## Cours de Prolog

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

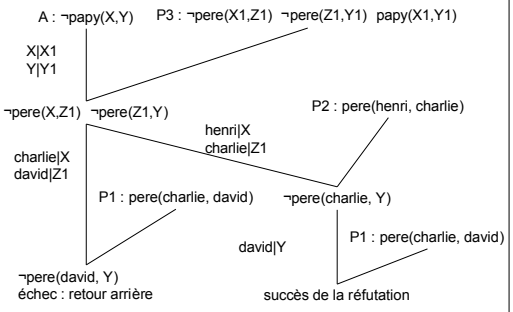


## Résolution par réfutation

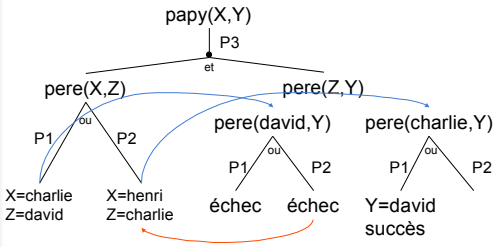
- Programme P
  - P1 : pere(charlie, david).
  - P2 : pere(henri, charlie).
  - P3 : papy(X,Y) :- pere(X,Z), pere(Z,Y).
- Appel du programme P
  - A : papy(X,Y).
- Réponse : X=henri, Y=david



## 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).
```

```
lisipc2$ 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).
?- [pere1].
% pere1 compiled 0.00 sec, 824 bytes
Yes
?- listing.
pere(charlie, david).
pere(henri, charlie).
papy(A, B) :-
        pere(A, C),
        pere(C, B).
Yes
```



## Mon premier programme (2)

```
?- pere(charlie,david).
Yes
?- pere(charlie,henri).
No
?- pere(X,Y).
X = charlie
Y = david
Yes
?- pere(X,Y).
X = charlie
Y = david ;
X = henri
Y = charlie ;
No
```

```
?- papy(x,y).
No
?- papy(X,Y).
X = henri
Y = david ;
No
?- papy(henri,X).
X = david
Yes
?- halt.
lisipc2$
```



## Ordre des réponses

```
?- listing.
pere(charlie, david).
pere(henri, charlie).
pere(david, luc).

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

parents(A, B, C) :-
        pere(B, A),
        mere(C, A).
Yes
```

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

X = david
Y = charlie
Z = anne ;

X = charlie
Y = henri
Z = sophie ;

No
```

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



## 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 »



## Prolog n'est pas équivalent à la logique

- ancetre(X,X).
- ancetre(X,Y) :- pere(X,Z), ancetre(Z,Y).
  et pas :
  ancetre(X,Y) :- ancetre(Z,Y), pere(X,Z).

## Bouclage

```
?- listing.
maries(jean, sophie).
maries(philippe, stephanie).
maries(A, B) :-
        maries(B, A).
Yes
?- maries(jean,sophie).
Yes
?- maries(sophie,jean).
Yes
?- maries(X,Y).
X = jean
Y = sophie ;
X = philippe
Y = stephanie ;
X = sophie
Y = jean ;
X = stephanie
Y = philippe ;
X = jean
Y = sophie ;
```

```
?- listing.
maries(jean, sophie).
maries(philippe, stephanie).
sont_maries(A, B) :-
        maries(A, B).
sont_maries(A, B) :-
        maries(B, A).
Yes
?- sont_maries(X,Y).
X = jean
Y = sophie ;
X = philippe
Y = stephanie ;
X = sophie
Y = jean ;
X = stephanie
Y = philippe ;
No
?-
```



## Arithmétique

- Comparaisons : >, <, >=, =<, =:=, =\=
- Affectaction : 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)

```
?- listing.
fact(1, 1).
fact(A, B) :-
        C is A-1,
        fact(C, D),
        B is A*D.
Yes

?- 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
```



## Un exemple : factorielle (2)

```
?- listing.
fact(1, 1).
fact(A, B) :-
       fact(C, D),
        C is A-1,
        B is A*D.
Yes
?- fact(4,B).
ERROR: Arguments are not
  sufficiently instantiated
^  Exception: (8) 1 is
  _G201-1 ? creep
```

```
?- 5 is X-1.
ERROR: Arguments
   are not
   sufficiently
   instantiated
% Execution
   Aborted
?- plus(3,2,5).
Yes
?- plus(X,2,5).
X = 3
Yes
```



## Une factorielle « itérative »

```
?- listing.
fact(A, B) :-
       fact(A, 1, B).
fact(A, B, C) :-
       A>1,
       D is B*A,
       E is A-1,
       fact(E, D, C).
fact(1, A, A).
Yes
?- 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\==T2 réussit si T1 n'est pas identique à T2

  T1=T2 unifie T1 avec T2

  T1\=T2 réussit si T1 n'est pas unifiable à T2

## Listes

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



## Exemples

- [X|L] = [a,b,c] □ X = a, L = [b,c]
- [X|L] = [a] □ X = a, L = []
- [X|L] = [] □ échec
- [X,Y]=[a,b,c] □ échec
- [X,Y|L]=[a,b,c] □ X = a, Y = b, L = [c]
- [X|L]=[X,Y|L2] □ L=[Y|L2]



## Somme des éléments d'une liste de nombres

```
?- [somme].
% somme compiled 0.01 sec, 692 bytes
Yes
?- listing.
somme([], 0).
somme([A|B], C) :-
        somme(B, D),
        C is D+A.
Yes
?- somme([1,2,3,5],N).
N = 11 ;
No
```



## Variable indéterminée (1)

```
?- [ieme].
Warning: (/Users/nathalie/Enseignement/IA/Cours/Prolog/ieme:1):
        Singleton variables: [L]
Warning: (/Users/nathalie/Enseignement/IA/Cours/Prolog/ieme:1):
        Singleton variables: [X]
% ieme compiled 0.01 sec, 544 bytes

?- listing.
ieme([A|B], 1, A).
ieme([A|B], C, D) :-
        E is C-1,
        ieme(B, E, D).
Yes
```





## Variable indéterminée (2)

```
ieme([X|_],1,X).
ieme([_|L],I,Y):- Im1 is I-1, ieme(L,Im1,Y).

?- ieme([a,b,c,d],2,N).
N = b ;
No
```



## Test ou génération

```
?- listing.
appart(A, [A|B]).
appart(A, [B|C]) :-
        appart(A, C).
Yes
?- appart(a,[b,a,c]).
Yes
?- appart(d,[b,a,c]).
No
?- appart(X,[b,a,c]).
X = b ;
X = a ;
X = c ;
No
?- 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
   No
```

## 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 on veut une seule solution, il faut faire des cas exclusifs



## Chaînes de caractères

- Une chaîne de caractères est représentée par une liste de codes ASCII :
  ?- X="toto".
  X = [116, 111, 116, 111]
- Pour visualiser la chaîne :
  ?- name(S, [116, 111, 116, 111]).
  S = toto
- On manipule donc les chaînes avec des opérations de listes



## Exemple : les mutants

```
non_vide([_|_]).

mutant(S) :-
   animal(D), animal(F),
   append(Debut,Milieu,D),
   non_vide(Debut),
   non_vide(Milieu),
   append(Milieu,_,F),
   append(Debut,F,M),
   name(S,M).

animal("alligator").
animal("lapin").
animal("tortue").
animal("pintade").
animal("cheval").
```

```
?- mutant(X).

X = alligatortue ;

X = lapintade ;

X = chevalligator ;

X = chevalapin ;

No
```



## Entrées-Sorties

nl
put(Char)
get(Char)
write(Term)
read(Term)



## Boucles mues par échec

Deux prédicats prédéfinis :
  true : réussit toujours
  fail : échoue toujours

?- appart(X,[a,b,c]), write_ln(X), fail.
a
b
c
No



## Trouver toutes les solutions

findall(Variable, But, Liste)

?- findall(X, member(X,[a,b,c]), R).
R = [a, b, c]

## Manipuler des termes

?- functor(pere(pierre,paul), F, A)
F = pere
A = 2
?- functor(X, pere, 2).
X = pere(A, B)

?- arg(2, pere(pierre, paul), X).
X = paul
?- arg(2, pere(pierre, X), paul).
X = paul

?- X=..[pere, pierre, paul].
X = pere(pierre, paul)
?- pere(pierre, paul)=..X.
X = [pere, pierre, paul]



## Manipuler des programmes (1)

- Trouver une clause : clause(Tete, Queue)

appart(A,[A|B]).
appart(A,[B|C]) :- appart(A,C).

?- clause(appart(X,Y),Q).
X = A
Y = [A|B]
Q = true;
X = A
Y = [B|C]
Q = appart(A,C);
no



## Manipuler des programmes (2)

- Ajouter une clause

  assert(pere(pierre, paul)).
  assert((papy(X,Y) :- pere(X,Z), pere(Z,Y))).
- Enlever une clause

  ?- retract(pere(X,Y)).
  X = pierre
  Y = paul ;
  X = paul
  Y = jean ;
  No
  ?- retractall(pere(_,_)).



## Manipuler des programmes (3)

?- toto(X).
ERROR : Undefined procedure : toto/1

:- dynamic toto/1, tata/2.

?- toto(X).
No



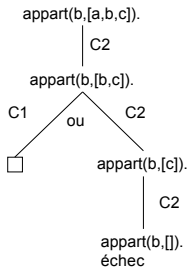
## Points de choix

C1 : appart(X,[X|_]).
C2 : appart(X,[_|L]) :- appart(X,L).

```
?- trace, appart(b,[a,b,c]),fail.
   Call: (8) appart(b, [a, b, c]) ? creep
   Call: (9) appart(b, [b, c]) ? creep
   Exit: (9) appart(b, [b, c]) ? creep
   Redo: (9) appart(b, [b, c]) ? creep
   Call: (10) appart(b, [c]) ? creep
   Call: (11) appart(b, []) ? creep
   Fail: (11) appart(b, []) ? creep
No
```



## Représentation par un graphe ET/OU



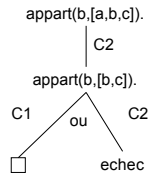
## Première solution : faire des cas exclusifs

appart(X, [X|_]).
appart(X, [Y|L]) :- X\=Y, appart(X, L).

```
?- trace, appart(b,[a,b,c]),fail.
   Call: (7) appart(b, [a, b, c]) ? creep
   Call: (8) b\=a ? creep
   Exit: (8) b\=a ? creep
   Call: (8) appart(b, [b, c]) ? creep
   Exit: (8) appart(b, [b, c]) ? creep
   Exit: (7) appart(b, [a, b, c]) ? creep
   Redo: (8) appart(b, [b, c]) ? creep
   Call: (9) b\=b ? creep
   Fail: (9) b\=b ? creep
   Fail: (8) appart(b, [b, c]) ? creep
   Fail: (7) appart(b, [a, b, c]) ? creep
No
```



## Représentation par un graphe ET/OU





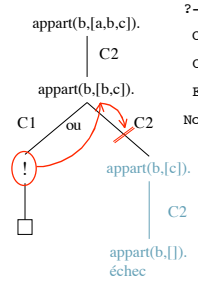

## Deuxième solution : la coupure

La coupure interdit le retour arrière sur des points de choix.

C1 : appart(X,[X|_]) :- ! .
C2 : appart(X,[_|L]) :- appart(X,L).



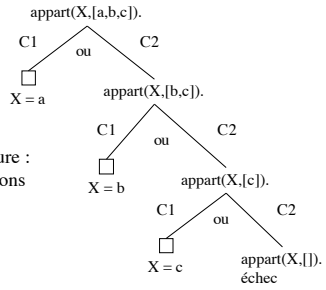
## Coupure verte : éliminer des points de choix inutiles




```
?- trace, appart(b,[a,b,c]),fail.
Call: (8) appart(b, [a, b, c]) ?
      creep
Call: (9) appart(b, [b, c]) ?
      creep
Exit: (9) appart(b, [b, c]) ?
      creep
No
```



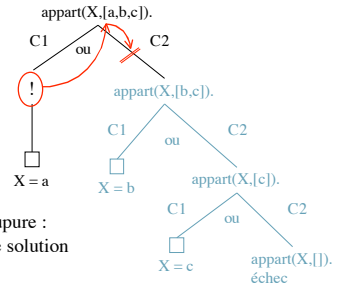
## Coupure rouge : modifier les solutions (1)




Sans coupure :
trois solutions



## Coupure rouge : modifier les solutions (2)




Avec coupure :
une seule solution

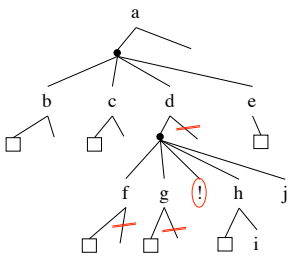
## Test, génération, coupure …

- Il est souvent utile de faire deux versions d'un prédicat
  - Une version sans coupure utilisable en génération
  - Une version avec coupure utilisable en test ou pour produire une seule solution



## Coupure : quels points de choix supprimés ?

a :- b, c, d, e.
a :- …
b.
b :- …
c.
c :- …
d :- f, g, !, h, j.
(d :- …)
f.
(f :- …)
g.
(g :- …)
h.
h :- i.
e.




La coupure supprime les points de choix sur les sommets aînés et sur le sommet père



## Exemple

```
bb(1).
bb(2).
cc(3).
cc(4).
dd(5).
dd(6).
aa(X,Y,Z) :-  bb(X), cc(Y), dd(Z).
```



## Coupure rouge : omettre des conditions

```
delete([],X,[]).
delete([X|L],X,R) :- !, delete(L,X,R).
delete([Y|L],X,[Y|R]) :- Y\==X,  delete(L,X,R).

delete([],X,[]).
delete([X|L],X,R) :- !, delete(L,X,R).
delete([Y|L],X,[Y|R]) :- delete(L,X,R).
```



## If-Then-Else

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.

(P -> Q ; R).
```



## Négation par échec

not(But) est vrai si But échoue

```
not(B) :- B, !, fail.
not(B).
```