

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

# **CHAPITRE 4 :**

## **Le langage Prolog**

# Rappels de logique pour Prolog

---

## Définitions

- Terme :
  - Une variable est un terme
  - Une constante est un terme
  - Si  $t_1, t_2, \dots, t_n$  sont des termes, alors  $f(t_1, t_2, \dots, t_n)$  est un terme
  - Exemple : `fil(x)`
- Atome :
  - Si  $t_1, t_2, \dots, t_n$  sont des termes, et  $p$  un prédicat, alors  $p(t_1, t_2, \dots, t_n)$  est un atome
  - Exemple : `pere(x,y)`

# Rappels de logique pour Prolog

---

## Définitions

- **Littéral**
  - Un atome est un littéral (positif)
  - La négation d'un atome est un littéral (négatif)
- Une **clause** est une formule qui a la forme d'une disjonction de littéraux
  - Exemple :  $P(x,y) \vee \neg Q(z)$
- Une **clause concrète** est une clause sans variable
- Une clause de Horn est une clause de la forme :

$$r_1 \wedge r_2 \wedge \dots \wedge r_n \Rightarrow h$$

On peut toujours transformer une formule en un ensemble de clauses

# Rappels de logique pour Prolog

---

## Unification

- Deux termes  $t_1$  et  $t_2$  sont unifiables s'il existe une substitution  $\sigma$  des variables de  $t_1$  et  $t_2$  telle que  $\sigma t_1 = \sigma t_2$
- Exemples
  - $\text{pere}(X, \text{jean})$  s'unifie avec  $\text{pere}(Y, Z)$   
si  $X \mid Y$  et  $\text{jean} \mid Z$
  - $\text{pere}(\text{jean}, \text{mere}(X))$  s'unifie avec  $\text{pere}(Y, \text{mere}(\text{pierre}))$   
si  $\text{jean} \mid Y$  et  $X \mid \text{pierre}$

# Rappels de logique pour Prolog

---

## Les clauses de Horn

- Un littéral est dit positif s'il consiste en une seule lettre (sans négation), il est dit négatif dans le cas contraire.
- On appelle **clause de Horn** toute clause qui possède au plus un littéral positif.
- On appelle **règle**, ou clause de Horn **stricte**, toute clause qui possède exactement un littéral positif.
- On appellera clause de Horn **négative** toute clause sans littéral positif,
- on appellera **fait**, ou clause **unitaire positive** toute clause consistant en un littéral positif.

Exemple :  $\neg p \vee \neg q \vee r$  (règle)

$r$  (fait)

$\neg p \vee \neg q \vee \neg r$  (clause de Horn négative)

# Prolog

---

- Prolog est un langage de programmation à part.
- Le nom Prolog vient de **P**rogrammation **L**ogique.
- Il est utilisé principalement en **Intelligence Artificielle**.
- Ce qui est original, c'est qu'en Prolog, il suffit
  - de **décrire** ce que l'on sait sur le domaine étudié, (en Intelligence Artificielle, on appelle cela une base de connaissances),
  - puis on pose une **question** à propos de ce domaine et Prolog va nous répondre, sans que l'on ait à lui dire comment construire sa réponse !

# Prolog

---

## Prolog utilisé dans ce cours (et en TD)

- **SWI-Prolog**
  - Fonctionne sous Linux, Windows et MacOS.
  - Disponible gratuitement (licence BSD) au département informatique de l'Université de Psychologie d'Amsterdam  
<http://www.swi-prolog.org/download/stable>
- Autres Prolog gratuits
  - Il y a également Visual Prolog Personal Edition, gratuit pour un usage privé
  - GNU-Prolog (par l'INRIA).



# Prolog

---

## LE LANGAGE PROLOG

- Premier interprète PROLOG (A. Colmerauer et P. Roussel), 1972, Université d'Aix-Marseille.
- 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

- Chaîne de caractère démarrant par une **lettre majuscule** ou le **caractère de soulignement** \_
- Peut contenir des lettres, chiffres ou le caractère de soulignement
- Exemples : **X**, **Y**, **\_var**
- \_ est appelée la **variable anonyme** (« indéterminée » )

Exemple : Superman est plus fort que tout le monde se traduit par :  
***plusfort(superman,X).***

# Termes complexes

---

- **Terme simple** : atome, nombre ou variable
- Un **terme complexe** est construit via un **foncteur** suivi d'une séquence d'**arguments**
- **Foncteur** (nom de prédicat) : **atome**
- **Argument** : **terme simple ou complexe**
- Les arguments sont placés après le foncteur, entre parenthèses et séparés par des virgules
- L'**arité** d'un foncteur est son nombre  $n$  d'arguments.  
Indiquée en suffixant par  $/n$
- Exemple : donne/2 diffère de donne/3

# Prolog

---

## TROIS SORTES DE CONNAISSANCES :

### FAITS, REGLES, QUESTIONS

#### 1. Les Faits

- Généralement, on place toutes les déclarations de faits au début du programme même si ce n'est pas obligatoire.
- Un fait se termine toujours par un point « . »
- Faits :  $P(\dots).$  avec  $P$  un prédicat
  - $pere(jean, paul).$
  - $pere(albert, jean).$
  - Clause de Horn réduite à un littéral positif

# Prolog

---

## TROIS SORTES DE CONNAISSANCES : FAITS, REGLES, QUESTIONS

### 2. Règles Prolog

- Enoncent la dépendance d'un prédicat par rapport à d'autres prédicats
- Concernent des catégories d'objets / faits
  - fait : *enColere(fido)*.
  - règle : *Fido aboie si Fido est en colère*
- Le « **si** » s'écrit « **:-** » en Prolog et correspond à l'implication  $\Rightarrow$   
 $\text{enColere(fido)} \Rightarrow \text{aboie(fido)}$ .

s'écrit en Prolog : ***aboie(fido) :- enColere(fido)***.

- Règles : ***P(...)*** :- ***Q(...), ..., R(...)***.
  - *papy(X,Y) :- pere(X,Z), pere(Z,Y)*.
  - Clause de Horn complète

# Prolog

---

## TROIS SORTES DE CONNAISSANCES : FAITS, REGLES, QUESTIONS

### 3. Questions : S(...), ..., T(...).

- pere(jean,X), mere(annie,X).
- Clause de Horn sans littéral positif

### Exercice:

- Exprimer en Prolog les propositions suivantes, identifier les objets, les faits, les règles
  - *la chèvre est un animal herbivore*
  - *le loup est un animal cruel*
  - *toute chose cruelle est carnivore*
  - *un animal carnivore mange de la viande et un animal herbivore mange de l'herbe*
  - *un animal carnivore mange des animaux herbivores*
  - *les carnivores et les herbivores boivent de l'eau*
  - *un animal consomme ce qu'il boit ou ce qu'il mange*
  - Question : y a-t-il un animal cruel et que consomme-t-il ?

# Solution possible

---

animal(chevre).

herbivore(chevre).

animal(loup).

cruel(loup).

carnivore(X) :- cruel(X).

mange(X,viande):- animal(X), carnivore(X).

mange(X,herbe):- animal(X), herbivore(X).

mange(X,Y):- animal(X), animal(Y), carnivore(X),herbivore(Y).

boit(X,eau):- animal(X), (carnivore(X); herbivore(X)).

consomme(X,Y) :- mange(X,Y); boit(X,Y).

?- animal(X), cruel(X), consomme(X,Y).

*la chèvre est un animal herbivore*

*le loup est un animal cruel*

*toute chose cruelle est carnivore*

*un animal carnivore mange de la viande et un animal herbivore mange de l'herbe*

*un animal carnivore mange des animaux herbivores*

*les carnivores et les herbivores boivent de l'eau*

*un animal consomme ce qu'il boit ou ce qu'il mange*

*Question : y a-t-il un animal cruel et que consomme-t-il ?*

# Graphe de résolution

## Programme P

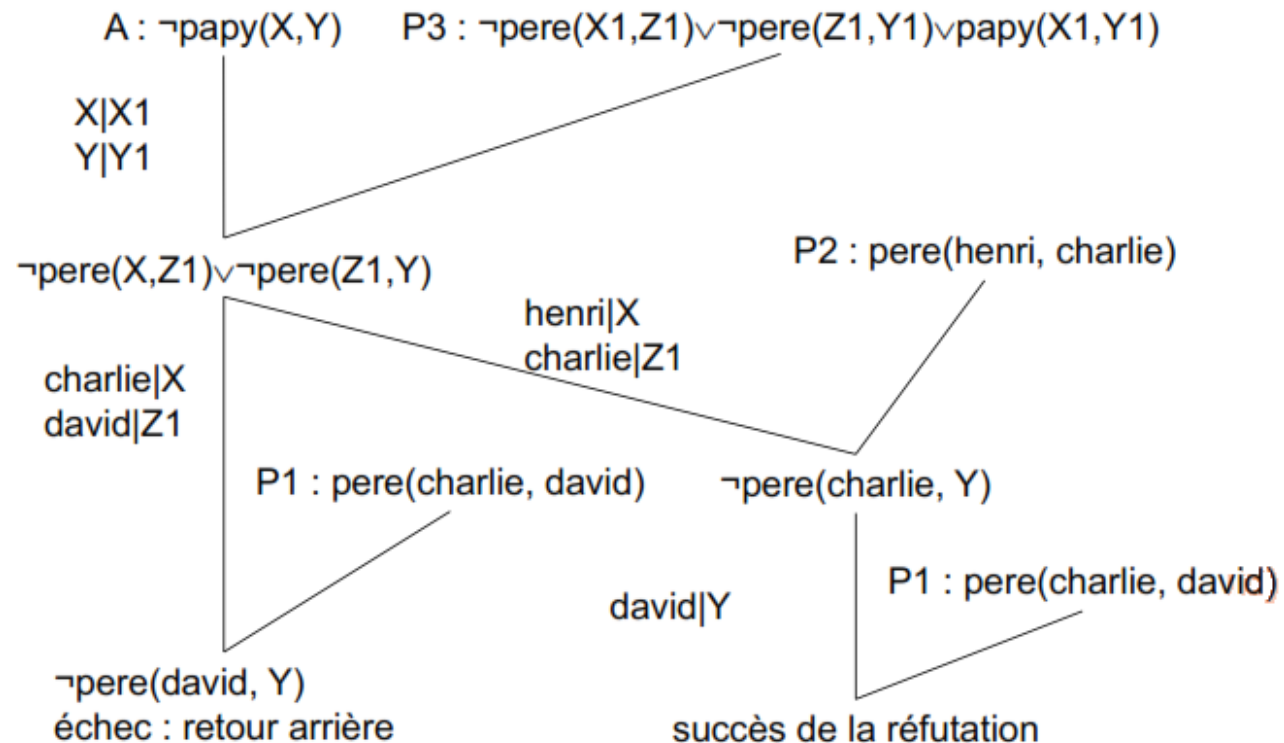
- P1 : `pere(charlie, david).`
- P2 : `pere(henri, charlie).`
- P3 : `papy(X,Y) :- pere(X,Z), pere(Z,Y).`

## Appel du programme P

- A : `papy(X,Y).`

## Réponse : X=henri, Y=david

## Résolution par réfutation





# Graphe de résolution (2)

---

- Pour résoudre une question, Prolog construit l'arbre de recherche de la question :
  - Racine de l'arbre : la question initiale
  - Nœuds : points de choix (formules à démontrer)
  - Passage d'un nœud vers son successeur en effectuant une unification
- Les nœuds sont démontrés de gauche à droite, dans l'ordre de déclaration dans la règle
- Nœuds d'échec : aucune règle ne permet de démontrer la première formule du nœud
- Nœuds de succès : ne contient plus aucune formule à démontrer, tout a été démontré et les éléments de solution sont trouvés en remontant vers la racine de l'arbre

# Graphe de résolution (3)

## INTERPRETATION PROCEDURALE : ARBRE ET-OU

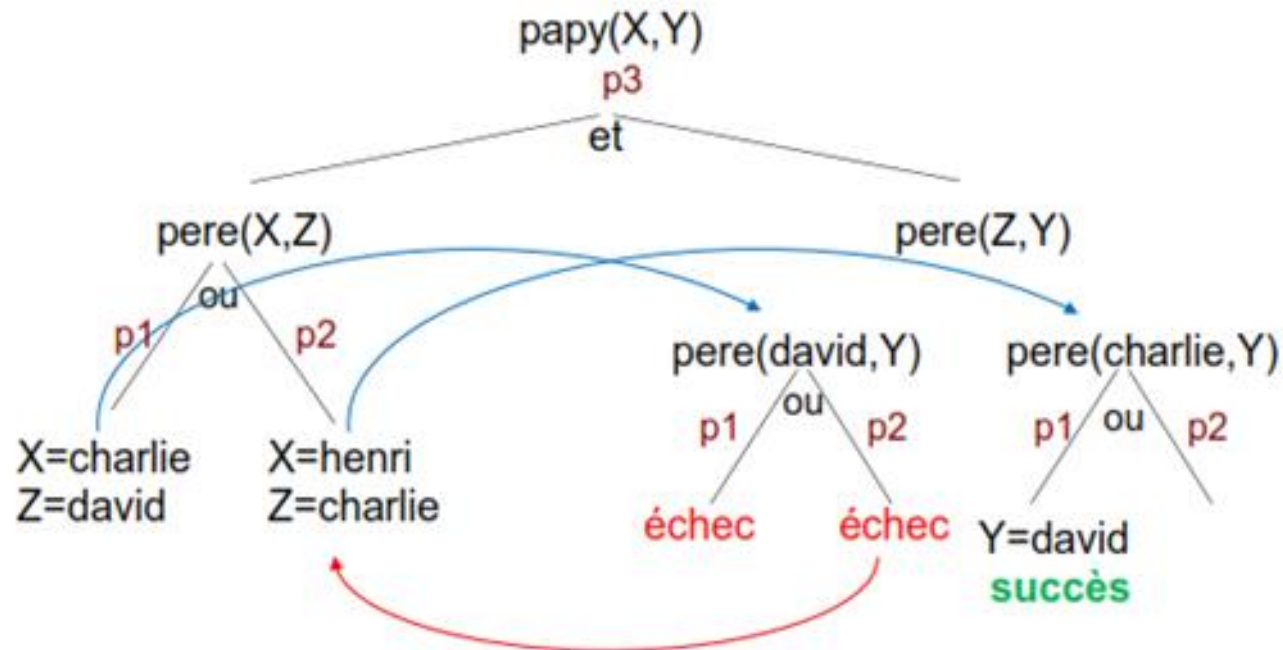
Exemple de programme:

pere(charlie, david). (p1)

pere(henri, charlie). (p2)

papy(X,Y) :- pere(X,Z), pere(Z,Y). (p3)

Appel du programme : papy(X,Y).



# Mon premier programme

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

```
?- 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.  
lirispcl$
```

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

# Arithmétique

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

- Comparaison

Arithmétique	Prolog
$X < Y$	$X < Y.$
$X \leq Y$	$X =\leq Y.$
$X = Y$	$X =:= Y.$
$X \neq Y$	$X =\backslash= Y.$
$X \geq Y$	$X >= Y.$
$X > Y$	$X > Y.$

## Symboles fonctionnels

Dans le fait **age(homme(ahmed), 25)**, le symbole `homme(ahmed)` est un fonctionnel, ce n'est pas un prédicat.

- Commentaire sur plusieurs lignes `/* ...lignes de commentaire.... */`
- Commentaire sur une seule ligne `% ...une ligne de commentaire.`

# 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

# Comparaison et unification de termes

**$:=$   $\backslash=$**

?- A is 3, A:=3.

A = 3.

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

A = 3.

?- a=\b.

ERROR

**$=$   $\backslash==$**

?- 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.

**$=$   $\backslash=$**

?- A=3.

A = 3.

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

false.

# Comparaison et unification de termes

---

?-  $X$  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.

# Comparaison et unification de termes

## Unification dans les Clauses

- L'unification réussie dans la tête se propage aux clauses

*frere(X,Y) :- homme(X), enfant(X,Z), enfant(Y,Z), X\=Y.*

frere(patrick, Qui).

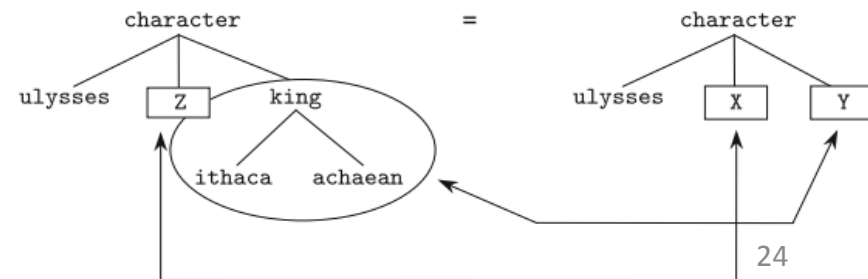
Unification avec frere(X,Y)  
X=patrick et Y=Qui

homme(patrick)  
enfant(patrick,Z)  
enfant(Qui,Z)  
patrick\=Qui

### Exemple 2 (Unification):

?- character(ulysses, Z, king(ithaca, achaeon)) =  
character(ulysses, X, Y).

Z = X, Y = king(ithaca, achaeon).





# 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

# Programmation réursive

## Un exemple : FACTORIELLE

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

```
?- fact(5, R).
```

```
R = 120
```

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

```
Call: (11) fact(3, _11584) ? creep
```

```
Call: (12) _12116 is 3+ -1 ? creep
```

```
Exit: (12) 2 is 3+ -1 ? creep
```

```
Call: (12) fact(2, _12206) ? creep
```

```
Call: (13) _12254 is 2+ -1 ? creep
```

```
Exit: (13) 1 is 2+ -1 ? creep
```

```
Call: (13) fact(1, _12344) ? creep
```

```
Exit: (13) fact(1, 1) ? creep
```

```
Call: (13) _12436 is 1*2 ? creep
```

```
Exit: (13) 2 is 1*2 ? creep
```

```
Exit: (12) fact(2, 2) ? creep
```

```
Call: (12) _11584 is 2*3 ? creep
```

```
Exit: (12) 6 is 2*3 ? creep
```

```
Exit: (11) fact(3, 6) ? creep
```

```
R = 6 .
```

# Bouclage : Cas 1

---

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

```
?- maries(X,Y).
```

```
X = jean,
```

```
Y = sophie ;
```

```
X = philippe,
```

```
Y = stephanie ;
```

```
X = sophie,
```

```
Y = jean ;
```

```
X = stephanie,
```

```
Y = philippe ;
```

```
X = jean,
```

```
Y = sophie ;
```

```
X = philippe,
```

```
Y = stephanie ;
```

```
X = sophie,
```

```
Y = jean ;
```

```
...
```

# Bouclage : Cas 2

---

```
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.
```

# LISTES

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

Liste	tête	queue
[a,b,c]	a	[b,c]
[a]	a	[ ]
[ ]	(fails)	(fails)
[[the,cat],sat]	[the,cat]	[sat]
[the,[cat,sat]]	the	[[cat,sat]]
[X+Y,x+y]	X+Y	[x+y]

## EXEMPLES

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

# LISTES

---

## RECHERCHE D'UN ELEMENT : SANS COUPE-CHOIX (sans le cut !)

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

A la question :   ?- element(X,[c,h,a,t]).

L'interpréteur répond par :

X=c;

X=h;

X=a;

X=t;

Une autre question:

```
?- element(a, [a, b, c, d, a]) .  
true ;  
true ;  
false.
```

# LISTES

---

## RECHERCHE D'UN ELEMENT : AVEC COUPE-CHOIX

Si maintenant, on souhaite avoir seulement un élément de la liste, on introduit une coupure dans l'arbre et la définition devient :

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

Maintenant à la même question, l'interpréteur répond :

```
?- element(X,[c,h,a,t]).
```

```
X = c.
```

# LISTES

---

## RECHERCHE D'UN ELEMENT : AVEC POSITION

(Variable indéterminée)

```
ieme(X,1,[X|_]).  
ieme(X,P,[_|R]) :- P1 is P-1, ieme(X, P1, R).
```

Vérifier si un élément X est à la position P dans la liste L,

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

**false.**

Renvoyer l'élément X qui se trouve à la position P d'une liste L.

?- ieme(X,4,[a,b,c,d]).

X = d ;



# LISTES

## LONGUEUR D'UNE LISTE

```
long([], 0).
```

```
long([_ | L], N1) :- long(L, N), N1 is N + 1.
```

```
?- long([a,b,c,d],N) .  
N = 4.
```

```
?- long([],N) .  
N = 0.
```

## SUPPRESSION D'UN ELEMENT DONNE D'UNE LISTE

```
efface(_, [], []).
```

```
efface(X,[X|L], L).
```

```
efface(X,[Y|L1], [Y|L2]) :- X \== Y, efface(X,L1,L2).
```

```
?- efface(c,[a,b,c,d],L2) .  
L2 = [a, b, d] ;
```

# LISTES

---

## SOMME DES ELEMENTS D'UNE LISTE DE NOMBRES

```
somme([],0).
```

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

```
?- somme([1,2,3,5],N).  
N = 11.
```

---

### 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 .

# LISTES

---

## Utilisation du 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(L1, [a|L2], [b,c,d,a,e,t]).
```

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

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

# 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: (11) appart(_16494, [b, a, c]) ? creep  
Exit: (11) appart(b, [b, a, c]) ? creep  
X = b ;  
Redo: (11) appart(_16494, [b, a, c]) ? creep  
Call: (12) appart(_16494, [a, c]) ? creep  
Exit: (12) appart(a, [a, c]) ? creep  
Exit: (11) appart(a, [b, a, c]) ? creep  
X = a ;  
Redo: (12) appart(_16494, [a, c]) ? creep  
Call: (13) appart(_16494, [c]) ? creep  
Exit: (13) appart(c, [c]) ? creep  
Exit: (12) appart(c, [a, c]) ? creep  
Exit: (11) appart(c, [b, a, c]) ? creep  
X = c ;  
Redo: (13) appart(_16494, [c]) ? creep  
Call: (14) appart(_16494, []) ? creep  
Fail: (14) appart(_16494, []) ? creep  
Fail: (13) appart(_16494, [c]) ? creep  
Fail: (12) appart(_16494, [a, c]) ? creep  
Fail: (11) appart(_16494, [b, a, c]) ? creep  
false.
```

Le prédicat en génération ➡ pour avoir toutes les solutions

Le prédicat en test ➡ calcul d'une seule solution

# Définition d'un prédicat

---

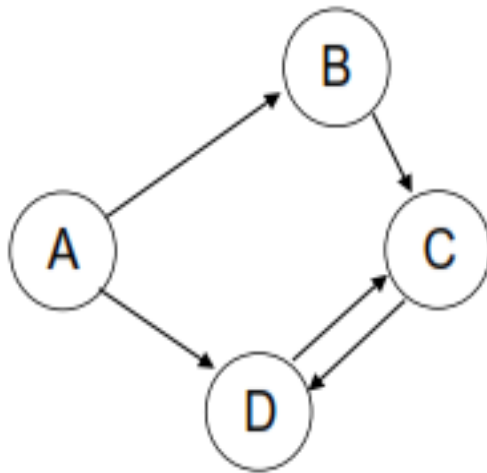
## QUESTIONS A SE POSER

- 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

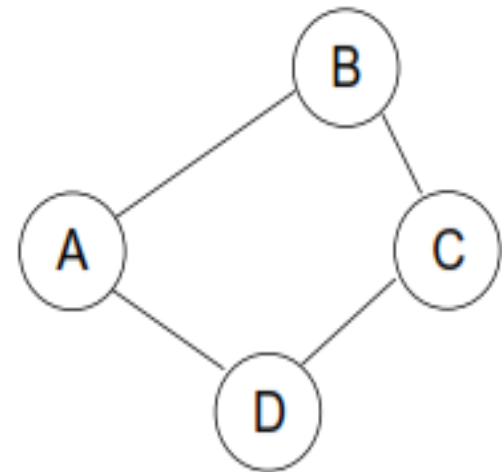
# Les graphes en Prolog

---

- Graphe
  - Ensemble de sommets (ou nœuds) reliés par des arcs(graphe orienté) ou des arêtes (graphe non orienté).



Graphe orienté

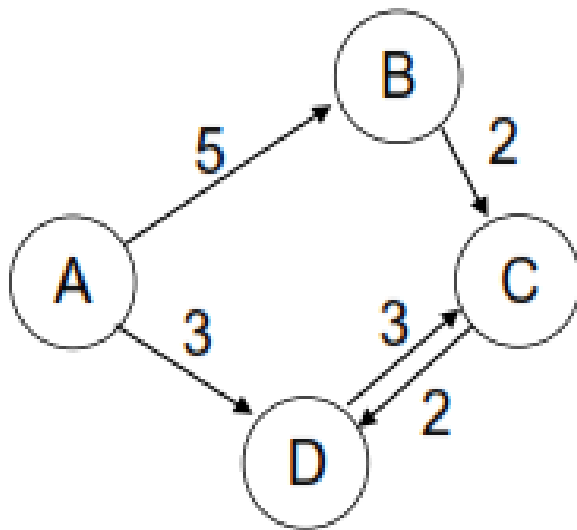


Graphe non orienté

# Les graphes en Prolog

## Représentation des graphes

- Une manière simple de représenter un graphe en Prolog est de décrire
  - Les arcs qui relient les sommets et leurs caractéristiques
  - Les sommets et leur caractéristiques



Graphe orienté

```
arc(a,b,5).  
arc(b,c,2).  
arc(a,d,3).  
arc(c,d,2).  
arc(d,c,3).  
sommet(a).  
sommet(b).  
sommet(c).  
sommet(d).
```

# Les graphes en Prolog

---

## Exemple : des figures à colorier

Problème posé :

- On veut colorier les zones des figures, de sorte que deux zones adjacentes n'aient pas la même couleur.
- On peut représenter ces figures par des graphes :
  - Les sommets représentent les zones de la figure
  - Les arêtes relient deux zones adjacentes

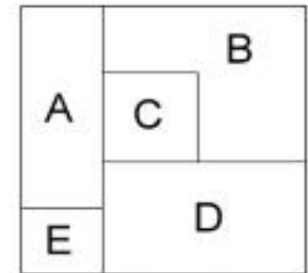


Figure 1

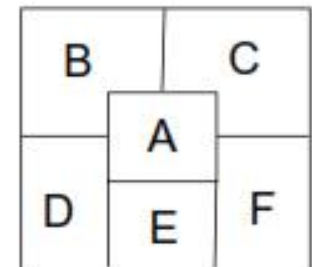
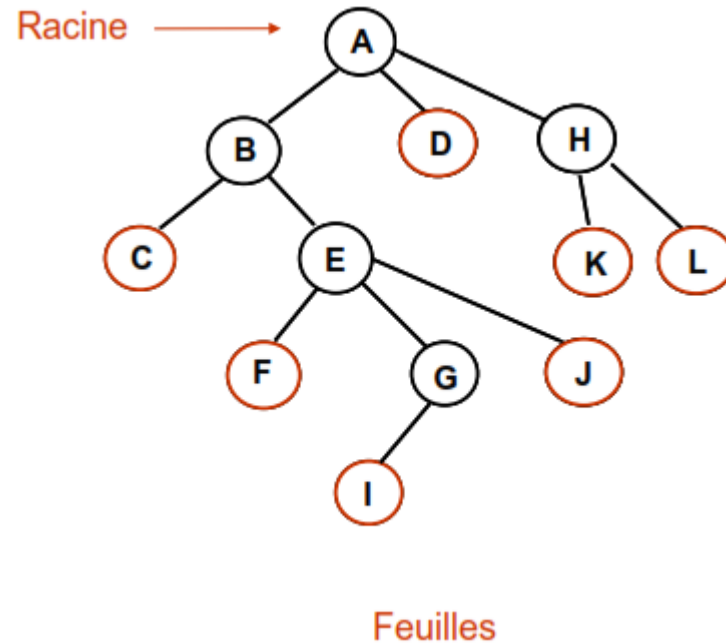
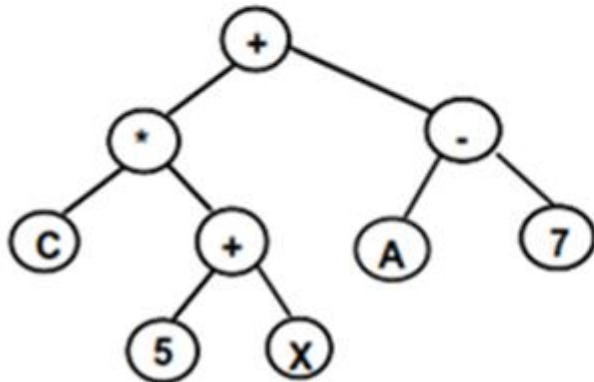
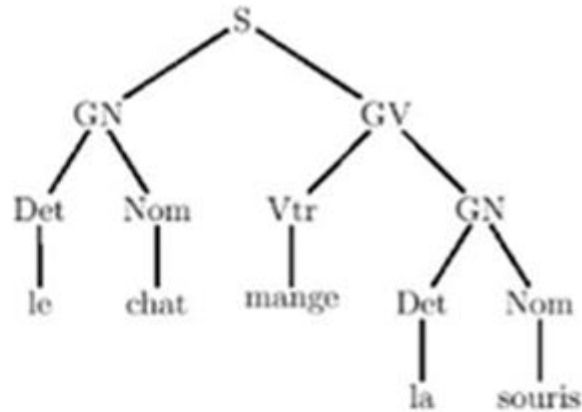
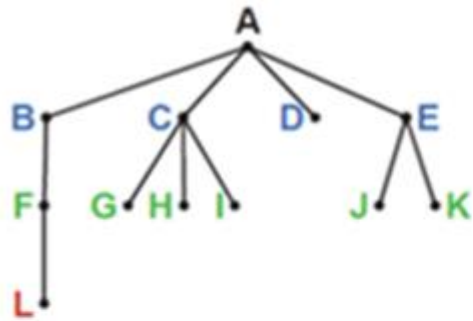


Figure 2



# Les Arbres en Prolog

## Exemples d'arbres

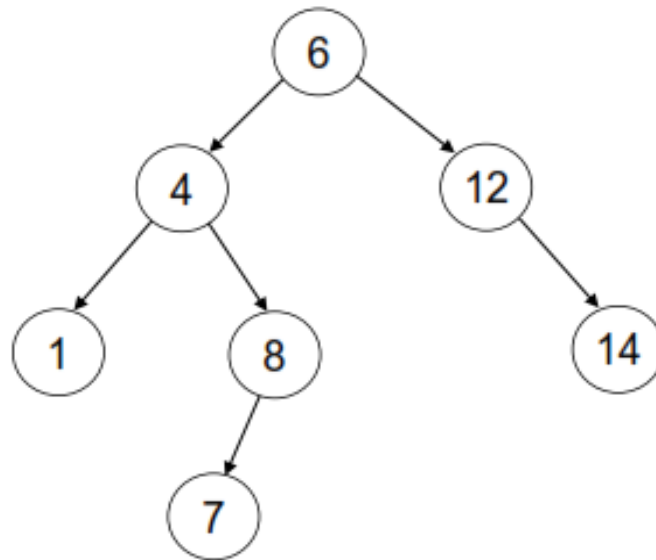


# Les Arbres en Prolog

---

## Représentation des arbres binaires en Prolog

Exemple de représentation d'un arbre binaire en Prolog



Cet arbre binaire ordonné est représenté par la liste suivante :

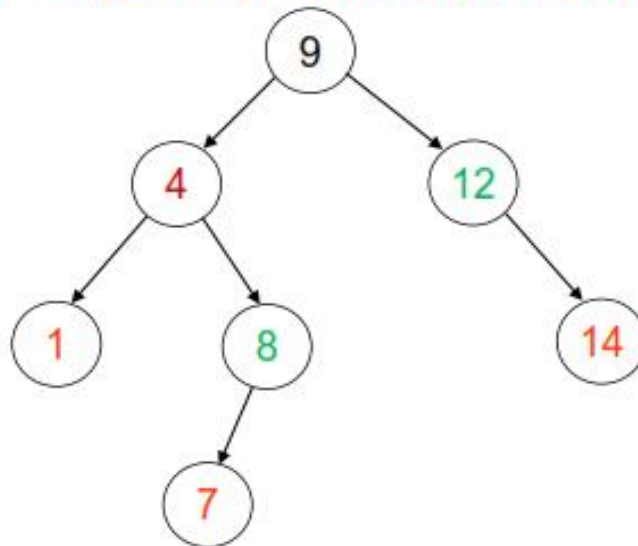
`[6, [4, [1, [], []], [8, [7, [], []], []]], [12, [], [14, [], []]]]`

# Les Arbres en Prolog

---

## Une autre manière de représenter les arbres binaires en Prolog

- Un arbre binaire peut être représenté par le terme de suivant: `t(SousArbreG, Racine, SousArbreD)`.



- `t( t( t(nil, 1, nil), 4, t( t(nil, 7, nil), 8, nil) ), 9, t(nil, 12, t(nil, 14, nil) ) )`
- Arbre vide : `nil`