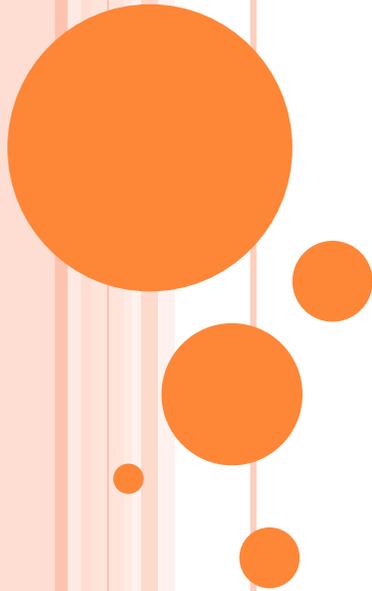


PREMIERS PAS EN PROLOG



Fonctionnement

Utilisation pour des bases de connaissances

Listes

PROGRAMMATION LOGIQUE

○ Origines :

- 1970, Marseille, Colmerauer
- Edimbourg, Warren

○ Bibliographie

- L. Sterling, E. Shapiro, L'art de Prolog, Masson
- Clocksin, Mellish, Programmer en Prolog, Eyrolles

LE LANGAGE PROLOG

- Langage d'expression des connaissances fondé sur le langage des prédicats du premier ordre
- Programmation déclarative :
 - L'utilisateur définit une base de connaissances
 - L'interpréteur Prolog utilise cette base de connaissances pour répondre à des questions

CONSTANTES ET VARIABLES

○ Constantes

- Nombres : 12, 3.5
- Atomes
 - Chaînes de caractères commençant par une minuscule
 - Chaînes de caractères entre " "
 - Liste vide []

○ Variables

- Chaînes de caractères commençant par une majuscule
- Chaînes de caractères commençant par _
- La variable « indéterminée » : _

TROIS SORTES DE CONNAISSANCES : FAITS, RÈGLES, QUESTIONS

- Faits : $P(\dots)$. avec P un prédicat
pere(jean, paul).
pere(albert, jean).
Clause de Horn réduite à un littéral positif
- Règles : $P(\dots) :- Q(\dots), \dots, R(\dots)$.
papy(X,Y) :- pere(X,Z), pere(Z,Y).
Clause de Horn complète
- Questions : $S(\dots), \dots, T(\dots)$.
pere(jean,X), mere(annie,X).
Clause de Horn sans littéral positif

RÉFUTATION PAR RÉOLUTION

○ Programme P

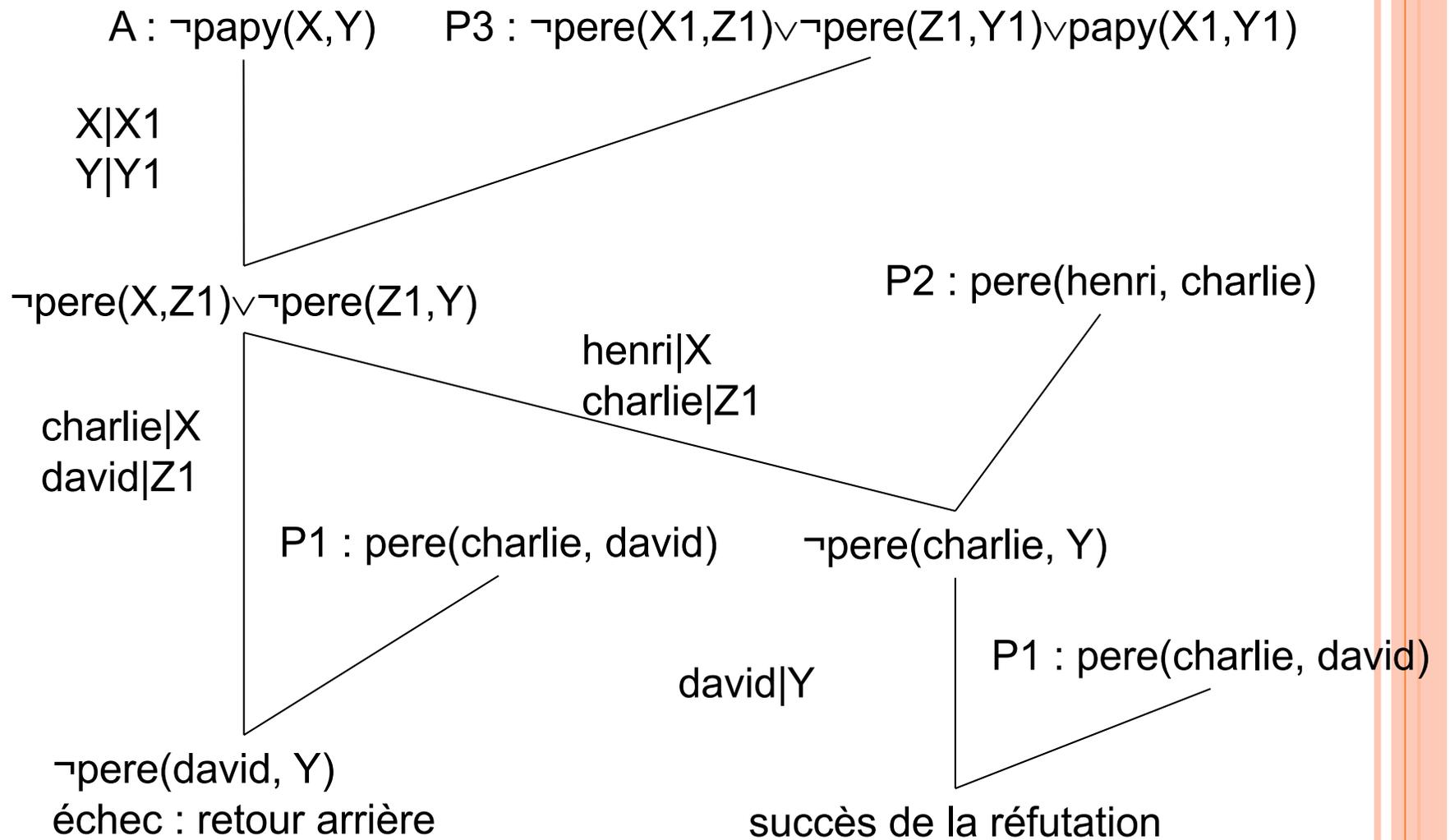
- P1 : pere(charlie, david).
- P2 : pere(henri, charlie).
- P3 : papy(X,Y) :- pere(X,Z), pere(Z,Y).

○ Appel du programme P

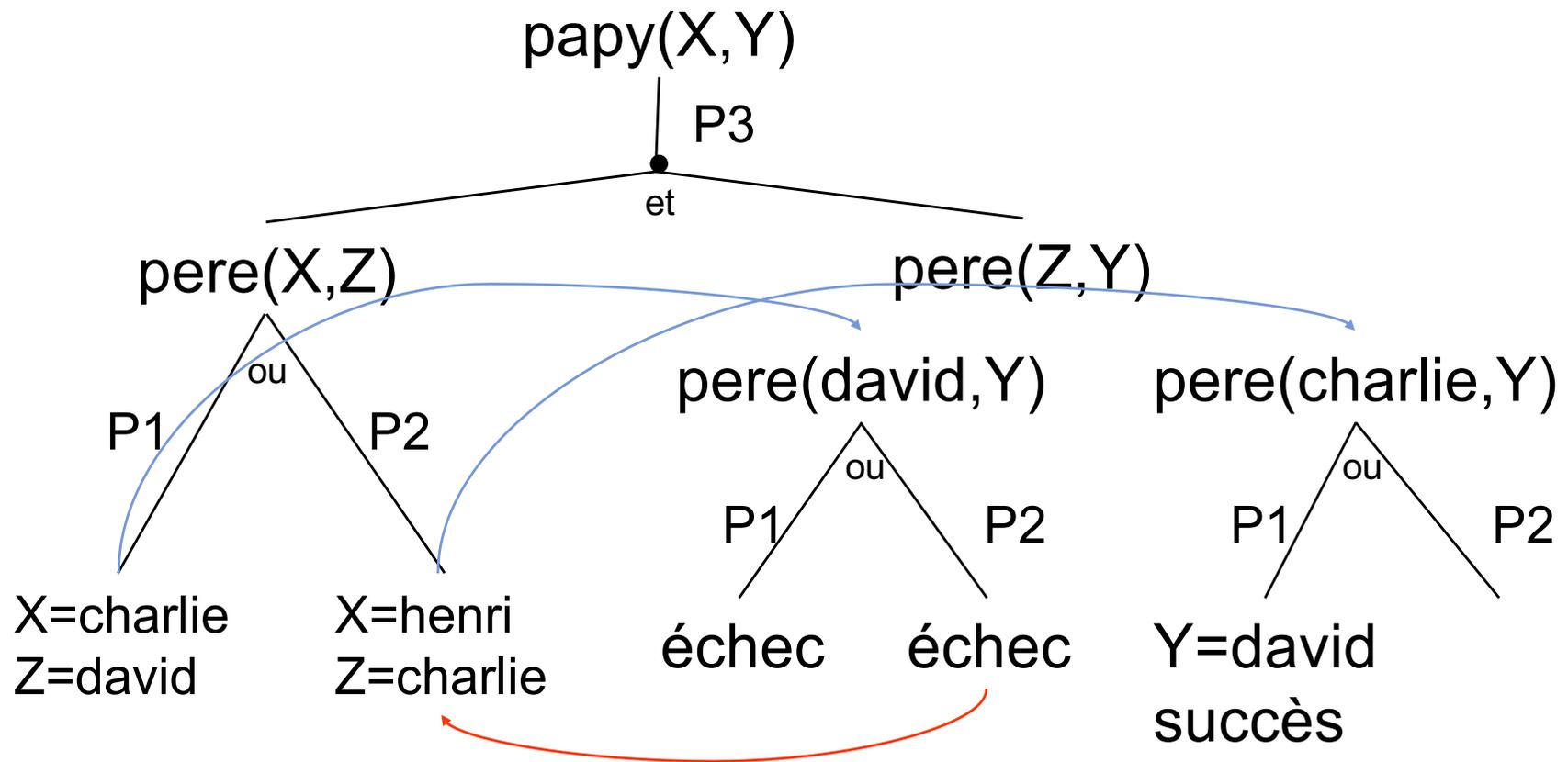
- A : papy(X,Y).

○ Réponse : X=henri, Y=david

GRAPHE DE RÉSOLUTION



INTERPRÉTATION PROCÉDURALE : ARBRE ET-OU



MON PREMIER PROGRAMME (1)

```
pere(charlie,david).  
pere(henri,charlie).  
papy(X,Y) :- pere(X,Z), pere(Z,Y).
```

```
lirispc1$ swiprolog
```

```
Welcome to SWI-Prolog (Version 3.3.0)
```

```
Copyright (c) 1993-1999 University of Amsterdam.
```

```
All rights reserved.
```

```
For help, use ?- help(Topic). or ?-apropos(Word).
```

```
?- [pere].
```

```
% pere compiled 0.00 sec, 824 bytes
```

```
true.
```

```
?- listing.
```

```
pere(charlie, david).
```

```
pere(henri, charlie).
```

```
papy(A, B) :-
```

```
    pere(A, C),
```

```
    pere(C, B).
```

```
true.
```

MON PREMIER PROGRAMME (2)

```
?- pere(charlie,david).  
true.  
?- pere(charlie,henri).  
false.  
?- pere(X,Y).  
X = charlie  
Y = david  
true.  
?- pere(X,Y).  
X = charlie  
Y = david ;  
X = henri  
Y = charlie
```

```
?- papy(x,y).  
false.  
?- papy(X,Y).  
X = henri  
Y = david  
  
?- papy(henri,X).  
X = david  
true.  
?- halt.  
lirispc1$
```

ORDRE DES RÉPONSES

```
pere(charlie, david).
pere(henri, charlie).
pere(david, luc).

mere(sophie, charlie).
mere(anne, david).

parents(E, P, M) :-
    pere(P, E),
    mere(M, E).
```

```
?- parents(X,Y,Z).
```

```
X = david
Y = charlie
Z = anne ;
```

```
X = charlie
Y = henri
Z = sophie ;
```

```
false.
```

Prolog parcourt le paquet de clauses de haut en bas, chaque clause étant parcourue de gauche à droite

EXERCICES

- Construire l'arbre ET-OU permettant à Prolog de donner l'ensemble des réponses satisfaisant la requête $\text{parents}(X, Y, Z)$.
- On définit le programme suivant :
b(1). b(2). c(3). c(4). d(5). d(6).
a(X, Y, Z) :- b(X), c(Y), d(Z).
 - Donner toutes les réponses à la requête $a(X, Y, Z)$ dans l'ordre où Prolog les fournit.

L'ÉNIGME POLICIÈRE EN PROLOG

- On dispose des informations suivantes :
 - La secrétaire déclare qu'elle a vu l'ingénieur dans le couloir qui donne sur la salle de conférences
 - Le coup de feu a été tiré dans la salle de conférences, on l'a donc entendu de toutes les pièces voisines
 - L'ingénieur affirme n'avoir rien entendu
- On souhaite démontrer que si la secrétaire dit vrai, alors l'ingénieur ment

L'ÉNIGME POLICIÈRE EN PROLOG

- Ordre 1 : un individu entend un bruit s'il se trouve dans une pièce voisine de celle où le bruit a été produit

entend(Ind,Bruit) :- lieu(Ind,Piece1), lieu(Bruit,Piece2),
voisin(Piece1,Piece2).

- Faits relatifs à l'énigme :

voisin(couloir,salle_de_conf).

lieu(coup_de_feu,salle_de_conf).

lieu(ingenieur,couloir) :- secretaire_dit_vrai.

ingenieur_ment :- entend(ingenieur,coup_de_feu).

L'ÉNIGME POLICIÈRE EN PROLOG

- Hypothèse
secretaire_dit_vrai.
- Pour la démonstration, on pose la requête :
ingenieur_ment.

SYMBOLES FONCTIONNELS

- La fonction « femme de Jean » est différente du prédicat femme(marie,jean).
nom(femme(jean),marie).
age(femme(jean),25).
- On peut parler de la femme de jean, mais pas la « calculer »

PROGRAMMATION RÉCURSIVE

- Un programme récursif est un programme qui s'appelle lui-même
- Exemple : factorielle
 - $\text{factorielle}(1) = 1$ (Cas d'arrêt)
 - $\text{factorielle}(n) = n * \text{factorielle}(n-1)$ si $n \neq 1$

Appel récursif



POUR ÉCRIRE UN PROGRAMME RÉCURSIF

○ Il faut :

- Choisir sur quoi faire l'appel récursif
- Choisir comment passer du résultat de l'appel récursif au résultat que l'on cherche
- Choisir le(s) cas d'arrêt

BOUCLAGE

```
maries(jean, sophie).
maries(philippe,
stephanie).
maries(A, B) :-
    maries(B, A).
```

```
?- maries(jean,sophie).
true.
?- maries(sophie,jean).
true.
?- maries(X,Y).
X = jean
Y = sophie ;
X = philippe
Y = stephanie ;
X = sophie
Y = jean ;
X = stephanie
Y = philippe ;
X = jean
Y = sophie ;
...
```

```
maries(jean, sophie).
maries(philippe,
stephanie).
sont_maries(A, B) :-
    maries(A, B).
sont_maries(A, B) :-
    maries(B, A).
```

```
?- sont_maries(X,Y).
X = jean
Y = sophie ;
X = philippe
Y = stephanie ;
X = sophie
Y = jean ;
X = stephanie
Y = philippe ;
false.
?-
```

ARITHMÉTIQUE

- Comparaisons : $:=$, $=$, \neq , $>$, $<$, \geq , \leq
- Affectation : is
 - ?- X is 3+2.
 - X=5
- Fonctions prédéfinies : -, +, *, /, ^, mod, abs, min, max, sign, random, sqrt, sin, cos, tan, log, exp, ...

UN EXEMPLE : FACTORIELLE (1)

```
fact(1, 1).
fact(N, R) :-
    Nm1 is N-1,
    fact(Nm1, Rnm1),
    R is Rnm1*N.
```

```
?- fact(5,R).
R = 120 ;
ERROR: Out of local
stack
Exception: (36,276)
_G4661 is-36263-1 ?
abort
% Execution Aborted
```

```
?- trace, fact(3,R).
Call: (8) fact(3, _G237) ? creep
^ Call: (9) _G308 is 3-1 ? creep
^ Exit: (9) 2 is 3-1 ? creep
Call: (9) fact(2, _G306) ? creep
^ Call: (10) _G311 is 2-1 ? creep
^ Exit: (10) 1 is 2-1 ? creep
Call: (10) fact(1, _G309) ? creep
Exit: (10) fact(1, 1) ? creep
^ Call: (10) _G314 is 2*1 ? creep
^ Exit: (10) 2 is 2*1 ? creep
Exit: (9) fact(2, 2) ? creep
^ Call: (9) _G237 is 3*2 ? creep
^ Exit: (9) 6 is 3*2 ? creep
Exit: (8) fact(3, 6) ? creep
R = 6 ;
Redo: (10) fact(1, _G309) ? creep
^ Call: (11) _G314 is 1-1 ? creep
^ Exit: (11) 0 is 1-1 ? creep
Call: (11) fact(0, _G312) ? creep
^ Call: (12) _G317 is 0-1 ? creep
^ Exit: (12) -1 is 0-1 ? creep
Call: (12) fact(-1, _G315) ? creep
^ Call: (13) _G320 is-1-1 ? creep
^ Exit: (13) -2 is-1-1 ? creep
```

Il faut faire des cas exclusifs

UN EXEMPLE : FACTORIELLE (2)

```
fact(1, 1).  
fact(N, R) :-  
    fact(Nm1, Rnm1),  
    Nm1 is N-1,  
    R is Rnm1*N.
```

```
?- fact(3,R).
```

```
ERROR: Arguments are not  
sufficiently instantiated
```

```
^ Exception: (9) 1 is  
_G241-1 ? creep  
Exception: (8)  
fact(_G241, _G255) ? creep  
Exception: (7) fact(3,  
_G195) ? creep  
% Execution Aborted
```

```
?- 5 is X-1.
```

```
ERROR: Arguments  
are not  
sufficiently  
instantiated
```

```
% Execution  
Aborted
```

```
?- plus(3,2,5).  
true.
```

```
?- plus(X,2,5).  
X = 3  
true.
```

EXERCICE

- Définir un prédicat calculant le $n^{\text{ième}}$ terme de la suite : $u_0 = 2$, $u_n = 2u_{n-1} + 3$

UNE FACTORIELLE AVEC ACCUMULATEUR

```
fact(N,R) :- fact(N,1,R).  
  
fact(1,R,R).  
fact(N,I,R) :- N>1,  
               Nm1 is N-1,  
               NewI is N*I,  
               fact(Nm1,NewI,R).
```

```
?- trace, fact(3,N).  
   Call: (7) fact(3, _G234) ?  
   creep  
   Call: (8) fact(3, 1, _G234)  
   ? creep  
   Call: (9) 3>1 ? creep  
   Exit: (9) 3>1 ? creep  
^ Call: (9) _G305 is 1*3 ?  
   creep  
^ Exit: (9) 3 is 1*3 ? creep
```

```
^ Call: (9) _G308 is 3-1 ? creep  
^ Exit: (9) 2 is 3-1 ? creep  
   Call: (9) fact(2, 3, _G234) ?  
   creep  
   Call: (10) 2>1 ? creep  
   Exit: (10) 2>1 ? creep  
^ Call: (10) _G311 is 3*2 ?  
   creep  
^ Exit: (10) 6 is 3*2 ? creep  
^ Call: (10) _G314 is 2-1 ?  
   creep  
^ Exit: (10) 1 is 2-1 ? creep  
   Call: (10) fact(1, 6, _G234) ?  
   creep  
   Call: (11) 1>1 ? creep  
   Fail: (11) 1>1 ? creep  
   Redo: (10) fact(1, 6, _G234) ?  
   creep  
   Exit: (10) fact(1, 6, 6) ?  
   creep
```

N = 6

COMPARAISON ET UNIFICATION DE TERMES

- Vérifications de type : var, nonvar, integer, float, number, atom, string, ...
- Comparer deux termes :
 - $T1 == T2$ réussit si T1 est **identique** à T2
 - $T1 \neq T2$ réussit si T1 n'est pas **identique** à T2
 - $T1 = T2$ **unifie** T1 avec T2
 - $T1 \neq T2$ réussit si T1 n'est pas **unifiable** à T2

DIFFÉRENTS PRÉDICATS DE COMPARAISON

:= \=

?- A is 3, A:=3.

A = 3.

?- A is 3, A:=2+1.

A = 3.

?- a=\b.

ERROR

== \==

?- A is 3, A==3.

A = 3.

?- A is 3, A==2+1.

false.

?- a\==b.

true.

?- A==3.

false.

?- p(A)\==p(1).

true.

= \=

?- A=3.

A = 3.

?- p(A)\=p(1).

false.

LISTES

- Liste vide : []
- Cas général : [Tete|Queue]
[a,b,c] \equiv [a|[b|[c|[]]]]

EXEMPLES

- $[X|L] = [a,b,c] \rightarrow X = a, L = [b,c]$
- $[X|L] = [a] \rightarrow X = a, L = []$
- $[X|L] = [] \rightarrow \text{échec}$
- $[X,Y]=[a,b,c] \rightarrow \text{échec}$
- $[X,Y|L]=[a,b,c] \rightarrow X = a, Y = b, L = [c]$
- $[X|L]=[X,Y|L2] \rightarrow L=[Y|L2]$

SOMME DES ÉLÉMENTS D'UNE LISTE DE NOMBRES

```
/* somme(L, S) L liste de nb donnée, S nb résultat */  
somme([], 0).  
somme([X|L], N) :- somme(L, R), N is R+X.
```

?- somme([1,2,3,5],N).

N = 11

EXERCICE

- Définir un prédicat `ajoute1(L,L1)` où `L` est une liste de nombres, et `L1` une liste identique où tous les nombres sont augmentés de 1.

VARIABLE INDÉTERMINÉE (1)

```
/* ieme(L,I,X) L liste donnée, I entier donné,  
   X elt res */  
ieme([X|L],1,X).  
ieme([X|L],I,R) :- I>1, Im1 is I-1, ieme(L,Im1,R).
```

[ieme].

Warning: (/Users/nath/Enseignement/Option Prolog/ieme:2):

Singleton variables: [L]

Warning: (/Users/nath/Enseignement/Option Prolog/ieme:3):

Singleton variables: [X]

% ieme compiled 0.01 sec, 736 bytes

true.

VARIABLE INDÉTERMINÉE (2)

```
/* ieme(L,I,X) L liste donnée, I entier donné,  
   X elt res */  
ieme([X|_],1,X).  
ieme([_|L],I,R) :- I>1, Im1 is I-1, ieme(L,Im1,R).
```

?- ieme([a,b,c,d],2,N).

N = b ;

false.

TEST OU GÉNÉRATION

```
/* appart(X,L) X elt donné,  
   L liste donnée */  
appart(X,[X|_]).  
appart(X,[_|L]) :- appart(X,L).
```

```
?- appart(a,[b,a,c]).  
true.  
?- appart(d,[b,a,c]).  
false.  
?- appart(X,[b,a,c]).  
X = b ;  
X = a ;  
X = c ;  
false.  
?- trace, appart(X,[b,a,c]).  
Call: (7) appart(_G284, [b, a, c])  
? creep  
Exit: (7) appart(b, [b, a, c]) ?  
creep  
X = b ;
```

```
Redo: (7) appart(_G284, [b, a,  
c]) ? creep  
Call: (8) appart(_G284, [a,  
c]) ? creep  
Exit: (8) appart(a, [a, c]) ?  
creep  
X = a ;  
Redo: (8) appart(_G284, [a,  
c]) ? creep  
Call: (9) appart(_G284, [c])  
? creep  
Exit: (9) appart(c, [c]) ?  
creep  
X = c ;  
Redo: (9) appart(_G284, [c])  
? creep  
Call: (10) appart(_G284, [])  
? creep  
Fail: (10) appart(_G284, [])  
? creep  
false.
```

LE PRÉDICAT MEMBER

○ Le prédicat appart est prédéfini en Prolog

○ Il est très utile :

- ?- member(c,[a,z,e,c,r,t]).

true

- ?- member(X,[a,z,e,r,t]).

X = a ; X = z ; X = e ; X = r ; X = t.

- ?- member([3,V],[[4,a],[2,n],[3,f],[7,g]]).

V = f .

UTILISATION DU PRÉDICAT APPEND

Append est le prédicat prédéfini pour la concaténation de listes

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

```
L = [a, b, c, d, e]
```

Il est complètement symétrique et peut donc être utilisé pour

- Trouver le dernier élément d'une liste :

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

```
X = d
```

- Couper une liste en sous-listes :

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

```
L1 = [b, c, d],
```

```
L2 = [e, t]
```

DÉFINITION D'UN PRÉDICAT : QUESTIONS À SE POSER

- Comment vais-je l'utiliser ?
- Quelles sont les données ?
- Quels sont les résultats ?
- Est-ce souhaitable qu'il y ait plusieurs solutions ?

- Si l'on veut une seule solution, il faut faire des cas exclusifs

EXERCICE

- Définir le prédicat `renverse(L1,L2)` satisfait si la liste `L2` est miroir de la liste `L1`.
- Construire l'arbre de résolution des requêtes suivantes :
 - `renverse([a,b,c],L)`
 - `renverse(L,[a,z,e])`.
- Définir une version avec accumulateur du prédicat `renverse`.
- Construire l'arbre de résolution des deux requêtes précédentes.