),
    liste_nombre(L,N).

somme_nombre(AutresTermes+L,NL) :-
```

```
    somme_nombre(L,N),
    somme_nombre(AutresTermes,NN),
    NL #= N+NN.


premierNonNul([N|_]) :- N #\= 0.

premierNonNul(AutresTermes+L) :-
    premierNonNul(L),
    premierNonNul(AutresTermes).

% ?- probleme([M,A,T,H]+[M,A,T,H]=[H,A,B,I,T],Variables).
```