Bases de l'Intelligence Artificielle

CM4-5 : PROgrammation LOGique

Marie Lefevre

2025-2026

Université Claude Bernard Lyon 1

Sondage Tomuss…

Qui a déjà fait du Prolog ?

Qui a un ordi perso pour les TP ?

⇒ Merci de répondre ☺

Origines

- 1972 : Prolog I
  - Alain COLMERAUER (Université Aix-Marseille)
    - Découverte de la programmation logique et premier interpréteur
  - Robert A. KOWALSKY (Edinburgh University)
    - Cadre théorique et premier compilateur
- 1982 : Prolog II
  - M. VAN CANEGUEM et al.
- ….
- Nombreux interpréteurs Prolog avec des syntaxes différentes
- …
- 1996 : Prolog IV
  - SWI-Prolog : syntaxe, conventions et interpréteur utilisé dans ce cours

Différents modes de programmation

- Langages de style impératif
  - C, ADA, PASCAL, JAVA (objet)…
  - ➢ On décrit la suite d'actions permettant de passer des données aux résultats
- Langages de style fonctionnel
  - Lisp, Scheme, CAML ...
  - ➢ On décrit le résultat comme une composition de fonctions agissant sur les données
- Langage de style déclaratif
  - Prolog
  - ➢ On décrit le problème (les objets, leurs propriétés, leurs relations)
  - ➢ Mais pas comment on le résout
  - ➢ Souvent utilisé en Intelligence Artificielle

Le langage PROLOG

- Langage d'expression des connaissances
  fondé sur la logique des prédicats du premier ordre
- Implémentation du principe de résolution

- Programmation déclarative
  - L'utilisateur définit une base de connaissances
    (des faits, des règles)
    - Pas d'instruction, pas d'affectation, pas de boucles explicites
    - Uniquement des clauses exprimant des faits, des règles,
      des questions
  - L'interpréteur PROLOG utilise cette base de connaissances
    pour répondre à des questions / résoudre des problèmes

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

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

- Programme Prolog
  - Ensemble de faits et de règles qui décrivent un problème

- Faits : P(…). avec P un prédicat
  - Clause de Horn réduite à un littéral positif
    - pere(jean, paul).
    - pere(albert, jean).

Trois sortes de connaissances : faits, règles, questions

- Règles : P(…) :- Q_1(…), …, Q_n(…).
  - Clause de Horn complète
  - Correspond en logique à $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
    P : tête de clause, Q_1,…,Q_n queue de clause
  - Paquet de clauses = clauses ayant le même nom de littéral de tête

- Exemple de règle
  - Relation grand-père en logique du premier ordre :
    - $\forall X \forall Y\ (\exists Z\ pere(X,Z) \wedge (pere(Z,Y) \vee mere(Z,Y))) \rightarrow papy(X,Y)$
    - $\forall X \forall Y \forall Z\ (((pere(X,Z) \wedge pere(Z,Y)) \rightarrow papy(X,Y)) \vee ((pere(X,Z) \wedge mere(Z,Y)) \rightarrow papy(X,Y)))$
  - Formulation Prolog :
    - papy(X,Y) :- pere(X,Z), pere(Z,Y).
    - papy(X,Y) :- pere(X,Z), mere(Z,Y).

Trois sortes de connaissances : faits, règles, questions

- Questions : S(…), …, T(…).
  - Clause de Horn sans littéral positif
    - pere(jean,X), mere(annie,X).
    - :- pere(jean,X), mere(annie,X).

- Exécuter un programme = résoudre un but
  - Trouver toutes les valeurs des variables qui apparaissent dans la question et qui amènent à une contradiction

  ➢ Faire une preuve

L'unification

- Procédé par lequel on essaie de rendre deux formules identiques en donnant des valeurs aux variables qu'elles contiennent

- Le résultat est un unificateur
  - Par exemple : {Jean/X, Paul/Y}
  - Pas forcément unique, on cherche l'unificateur le plus général

- L'unification peut réussir ou échouer
  - P(X , X) et P(2, 3) ne peuvent pas être unifiés

Déclaratif vs Procédural

- Soit la clause P :- Q, R.

- Aspect déclaratif :
  - P est vrai si Q et R sont vrais
  - ➢ L'ordre des clauses n'a pas d'importance

- Aspect procédural :
  - Pour résoudre P, résoudre d'abord le sous-problème Q, puis le sous-problème R
  - ➢ L'interpréteur considère les clauses les unes après les autres, dans l'ordre dans lesquelles elles se trouvent, celui-ci a de l'importance

Littéraux ordonnés

- P.
  - P est toujours vrai
- P :– Q_1, Q_2, ... , Q_n.
  - Pour résoudre P, il faut résoudre dans l'ordre Q_1 puis Q_2 puis ... puis Q_n
- :– Q_1, Q_2, ... , Q_n.
  - Pour résoudre la question, il faut résoudre dans l'ordre Q_1 puis Q_2 puis ... puis Q_n

Réfutation par résolution

- Programme P

P1 : pere(charlie, david).

P2 : pere(henri, charlie).

P3 : papy(X,Y) :- pere(X,Z), pere(Z,Y).

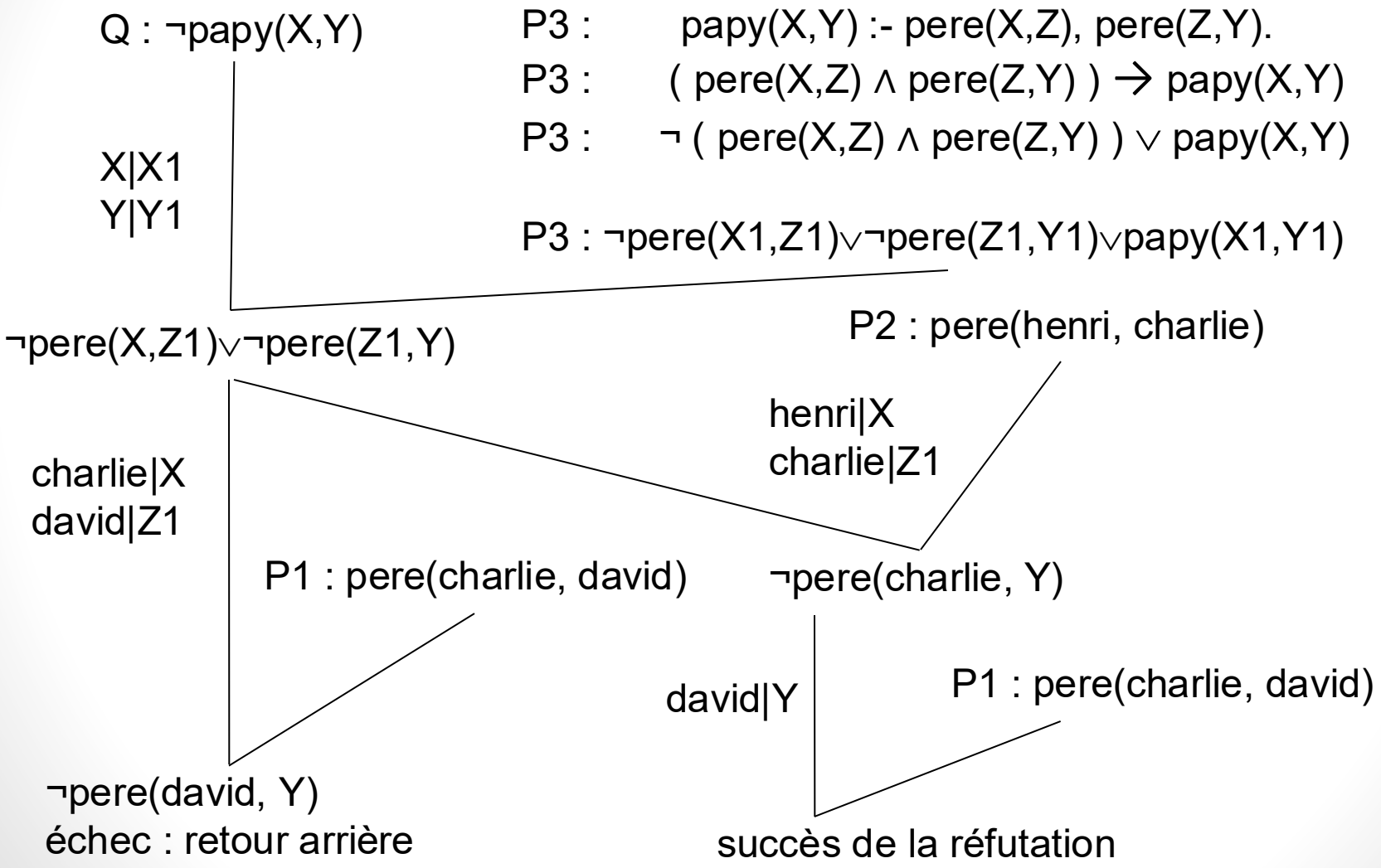
- Appel du programme P

Q : papy(X,Y).

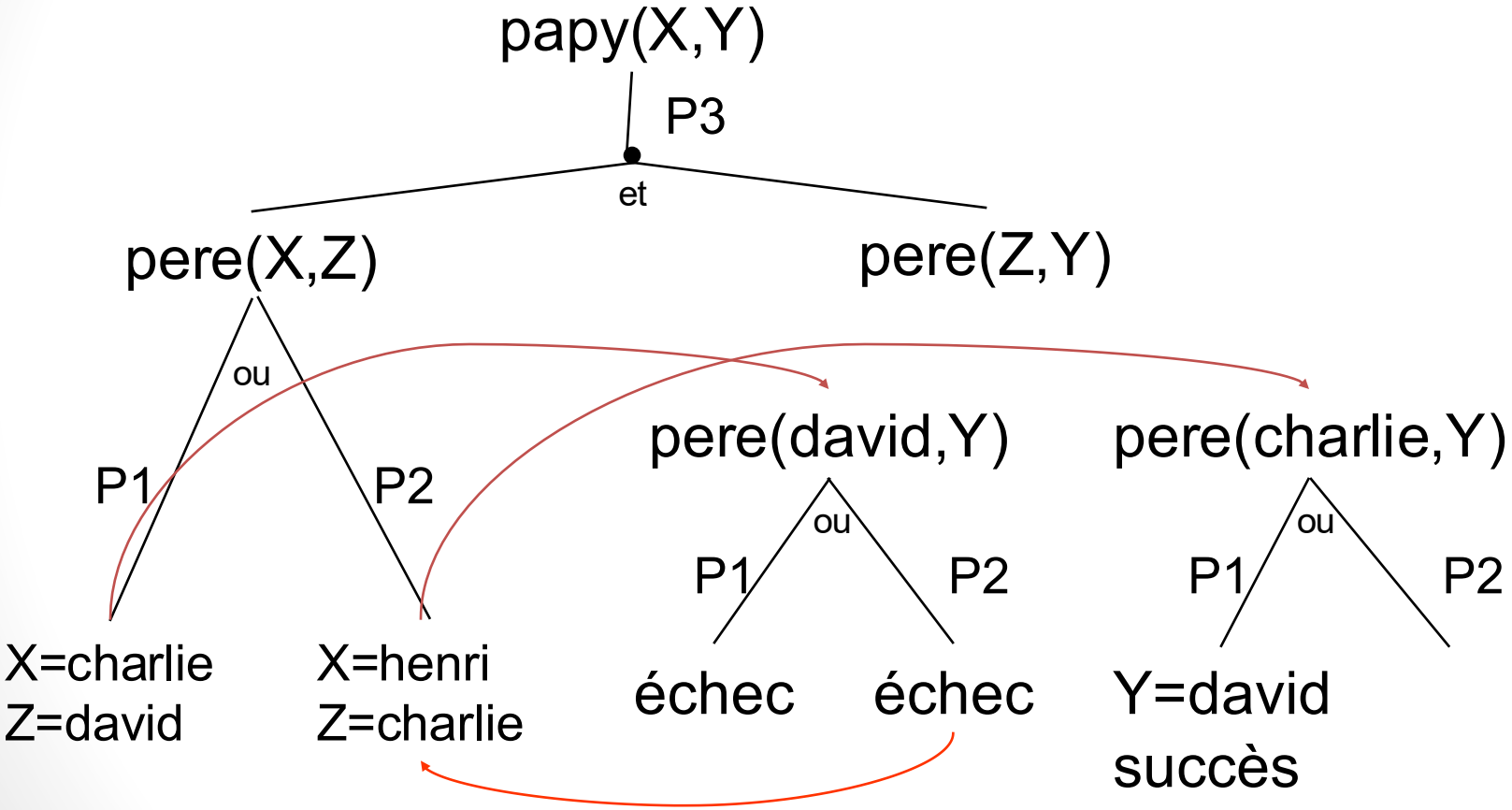
- Réponse

X=henri, Y=david

Graphe de résolution



Interprétation procédurale : arbre ET-OU



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

Fonctionnement de l'interpréteur

- L'interpréteur prend un premier but
- Il essaie de le résoudre = de l'unifier avec une tête de clause
  - Si il réussit, il cherche à résoudre la queue de la clause instanciée par l'unification
  - En résolvant dans l'ordre chacun des littéraux
- Puis il passe au prochain but en attente
- En cas d'échec, retour arrière au dernier choix effectué
- Quand une solution est trouvée, on peut
  - Arrêter la recherche
  - Demander une autre solution
- La résolution est terminée s'il n'y a plus de littéraux à résoudre et plus de choix à traiter

Un programme Prolog

papy.pl

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

Swi-Prolog

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.2.6)
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?-
```

Des questions Prolog

```
?- [papy].
% papy compiled 0.00 sec, 4 clauses
true.

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

papy(A, C) :-
        pere(A, B),
        pere(B, C).
true.

?-
```

```
?- pere(charlie,david).
true.

?- pere(charlie,henri).
false.

?- pere(X,Y).
X = charlie,
Y = david. // touche Entrée

?- pere(X,Y).
X = charlie,
Y = david ; // point virgule
X = henri,
Y = charlie.

?-
```

Des questions Prolog

```
?- papy(x,y). // minuscule

false.

?- papy(X,Y). // majuscule
X = henri,
Y = david.

?- papy(henri,X).
X = david.

?- halt. // sortie
```

Exercice : L'énigme policière

- 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

- On souhaite démontrer que si la secrétaire dit vrai, alors l'ingénieur ment

- Hypothèse

```
secretaire_dit_vrai.
```

- Pour la démonstration, on pose la requête :

```
ingenieur_ment.
```

L'énigme policière en Prolog

- A partir des informations suivantes :
  - Le coup de feu a été tiré dans la salle de conférences, on l'a donc entendu de toutes les pièces voisines

- Règle de la logique du 1er ordre
  - Un individu entend un bruit s'il se trouve dans une pièce voisine de celle où le bruit a été produit

```
entend(Individu,Bruit) :-
    lieu(Individu,Piece1),
    lieu(Bruit,Piece2),
    voisine(Piece1,Piece2).
```

L'énigme policière en Prolog

- A partir 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

- Base de faits :

```
voisine(couloir,salle_de_conf).
lieu(ingenieur,couloir) :- secretaire_dit_vrai.
lieu(coup_de_feu,salle_de_conf).
ingenieur_ment :- entend(ingenieur,coup_de_feu).
```

L'énigme policière en Prolog

enigme.pl

```
voisine(couloir,salle_de_conf).
lieu(coup_de_feu,salle_de_conf).
lieu(ingenieur,couloir) :- secretaire_dit_vrai.
ingenieur_ment :- entend(ingenieur,coup_de_feu).
entend(Individu,Bruit) :- lieu(Individu,Piece1), lieu(Bruit,Piece2),
         voisine(Piece1,Piece2).
secretaire_dit_vrai.
```

Swi-Prolog

```
[trace]  ?- ingenieur_ment.
   Call: (6) ingenieur_ment ? creep
   Call: (7) entend(ingenieur, coup_de_feu) ? creep
   Call: (8) lieu(ingenieur, _G2981) ? creep
   Call: (9) secretaire_dit_vrai ? creep
   Exit: (9) secretaire_dit_vrai ? creep
   Exit: (8) lieu(ingenieur, couloir) ? creep
   Call: (8) lieu(coup_de_feu, _G2981) ? creep
   Exit: (8) lieu(coup_de_feu, salle_de_conf) ? creep
   Call: (8) voisin(couloir, salle_de_conf) ? creep
   Exit: (8) voisin(couloir, salle_de_conf) ? creep
   Exit: (7) entend(ingenieur, coup_de_feu) ? creep
   Exit: (6) ingenieur_ment ? creep
true.
```

Programmation récursive

- Un programme récursif est
  un programme qui s'appelle lui-même

- Exemple : factorielle
  - factorielle(1) = 1 ← Cas d'arrêt
  - factorielle(n) = n * factorielle(n-1) si n≠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

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

?- 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 ;
...
```

```
 Call: (7) maries(_G2307, _G2308) ? creep
   Exit: (7) maries(jean, sophie) ? creep
X = jean,
Y = sophie ;
   Redo: (7) maries(_G2307, _G2308) ? creep
   Exit: (7) maries(philippe, stephanie) ?
creep
X = philippe,
Y = stephanie ;
   Redo: (7) maries(_G2307, _G2308) ? creep
   Call: (8) maries(_G2308, _G2307) ? creep
   Exit: (8) maries(jean, sophie) ? creep
   Exit: (7) maries(sophie, jean) ? creep
X = sophie,
Y = jean ;
   …
```

Bouclage

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

?- 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 ;
...
```

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

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

Influence de l'ordre des clauses

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).

ancetre(X, Z) :- parent(X,Z).
ancetre(X, Z) :- parent(X,Y), ancetre(Y,Z).

ancetre2(X, Z) :- parent(X,Y), ancetre2(Y,Z).
ancetre2(X, Z) :- parent(X,Z).

ancetre3(X, Z) :- parent(X,Z).
ancetre3(X, Z) :- ancetre3(X,Y), parent(Y,Z).

ancetre4(X, Z) :- ancetre4(X,Y), parent(Y,Z).
ancetre4(X, Z) :- parent(X,Z).
```

Influence de l'ordre des clauses

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).

ancetre(X, Z) :-
        parent(X,Z).
ancetre(X, Z) :-
        parent(X,Y),
        ancetre(Y,Z).
```

```
?- ancetre(thomas, pierre).
true ;
false.

?- trace, ancetre(thomas, pierre).
   Call: (7) ancetre(thomas, pierre) ? creep
   Call: (8) parent(thomas, pierre) ? creep
   Fail: (8) parent(thomas, pierre) ? creep
   Redo: (7) ancetre(thomas, pierre) ? creep
   Call: (8) parent(thomas, _G929) ? creep
   Exit: (8) parent(thomas, bernard) ? creep
   Call: (8) ancetre(bernard, pierre) ? creep
   Call: (9) parent(bernard, pierre) ? creep
   Exit: (9) parent(bernard, pierre) ? creep
   Exit: (8) ancetre(bernard, pierre) ? creep
   Exit: (7) ancetre(thomas, pierre) ? creep
true .
```

Influence de l'ordre des clauses

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).

ancetre2(X, Z) :-
        parent(X,Y),
        ancetre2(Y,Z).
ancetre2(X, Z) :-
        parent(X,Z).
```

```
?- ancetre2(thomas, pierre).
true ;
false.

?- trace, ancetre2(thomas, pierre).
   Call: (7) ancetre2(thomas, pierre) ? creep
   Call: (8) parent(thomas, _G1064) ? creep
   Exit: (8) parent(thomas, bernard) ? creep
   Call: (8) ancetre2(bernard, pierre) ?
creep
   Call: (9) parent(bernard, _G1064) ? creep
   Exit: (9) parent(bernard, anne) ? creep
   Call: (9) ancetre2(anne, pierre) ? creep
   Call: (10) parent(anne, _G1064) ? creep
   Fail: (10) parent(anne, _G1064) ? creep
   Redo: (9) ancetre2(anne, pierre) ? creep
   Call: (10) parent(anne, pierre) ? creep
   Fail: (10) parent(anne, pierre) ? creep
   Fail: (9) ancetre2(anne, pierre) ? creep
   Redo: (9) parent(bernard, _G1064) ? creep
   Exit: (9) parent(bernard, pierre) ? creep
   Call: (9) ancetre2(pierre, pierre) ? creep
   Call: (10) parent(pierre, _G1064) ? creep
   Exit: (10) parent(pierre, jean) ? creep
   Call: (10) ancetre2(jean, pierre) ? Creep
   Call: (11) parent(jean, _G1064) ? creep
Fail: (11) parent(jean, _G1064) ? creep
   Redo: (10) ancetre2(jean, pierre) ? creep
   Call: (11) parent(jean, pierre) ? creep
   Fail: (11) parent(jean, pierre) ? creep
   Fail: (10) ancetre2(jean, pierre) ? creep
   Redo: (9) ancetre2(pierre, pierre) ? creep
   Call: (10) parent(pierre, pierre) ? creep
   Fail: (10) parent(pierre, pierre) ? creep
   Fail: (9) ancetre2(pierre, pierre) ? creep
   Redo: (8) ancetre2(bernard, pierre) ?
creep
   Call: (9) parent(bernard, pierre) ? creep
   Exit: (9) parent(bernard, pierre) ? creep
   Exit: (8) ancetre2(bernard, pierre) ?
creep
   Exit: (7) ancetre2(thomas, pierre) ? creep
  true .
```

Influence de l'ordre des clauses

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).

ancetre3(X, Z) :-
        parent(X,Z).
ancetre3(X, Z) :-
        ancetre3(X,Y),
        parent(Y,Z).
```

```
?- ancetre3(thomas, pierre).
true ;
ERROR: Out of local stack

?- trace, ancetre3(thomas, pierre).
   Call: (7) ancetre3(thomas, pierre) ? creep
   Call: (8) parent(thomas, pierre) ? creep
   Fail: (8) parent(thomas, pierre) ? creep
   Redo: (7) ancetre3(thomas, pierre) ? creep
   Call: (8) ancetre3(thomas, _G1298) ? creep
   Call: (9) parent(thomas, _G1298) ? creep
   Exit: (9) parent(thomas, bernard) ? creep
   Exit: (8) ancetre3(thomas, bernard) ?
creep
   Call: (8) parent(bernard, pierre) ? creep
   Exit: (8) parent(bernard, pierre) ? creep
   Exit: (7) ancetre3(thomas, pierre) ? creep
true ;
   Redo: (9) parent(thomas, _G1298) ? creep
   Exit: (9) parent(thomas, lise) ? creep
   Exit: (8) ancetre3(thomas, lise) ? creep
   Call: (8) parent(lise, pierre) ? creep
   Fail: (8) parent(lise, pierre) ? creep
   Redo: (8) ancetre3(thomas, _G1298) ? creep
   Call: (9) ancetre3(thomas, _G1298) ? creep
   Call: (10) parent(thomas, _G1298) ? creep
   Exit: (10) pa
   Exit: (9) anc
creep
   Call: (9) par
   Exit: (9) pare
   Exit: (8) ancetre
   Call: (8) parent(anne, pierre) ? creep
   Fail: (8) parent(anne, pierre) ? creep
   Redo: (9) parent(bernard, _G1298) ? creep
   Exit: (9) parent(bernard, pierre) ? creep
   Exit: (8) ancetre3(thomas, pierre) ? creep
   Call: (8) parent(pierre, pierre) ? creep
   Fail: (8) parent(pierre, pierre) ? creep
   Redo: (10) parent(thomas, _G1298) ? creep
   Exit: (10) parent(thomas, lise) ? creep
   Exit: (9) ancetre3(thomas, lise) ? creep
   Call: (9) parent(lise, _G1298) ? creep
   Fail: (9) parent(lise, _G1298) ? creep
   Redo: (9) ancetre3(thomas, _G1298) ? creep
   Call: (10) ancetre3(thomas, _G1298) ?
creep
   Call: (11) parent(thomas, _G1298) ? creep
   Exit: (11) parent(thomas, bernard) ? creep
   …
   BRANCHE INFINIE
```

Influence de l'ordre des clauses

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).

ancetre4(X, Z) :-
        ancetre4(X,Y),
        parent(Y,Z).
ancetre4(X, Z) :-
        parent(X,Z).
```

```
?- ancetre4(thomas, pierre).
ERROR: Out of local stack

?- trace, ancetre4(thomas, pierre).
   Call: (7) ancetre4(thomas, pierre) ? creep
   Call: (8) ancetre4(thomas, _G1064) ? creep
   Call: (9) ancetre4(thomas, _G1064) ? creep
   Call: (10) ancetre4(thomas, _G1064) ? creep
   Call: (11) ancetre4(thomas, _G1064) ? creep
   Call: (12) ancetre4(thomas, _G1064) ? creep
   Call: (13) ancetre4(thomas, _G1064) ? creep
   Call: (14) ancetre4(thomas, _G1064) ? creep
   Call: (15) ancetre4(thomas, _G1064) ? creep
   Call: (16) ancetre4(thomas, _G1064) ? creep
   Call: (17) ancetre4(thomas, _G1064) ? creep
   Call: (18) ancetre4(thomas, _G1064) ? creep
   Call: (19) ancetre4(thomas, _G1064) ? creep
   Call: (20) ancetre4(thomas, _G1064) ? creep
   Call: (21) ancetre4(thomas, _G1064) ? creep
   Call: (22) ancetre4(thomas, _G1064) ? creep
   Call: (23) ancetre4(thomas, _G1064) ? creep
   Call: (24) ancetre4(thomas, _G1064) ? creep
   Call: (25) ancetre4(thomas, _G1064) ? creep
   Call: (26) ancetre4(thomas, _G1064) ? creep
   Call: (27) ancetre4(thomas, _G1064) ? creep
   Call: (28) ancetre4(thomas, _G1064) ? creep
```

```
Call: (29) anc
Call: (30) anc
Call: (31) anc
Call: (32) anc
Call: (33) ance
Call: (34) ancetre
Call: (35) ancetre4(thomas, _G1064) ? creep
Call: (36) ancetre4(thomas, _G1064) ? creep
Call: (37) ancetre4(thomas, _G1064) ? creep
Call: (38) ancetre4(thomas, _G1064) ? creep
Call: (39) ancetre4(thomas, _G1064) ? creep
Call: (40) ancetre4(thomas, _G1064) ? creep
Call: (41) ancetre4(thomas, _G1064) ? creep
Call: (42) ancetre4(thomas, _G1064) ? creep
Call: (43) ancetre4(thomas, _G1064) ? creep
Call: (44) ancetre4(thomas, _G1064) ? creep
Call: (45) ancetre4(thomas, _G1064) ? creep
Call: (46) ancetre4(thomas, _G1064) ? creep
Call: (47) ancetre4(thomas, _G1064) ? creep
Call: (48) ancetre4(thomas, _G1064) ? creep
Call: (49) ancetre4(thomas, _G1064) ? creep
Call: (50) ancetre4(thomas, _G1064) ? creep
Call: (51) ancetre4(thomas, _G1064) ? Creep
…
BRANCHE INFINIE
```

Influence de l'ordre des clauses

- Plan déclaratif : les quatre versions sont équivalentes
- Plan procédural : seules *ancetre* et *ancetre2* sont correctes

➢ Pourquoi?

- Conclusion :
  - Mettre les cas les plus simples d'abord (non récursif)
    - *parent* plutôt que *ancetre*
  - Ordonner avec les clauses et les littéraux
    - priorité à *parent*

Arithmétique

- Comparaison

Arithmétique	Prolog
X < Y	X < Y.
X ≤ Y	X =< Y.
X = Y	X =:= Y.
X ≠ Y	X =\= Y.
X ≥ Y	X >= Y.
X > Y	X > Y.

- Fonctions prédéfinies

  -, +, *, /, ^

  mod, abs, min, max,

  sing, random, sqrt,

  sin, cos, tan, log, exp,

  .....

Unification vs Affectation

?- X=4+5.

X = 4+5.

?- X=4+5, Y=5+4, X=Y.

false.

- Les résultats sont différents :
  Prolog n'a pas calculé le résultat mais a juste unifié X avec le terme '4+5'
  et Y avec '5+4'

?- X is 3+2.

X=5

?-X is 4+5, Y is 5+4, X=Y.

true.

- L'affectation se fait avec le prédicat **is/2**

Opérations arithmétiques

- Évaluer un terme représentant une expression arithmétique revient à appliquer les opérateurs *via* une affectation

?- X = 1 + 1 + 1.

X = 1 + 1 + 1 (ou X = +(+(1, 1), 1)).

?- X = 1 + 1 + 1, Y is X.

X = 1 + 1 + 1, Y = 3.

?- X is 1 + 1 + a.

erreur (a n'est pas un nombre)

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
  - Exemples :
    - ?- p(A)==p(1). → false.
    - ?- p(A)=p(1). → A=1.
    - ?- p(A)\==p(1). → true.
    - ?- p(A)\=p(1). → false.

Egalité arithmétique (=:=), comparaison (==), unification (=)

?- X is 1 + 1 + Z.
ERROR: Z non instancié à un nombre

?- Z = 2, X is 1 + 1 + Z.
Z = 2
X = 4

?- 1 + 2 =:= 2 + 1.
true.

?- 1 + 2 = 2 + 1.
false.

?- 1 + 2 = 1 + 2.
true.

?- 1 + X = 1 + 2.
X = 2.

?- 1 + X =:= 1 + 2.
ERROR: …

?- 1 + 2 == 1 + 2.
true.

?- 1 + 2 == 2 + 1.
false.

?- 1 + X == 1 + 2.
false.

?- 1 + a == 1 + a.
true.

Exos sur asker.univ-lyon1.fr pour vous entrainer

Un exemple : factorielle V1

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

?- 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 V2

```
?- listing.
fact(1, 1).
fact(A, B) :-
  fact(C, D), C is A-1, B is A*D.
true.
 ?- trace, fact(3, R).
    Call: (7) fact(3, _G824) ? creep
    Call: (8) fact(_G940, _G941) ? creep
    Exit: (8) fact(1, 1) ? creep
    Call: (8) 1 is 3-1 ? creep
    Fail: (8) 1 is 3-1 ? creep
    Redo: (8) fact(_G940, _G941) ? creep
    Call: (9) fact(_G940, _G941) ? creep
    Exit: (9) fact(1, 1) ? creep
    Call: (9) 1 is _G933-1 ? creep
ERROR: is/2: Arguments are not
sufficiently instantiated
```

```
?- 5 is X-1.
ERROR: Arguments
  are not
  sufficiently
  instantiated
% Execution
  Aborted

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

- Attention aux affectations

Un exemple : factorielle V3

```
?- listing.
facto(1, 1).
facto(A, B) :-
 A > 1,
  C is A-1,
  facto(C, D),
  B is A*D.
true.

?- facto(3,N).
N = 6 ;
false.
```

Factorielle avec accumulateur

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

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

Les listes

- Structure très importante en Prolog
- Peut contenir tout type de termes
- Entre crochets, éléments séparés par des virgules
  - Liste vide : [ ]
  - Cas général : [Tete|Queue]
  - [a,b,c] ≡ [a|[b|[c|[ ]]]]

- Exemples :
  - [a,b,c] [] [Y,b,[1,2,3]] [Y,b,Z]

Unification des listes

- Une liste entière :
  - ?- X = [a,b,c].
    X=[a,b,c]
    true.
- Certains éléments de la liste :
  - ?-[a,b,c]=[X,Y,Z].
    X=a Y=b Z=c
    true.
  - ?-[a,b,c]=[X,Y,c].
    X=a Y=b
    true.
  - ?- [a,b,c] = [X,Y,b].
    false.

Unification des listes

- Avec des listes imbriquées :
  - ?- [a,[2,2],c] = [a,Y,c] .
    Y=[2,2]
    true.
  - ?- [X,b,c] = [a,Y,c] .
    X=a
    Y=b
    true.
  - ?- [a,b,[c,d]] = [a,b,[c,X]] .
    X=d
    true.

Exemple d'unification de listes

- [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]

Exemple : somme des éléments d'une liste de nombres

```
?- [somme].
% somme compiled 0.01 sec, 692 bytes
true.
?- listing.
somme([], 0).
somme([A|B], C) :-
        somme(B, D),
        C is D+A.
true.
?- somme([1,2,3,5],N).
N = 11
```

Variable indéterminée (1)

```
[ieme].
Warning: (…/ieme:1):
        Singleton variables: [L]
Warning: (…/ieme:1):
        Singleton variables: [X]
% ieme compiled 0.01 sec, 736 bytes
true.
?- listing.
ieme([X|L], 1, X).
ieme([X|L], N, R) :-
        N>1,
        N2 is N-1,
        ieme(L, N2, R).
true.
```

Variable indéterminée (2)

```
?- listing.

ieme([X|_], 1, X).
ieme([_|L], N, R) :-
        N>1,
        N2 is N-1,
        ieme(L, N2, R).
true.

?- ieme([a,b,c,d],2,N).
N = b ;
false.
```

Test ou génération ?

```
?- listing.
appart(A, [A|B]).
appart(A, [B|C]) :-
        appart(A, C).
true.
?- 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/2

- 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 .
```

Le prédicat append/3

- 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(L2,L3,[b,c,a,d,e]),append(L1,[a],L2).

    L2 = [b, c, a] L3 = [d, e] L1 = [b, c]

Trouver toutes les solutions

- findall(Variable, But, Liste)

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

?- findall(Y, papy(X,Y), R).
R = [david, marie, paul].

?- findall(X, papy(X,Y), R).
R = [henri, henri, henri].

?- findall([X,Y], papy(X,Y), R).
R = [[henri, david], [henri, marie],
[henri, paul]].
```

```
pere(charlie,david).
pere(charlie,marie).
pere(charlie,paul).
pere(henri,charlie).

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

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.

```
ajoute1([],[]).
ajoute1([X|L],[X2|L1]) :-
          X2 is X+1,
          ajoute1(L, L1).

?- ajoute1([1,2,3], R).
R = [2, 3, 4].
```

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

Chaîne de caractères

- Une chaîne de caractères est représentée par une liste de codes ASCII :
  - "toto" équivaut à [116, 111, 116, 111]

- Pour manipuler ou visualiser la chaîne :
  name(chaineCaracteres, listeCaracteres)
  - ?- name(S, [116, 111, 116, 111]).
    S = toto
  - ?- name(S, "toto").
    S = toto

  - ?- name("toto", S).
    S = [116, 111, 116, 111].

- On manipule donc les chaînes avec des opérations de listes

Exemple : les mutants

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

mutant(S) :-
  animal(MotD),
  animal(MotF),
  name(MotD,D),
  name(MotF,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 ;

false.
```

Entrées - Sorties

- Saut de ligne
  - nl
- Affichage de M espaces
  - tab(M)
- Lecture d'un caractère et unification dans Char
  - Avec retour chariot : get(Char)
  - Sans retour chariot : get_single_char(Char)
- Lecture d'un terme (élémentaire ou composé) et unification dans Term
  - read(Term)
- Affichage d'un caractère
  - put(Char)
- Affichage d'un terme (élémentaire ou composé)
  - write(Term)

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

Boucles mues par échec

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

- ?- member(X,[a,b,c]), write_ln(X), fail.

  a

  b

  c

  false.

  ➢ Toujours échec donc pas d'unification réussie avec X.

  ➢ Permet de faire des boucles en Prolog

Exemple : conjuguer les verbes du premier groupe

```
?- conjugue("chanter").
je chante
tu chantes
il chante
nous chantons
vous chantez
ils chantent
true
```

Exemple : conjuguer les verbes du premier groupe

```
conjugue(MotInfinitif) :- name(MotInfinitif,LInfinitif),
       racine(LInfinitif,LRacine), termine(LRacine).

racine(LInfinitif,LRacine) :-
       append(LRacine,[101,114],LInfinitif).

terminaison("je","e").
terminaison("tu","es").
terminaison("il","e").
terminaison("nous","ons").
terminaison("vous","ez").
terminaison("ils","ent").

termine(LRacine) :-
    terminaison(MotDebut,MotFin),
    write(MotDebut), write(' '),
    name(MotFin,LFin),
    append(LRacine,LFin,LVerbe),
    name(MotVerbe,LVerbe), write(MotVerbe),nl,
    fail.
termine(_).
```

De quoi va-t-on parler ?

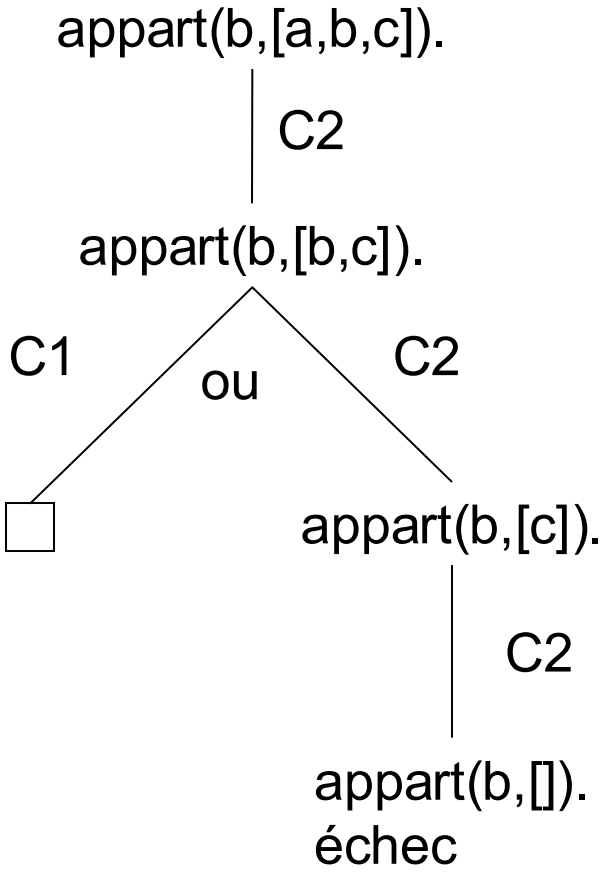
- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

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
false.
```

Représentation par un graphe ET/OU

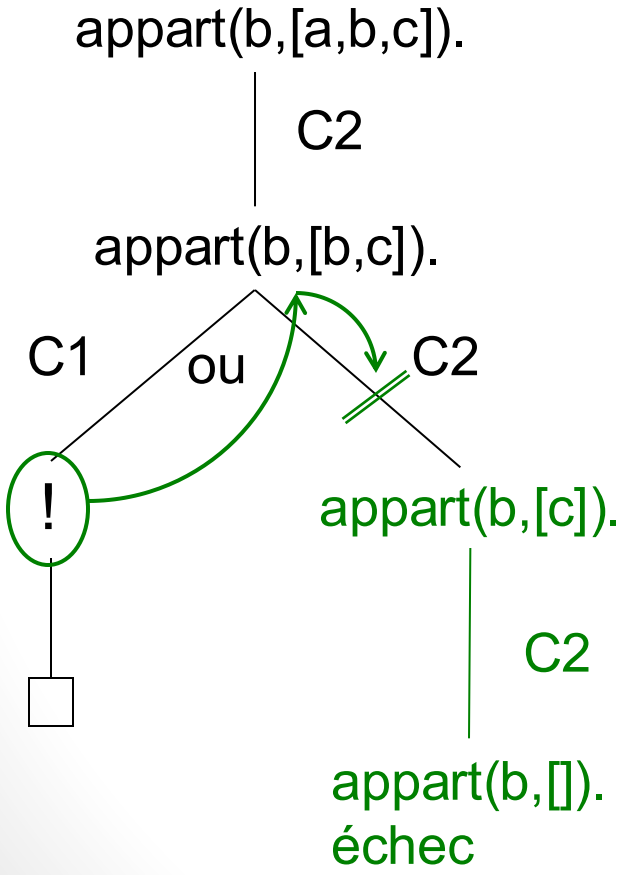


Le prédicat cut

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

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

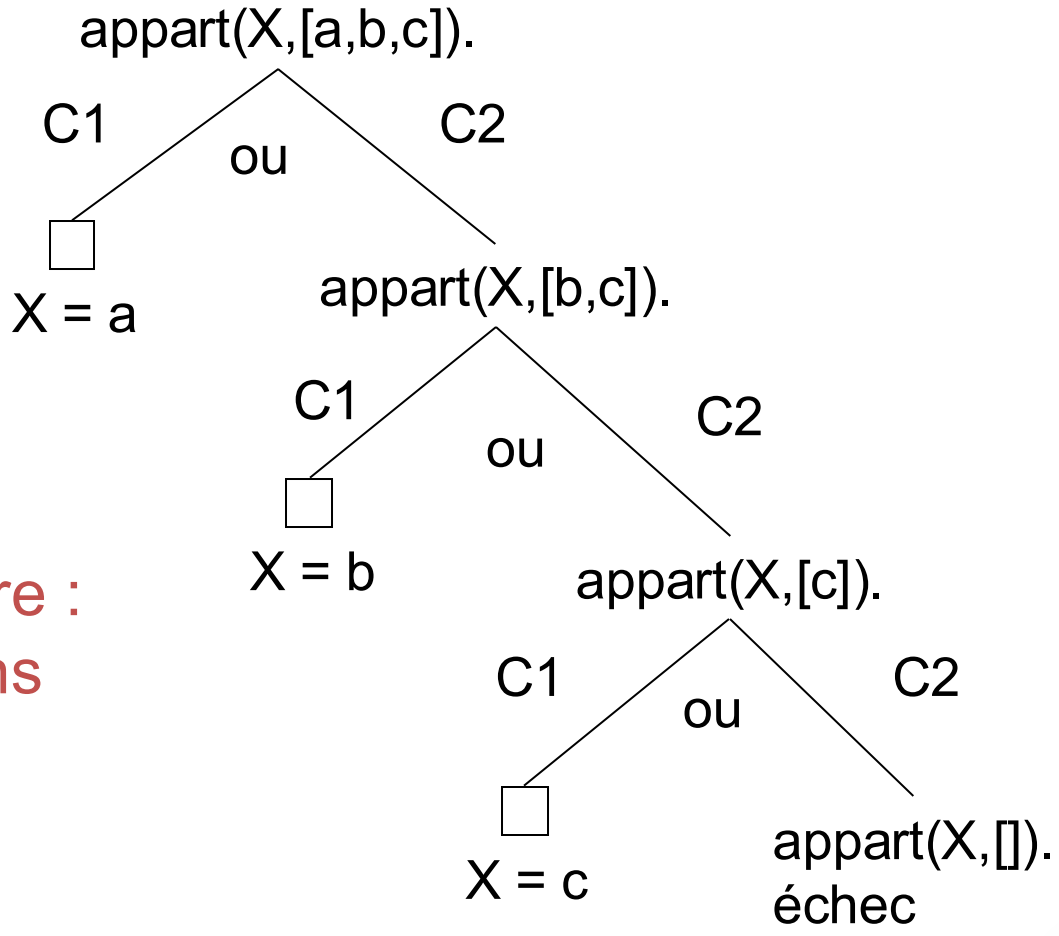
Coupure pour éliminer des points de choix inutiles



?- trace, appart(b,[a,b,c]),fail.

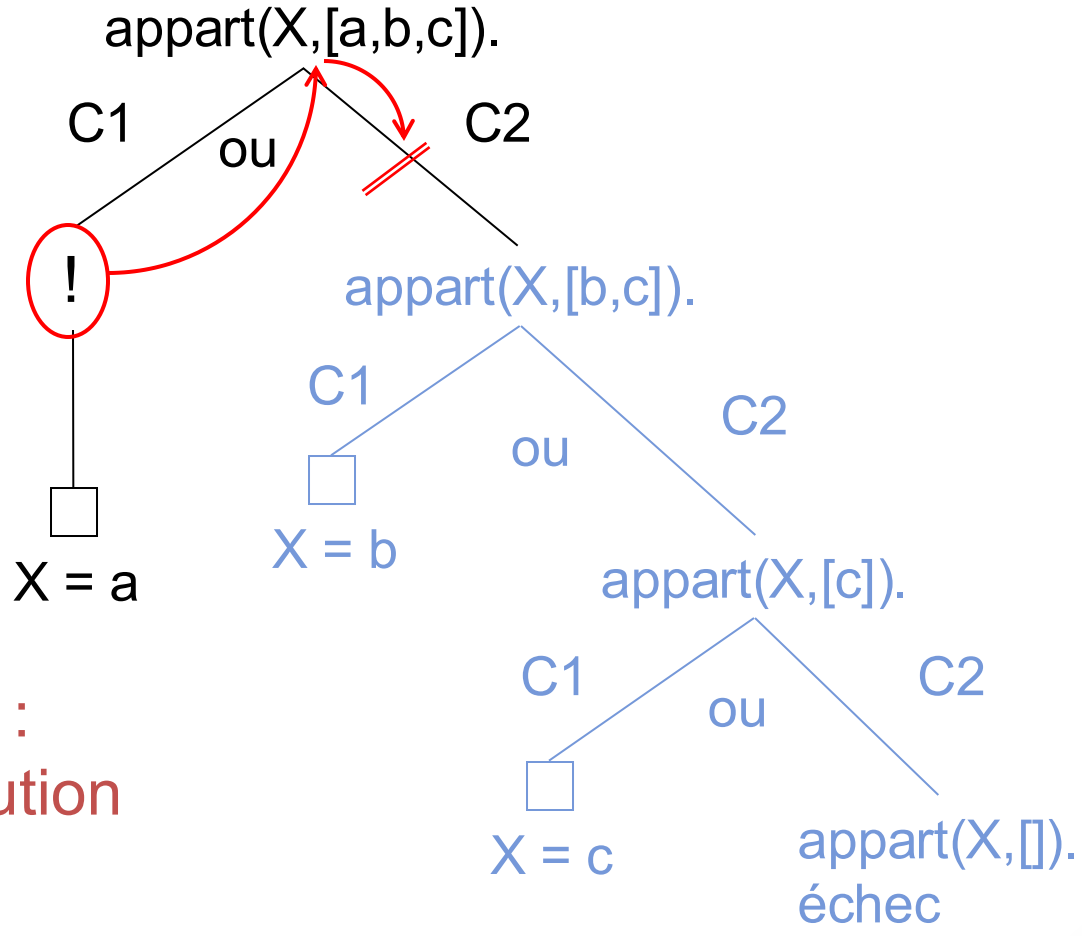
Call: (8) appart(b, [a, b, c]) ? creep
Call: (9) appart(b, [b, c]) ? creep
Exit: (9) appart(b, [b, c]) ? creep
false.

Coupure pour modifier les solutions (1)



Sans coupure :
trois solutions

Coupure pour modifier les solutions (2)



Avec coupure :
une seule solution

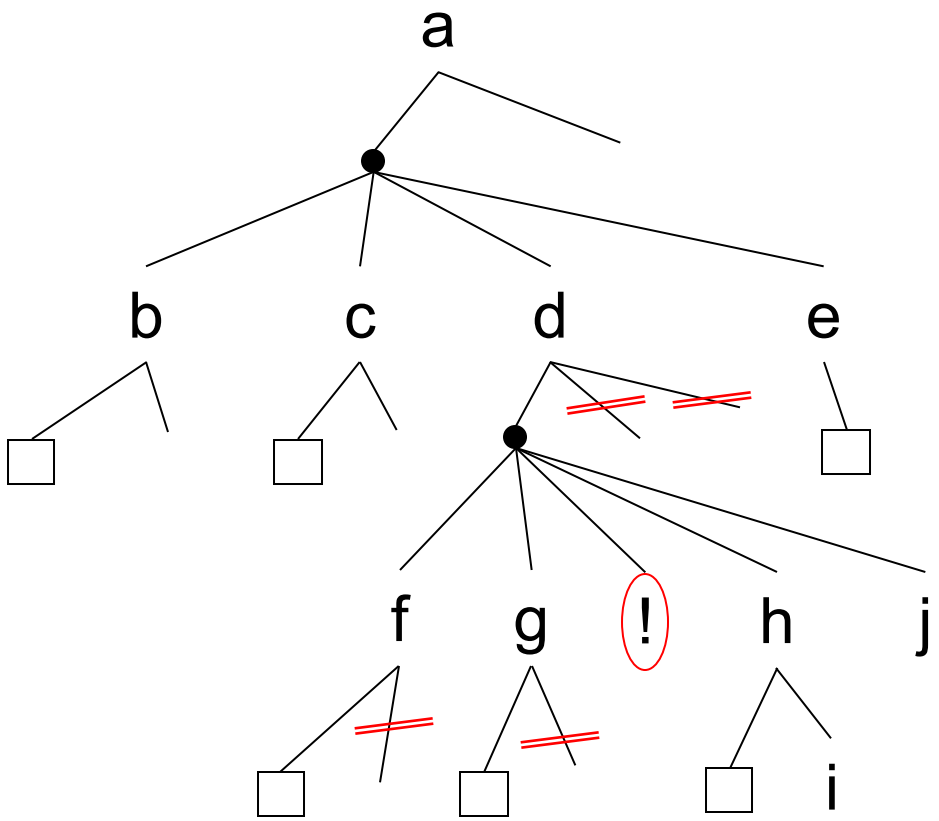
Coupure : différence entre génération et test

- Version sans coupure pour le prédicat en génération (pour avoir toutes les solutions)

- Version avec coupure pour le prédicat en test (ou calcul d'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 :- …)
(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

Coupure pour omettre des conditions

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

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

Coupure pour faire un « si alors sinon »

Pour faire “Si P alors Q sinon R” :

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

La coupure : synthèse

- Objectifs :
  - interdire l'exploration de certaines branches
  - améliorer l'efficacité d'un programme en éliminant des conditions
  - confirmer un choix que l'on sait être le seul ou plus pertinent
  - écrire un « si alors sinon »

- `p :- a, b, ...,!,...,z`
  - Tous les choix mémorisés depuis l'appel de la tête de clause jusqu'à l'exécution du CUT sont supprimés

La négation par l'échec

- Si F est une formule, sa négation est notée not(F)

- Prolog pratique la négation par l'échec
  - Pour démontrer not(F)
    Prolog va tenter de démontrer F
  - Si la démonstration de F échoue,
    Prolog considère que not(F) est démontrée
- not(F) signifie donc
  que la formule F n'est pas démontrable,
  et non que c'est une formule fausse
  - C'est ce que l'on appelle l'hypothèse du monde clos

La négation par l'échec

- Pour faire le prédicat not(F) qui retourne vrai si F échoue :

```
not(F) :- F, !, fail.
not(_).
```

- Pour démontrer not(F), Prolog essaie de démontrer F
- S'il réussit :
  - Le cut élimine les points de choix éventuellement créés durant cette démonstration
  - Le fail fait échouer la règle
  - Le cut ayant éliminé la deuxième règle, c'est la démonstration de not(F) qui échoue
- Si la démonstration de F échoue
  - C'est la deuxième règle qui réussit

Négation par l'échec : danger !

- Soit le programme : `p(a).`

- ```
  ?- X=b, not(p(X)).
  X = b.
  ```

➢ Prolog répond positivement car p(b) n'est pas démontrable.

- ```
  ?- not(p(X)), X=b.
  false.
  ```

➢ Prolog répond négativement car p(X) étant démontrable avec {X = a}, not p(X) est considéré comme faux.

➢ La négation par l'échec ne doit être utilisée que sur des prédicats dont les arguments sont déterminés et à des fins de vérification

➢ Son utilisation ne détermine jamais la valeur d'une variable

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

Chargement de programme

- Chargement d'un programme à partir d'un fichier
  - `?- consult('nom.pl').`
  - `?- consult(nom).`
  - `?- ['nom.pl'].`
  - `?- [nom].`
- Chargement de plusieurs fichiers
  - `?- [nom1, nom2].`
- Ecrire un programme au terminal
  - `?- consult(user).` ou `[user].`

    Puis les clauses

    Puis CTRL D

Chargement de programme

- On ne peut pas charger de clause négative à partir d'un fichier
  - Pour ne pas saisir les demandes au terminal
  - Dans le fichier

    ```
    q(_) :- <la/les questions>.
    ```

  - Puis demander au terminal

    `q(_).` pour une solution

    `q(X)` pour toutes

Affichage de programme

- L'affichage du contenu du fichier chargé
  - `?- listing.`
- L'affichage d'une clause particulière, ici clause1
  - `?- listing(clause1).`

- Edition avec l'éditeur vi
  - `?- edit(nomProgramme).`

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);
false.
```

Ajout de clauses

- `assert(C)`
  - Ajoute la clause C
  - Sa position dans la liste des clauses dépend de l'implémentation
    - `assert(pere(pierre, paul)).`
    - `assert((papy(X,Y) :- pere(X,Z), pere(Z,Y))).`
- `asserta(C)`
  - Ajoute la clause C au début de la liste des clauses
- `assertz(C)`
  - Ajoute la clause C à la fin de la liste des clauses

Retrait de clauses

- Enlever une clause
  - ```
    ?- retract(pere(X,Y)).
    X = pierre
    Y = paul.
    ```
  - ```
    ?- retract(pere(X,Y)).
    X = pierre
    Y = paul ;
    X = paul
    Y = jean ;
    false.
    ```
- Enlever toutes les clauses
  - ```
    ?- retractall(pere(_,_)).
    ```
    ```

Clauses dynamiques

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

:- dynamic toto/1, tata/2.

?- toto(X).
false.
```

Shell, commentaires et sortie

- Pour exécuter des commandes shell
  - `?- sh.`

    <les commandes>

    Pour revenir sous Prolog, CTRL D

- Les commentaires
  - `% sur une ligne`
  - `/* sur plusieurs ligne */`

- Pour sortir
  - `?- halt.`

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

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

De quoi va-t-on parler ?

- Syntaxe et fonctionnement du Prolog
- Les listes
- Les chaînes de caractères
- Les boucles mues par l'échec
- Points de choix et coupure
- Pour travailler en Swi-Prolog
- Questions à se poser pour « coder en prolog »
- Pour aller plus loin

Pour aller plus loin

- Prolog
  - W.F. Clocksin et al. Programmer en Prolog. Editions Eyrolles, 1985.
  - L. Sterling, E. Shapiro. L'art de Prolog. Masson, 1986.
  - M. Van Caneguem. Anatomie de Prolog. InterEditions, 1986.
  - M. Condillac. Prolog : fondements et applications. Dunod, 1986
  - I. Bratko. Programmation en Prolog pour l'Intelligence Artificielle. Intereditions, 1988.
- Swi-Prolog
  - http://www.swi-prolog.org/
- Sources pour construire ce cours
  - Les cours de Nathalie Guin