Prolog :

1) ouverture de logiciel

2 parties :

-programme

-question

Programme contient des termes composés :

- aime(romeo,juliette)

Le foncteur est *aime* et l'arité 2, le terme composé s'écrit aime/2.

- initialisation

Le foncteur est *initialisation* et l'arité 0, le terme composé s'écrit initialisation/0.

Un atome est donc un terme composé d'arité 0.

Predicat :
Fait : homme(socrate).
Règle : mortel(socrate) :- homme(socrate).

Question : homme(Rachid) ?

Réponse sous forme de booléen (avec parfois les différentes solutions qui permettent d'avoir vrai comme booléen).

```
capable de distinguer
```
salut(A,B).
salut(A,B,C).

2) Variables

Les variables commencent pas des majuscules.

3) Utilisation classique

homme(andre).
homme(bernard).
homme(babar).
femme(augustine).
femme(becassine).
femme(brigitte).
enfant(bernard,andre).
enfant(bernard,augustine).
enfant(becassine,augustine).

```
parent(Y,X) :- enfant(X,Y).
```

4) TD1

```
entier(zero).
entier(s(N)) :- entier(N).
```

```
pair(zero).
impair(s(zero)).
pair(s(s(N))) :- pair(N).

egal(s(N),s(N)) :- egal(N,N).

add3(zero,N2,N2) :- entier(N2).
add3(s(N1),N2,s(N3)) :- add(N1,N2,N3).


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

5) Les calculs

Le symbole « = » en prolog signifie l'unification et pas l'affectation. Pour affecter une valeur numérique à une variable, en évaluant mathématiquement le résultat, il faut utiliser « `is` ».

```
2+2=4.
2+2= :=4.
X=2+2.
x=X.
2+2 is 4.
4 is 2+2.
X is 2+2.
2+2 is X.

somme(X,Y,S) :- S is X+Y.

max2(X,Y,X) :- X>=Y.
max2(X,Y,Y) :- X<Y.

Afficher N fois 'bonjour'.
ecrit(0).
ecrit(N) :- N>0, write('bonjour'), nl, N1 is N-1, ecrit(N1).


Afficher les entiers
/* de N à 1 */
decroissant(0).
decroissant(N) :- N>0, write(N), nl, N1 is N-1, decroissant(N1).
/* de 1 à N */
croissant(0).
croissant(N) :- N>0, N1 is N-1, croissant(N1), write(N), nl.
```

```
Somme des N premiers entiers
som(0,0).
som(N,X) :- N>0, N1 is N-1, som(N1,X1), X is N+X1.


fibo(N,X) est vrai si X est la valeur de la suite de Fibonacci au rang N.
fibo(1,1).
fibo(2,1).
fibo(N,X) :- N>2, U is N-1, V is N-2, fibo(U,U1), fibo(V,V1),
         X is U1+V1.
```

6) Les listes

[] : liste vide
[T|R] : liste avec au moins un élément
[_|R] : liste dont on se fiche de savoir quel est la tête

 affiche(L) est vrai si tous les éléments de la liste L sont écrits.

```
affiche([]).
affiche([X|R]) :- write(X), nl, affiche(R).
```

 affiche2(L) est vrai si tous les éléments de la liste L sont écrits en ordre inverse.

```
affiche2([]).
affiche2([X|R]) :- affiche2(R), write(X), nl.
```

 Retrouver le premier élément d'une liste : premier1(L,X) est vrai si X est le premier élement de L.

```
premier1([X|_],X).
```

 Afficher le premier élément d'une liste (bis) : premier2(L) est vrai si le premier élément de la liste L est affiché (et aucun autre).

```
premier2([X|_]) :- write(X),nl.
```

 Retrouver le dernier élément d'une liste : dernier1(L,X) est vrai si X est le dernier élement de L.

```
dernier1([X],X).
dernier1([_|L],X) :- dernier1(L,X).
```

 Afficher le dernier élément d'une liste (bis) : dernier2(L) est vrai si le dernier élément de la liste L est affiché (et aucun autre).

```
dernier2([X]) :- write(X),nl.
dernier2([_|L]) :- dernier2(L).
```

 element(X,L) est vrai si X est élément de la liste L. **(member(X,L) le fait)**

```
element(X,[X|_]).
element(X,[_|R]) :- element(X,R).
```

 compte(L,N) est vrai si N est le nombre d'éléments dans la liste L.

```
compte([],0).
compte([_|R],N) :- compte(R,N1), N is N1+1, N>0.
```

 `somme(L,N)` est vrai si `N` est la somme des éléments de la liste d'entiers `L`.

```
somme([],0).
somme([X|R],N) :- somme(R,N1), N is N1+X.
```

 `nieme(N,L,X)` est vrai si `X` est le `N`-ème élément de la liste `L`.

```
nieme(1,[X|_],X) :- !.
nieme(N,[_|R],X) :- N1 is N-1, nieme(N1,R,X).
```

 `occurrence(L,X,N)` est vrai si `N` est le nombre de fois où `X` est présent dans la liste `L`.

```
occurrence([],_,0).
occurrence([X|L],X,N) :- occurrence(L,X,N1),N is N1+1.
occurrence([Y|L],X,N) :- X\==Y,occurrence(L,X,N).
```

7) Accumulateurs

Somme

```
%SommeA(L,Accu,Somme)

sommeA(L,S) :- sommeA(L,0,S).
sommeA([T|R],Accu,S) :- AccuN is Accu + T, sommeA(R, AccuN, S).
sommeA([],Accu,Accu).


somme([],0).
somme([T|R],S):- somme(R,S1), S is S1+T.
```

8) A connaitre

PERMUTATION

/** Permet d'obtenir toutes les permutations possibles de la liste
**la différence entre select et nth1 (hormis l'ordre) est que nth1 (correspondant à
**des insertions) prend pour premier paramètre l'emplacement de l'insertion.
**Rq : pour retenir l'ordre penser que NL est toujours en 2EME position
**explication nth1 : param1 pour placer à n'importe quelle position, NL pour la liste finale,
**X pour l'élément à insérer et LL pour la liste dans laquelle on veut l'insérer
**/

```
permutation([],[]).

permutation([X|L],NL) :- permutation(L,LL),  nth1(_,NL,X,LL).
```

OU

```
select(X,NL,LL).
```

CARRE MAGIQUE

/** = := est un égalité « classique »
**/

```
solution(L):-
	L=[X11,X12,X13,X21,X22,X23,X31,X32,X33],
	numlist(1,9,L2),
	permutation(L2,L),

	Ligne1 is X11+X12+X13,
	Ligne2 is X21+X22+X23,
	Ligne1 =:= Ligne2,
	Ligne3 is X31+X32+X33,
	Ligne1 =:= Ligne3,

	Colonne1 is X11+X21+X31,
	Colonne2 is X12+X22+X32,
	Colonne1 =:= Colonne2,
	Colonne3 is X13+X23+X33,
	Colonne1 =:= Colonne3,

	Ligne1 =:= Colonne1,

	Diag1 is X11+X22+X33,
	Diag2 is X13+X22+X31,
	Diag1 =:= Diag2,
	Diag1 =:= Colonne1.

print_carre([]).
print_carre([X1,X2,X3|L]) :-
	print(X1),print(X2),print(X3),print('\n'),print_carre(L).
```

CASSER UN CODE

/** Ce qui suit permet de casser le code à 9 chiffre possédant **différentes contraintes (cf TD)

**Le # permet de différencier la notation associée à la librairie de la notation normale

**clpr : le r signfie Réel

**#\/ est un ou

**label permet de trouver une solution correspondant aux conditions

**précédentes

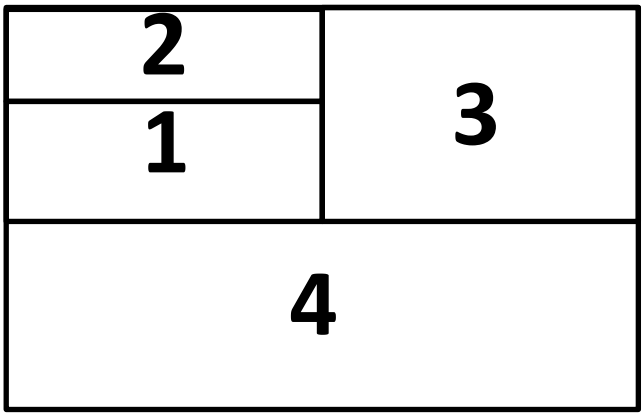
**

```
:-[library(clp/bounds)].
:-use_module(library('clp/bounds')).
:-use_module(library('clp/clpr')).

casse(ListeVar) :- ListeVar=[X1,X2,X3,X4,X5,X6,X7,X8,X9],
	ListeVar in 1..9,
	all_different(ListeVar),
	self_distinct(ListeVar,1),
```

```
        (X4-X6 #= X7) #\/ (X6-X4 #= X7),
        X1*X2*X3 #=  X8+X9,
        X2+X3+X6 #< X8,
        X9 #< X8,
        label(ListeVar).

self_distinct([],_).
self_distinct([H|T],N) :-
        H#\=N, %# pour le différent du bound sinon \n
        M is N+1,
        self_distinct(T,M).
```

CLP

/**On cherche à associer des couleurs au schema suivant pour **qu'aucun des polygones ne se touché.
**/




```
:-[library(clp/bounds)].
:-use_module(library('clp/bounds')).
:-use_module(library('clp/clpr')).
couleur(1,bleu).
couleur(2,jaune).
couleur(3,rouge).

colorier :- ListeVar=[C1,C2,C3,C4],
        ListeVar in 1..3,
        all_different([C1,C2,C3]),
        all_different([C1,C3,C4]),
        label(ListeVar), affiche(ListeVar).
```

```
affiche([]) :- nl.
affiche([T|R]) :- couleur(T,C), write('\n'), write(C), tab(2), affiche(R).
```

Cryptarithme

```
/** Le # permet de différencier la notation associée à la librairie de la notation normale
** ins signifie que l'on est dans un ensemble
**/
:-[library(clpfd)].
solution([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]):-
        L=[S,E,N,D,M,O,R,Y],
        L ins 0..9,
        1000*S+100*E+10*N+D +
        1000*M+100*O+10*R+E #=
        10000*M+1000*O+100*N+10*E+Y,
        M #\=0,
        S #\=0,
        all_distinct(L),
        label(L).
```

CONVERSION

```
/**Les accolades permettent de calculer C à partir de F OU F à partir de C
**/
convert(C,F):-
        C is (F-32)*5/9.
convert2(C,F):-
        {C = (F-32)*5/9}.
```

PREMIERS :

```
/** Le findall contient 3 paramètre ; le premier est "ce que l'on cherche", le second est "sous
**quelle condition", le troisième est "la liste dans laquelle on stocke les différentes valeurs
**possible du premier paramètre suivant les conditions.
**/
liste_diviseurs(N,L) :- findall(Div,(between(1,N,Div),N rem Div =:= 0),L).

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

liste_premiers(Borne,L) :- findall(N,(between(1,Borne,N),premier(N)),L).
```

FIBONACCI

Normal

```
fib(0,0).
fib(1,1).
fib(N,R) :-
```

```
        N>=2,
        N1 is N-1,
        fib(N1,R1),
        N2 is N-2,
        fib(N2,R2),
        R is R1+R2.
```

Avec ACCUMULATEUR

```
fib_efficace(0,1).
fib_efficace(1,1).
fib_efficace(N,R) :-
        N>=2,
        fib_efficace_aux(0,1,N,R).


fib_efficace_aux(_X,Y,1,Y).
fib_efficace_aux(X,Y,N,R) :-
        N>=2,
        N1 is N-1,
        NX is Y,
        NY is X+Y,
        fib_efficace_aux(NX,NY,N1,R).
```

RAPPEL :
Suite : 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13 ; 21 ; 34 …

ETOILE

```
etoile(0).
etoile(N) :- N>0, write(*), M1 is N-1, etoile(M1).
triangle(0).
triangle(D) :- D>0, nl, etoile(D), M is D-1, triangle(M).
```

Comprendre le CUT :

Il arrive que Prolog rencontre l'instruction !, connue sous le nom de « cut » ou « coupe-choix ». Ce symbole signifie qu'aucun retour arrière ne sera effectué sur les buts précédant le coupe-choix. Si les buts précédents retournaient plusieurs solutions, seule la solution courante sera conservée.
Empêche de retourner au nœud précédent dans un arbre.

SI ALORS SINON

```
/**avant le cut c'est le  « Si », derrière le cut et avant le « . » c'est un « alors », derrière le
**point on a un « sinon »
** truc(param,param) :- SI … ! ALORS …    .
**SINON truc(param,param) :- …    .
**/
elementGrandListe([],_,[]).
```

```
elementGrandListe([T|R],X,[T|L]) :- T>X, !, elementGrandListe(R,X,L).
elementGrandListe([T|R],X,L) :- elementGrandListe(R,X,L).

homme(socrate).
homme(allan).
homme(super).
mortel(benj).
mortel(X) :- homme(X), !.
différence avec
 mortel(X) :- homme(X).
```

NOMBRE PARFAIT

```
/** On écrit le prédicat somme qui permet de faire la somme des termes d'une liste.
** liste_diviseurs 2 prend en paramètre un nombre et une liste dans laquelle tous les
**diviseurs du nombre excepté ce dernier y sont stockés.
**nombreParfait vérifie que la somme des éléments de la liste récupérée fait bien le
**nombre.
**between exprime que Div va prendre toutes les valeurs entre 1 et N1
**/

somme([],0).
somme([X|R],N) :- somme(R,N1), N is N1+X.

liste_diviseurs2(N,L) :- N1 is N-1, findall(Div,(between(1,N1,Div),N rem Div =:= 0),L).
nombreParfait(N) :- liste_diviseurs2(N,L), somme(L,N).
```

NOMBRES AMICAUX :

```
/** la liste des diviseurs de A sans compter lui-même est placé dans L. On vérifie que la somme des
**éléments de L fait bien B. On fait pareil dans l'autre sens.
**/
nombreAmicaux(A,B) :- liste_diviseurs2(A,L), somme(L,B), liste_diviseurs2(B,L2),
somme(L2,A).
```