

Prolog

- ◆ Listes
- ◆ Arbres en prolog
- ◆ Structures : bases de données
- ◆ Opérateurs & prédicats prédéfinies

Inversion récursive d'une liste

- ◆ **Inversion récursive d'une liste :**

```
?- reverseRec([a,[x,[y,z]],[d,e],b],L).
```

```
L = [b, [e,d], [[z,y],x], a]
```

```
reverseRec([],[]).
```

```
reverseRec([X|L1],L2):-  
    atom(X) % not(list(X))  
    reverseRec(L1,L3),  
    append(L3,[X],L2).  
reverseRec([X|L1],L2):-  
    not(atom(X)), % list(X)  
    reverseRec(L1,L3),  
    reverseRec(X,L4),  
    append(L3,[L4],L2).
```

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

Inversion récursive d'une liste : utilisation d'un accumulateur

- ◆ **Inversion récursive d'une liste sans append :**

```
?- reverseRec([a,[x,[y,z]],[d,e],b],L).
```

```
L = [b, [e,d], [[z,y],x], a]
```

```
reverseRec(L,I):-
```

```
    reversRecAcc(L, [], I).
```

```
reverseRecAcc([],I, I).
```

```
reverseRecAcc([X|L1],I1, I2):-
```

```
    atom(X) % not(list(X))
```

```
    reverseRecAcc(L1, [X|I1], I2).
```

```
reverseRecACC([X|L1], I1, I2):-
```

```
    not(atom(X)), % list(X)
```

```
    reverseRec(X, I1, I3),
```

```
    reverseRec(L1, I3, I2).
```

```
list([]).
```

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

Chemin dans un graphe

Soit $G = (S,A)$ un graphe orienté sans boucle,
écrire un prédicat $\text{chemin}(X,Y, L)$, tq L représente
un chemin sans boucle dans G entre X et Y .

$\text{fleche}(1,2)$. $\text{fleche}(1,3)$.

$\text{fleche}(2,4)$. $\text{fleche}(3,2)$.

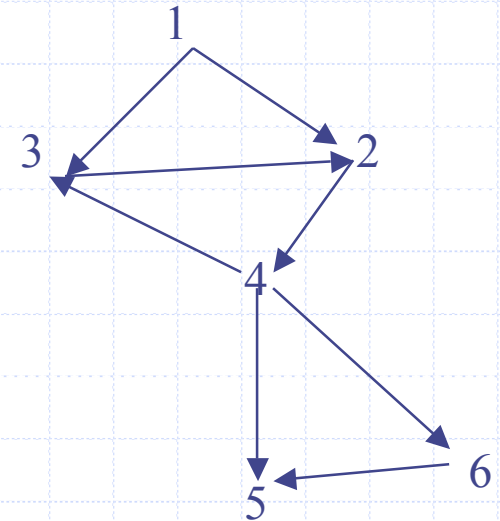
$\text{fleche}(4,3)$. $\text{fleche}(4,5)$.

$\text{fleche}(4,6)$. $\text{fleche}(6,5)$.

?- $\text{chemin}(1,5, L)$.

$L = [1, 2, 4, 5]$

$L = [1, 2, 4, 6, 5]$

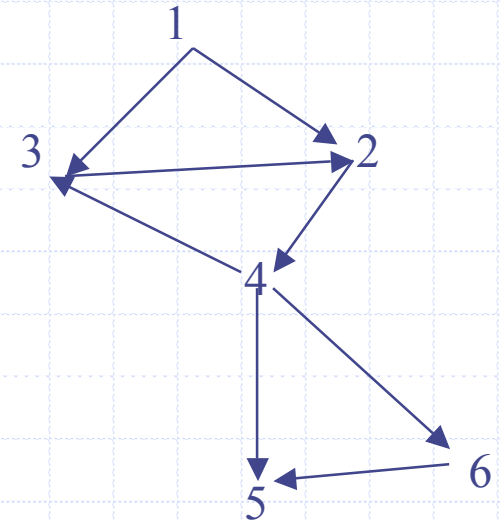


Chemin dans un graphe

```
chemin(X,Y,L) :-  
    chemin_s_boucle(X,Y, [X],L).
```

```
chemin_s_boucle(X,Y,L, [Y|L]) :-  
    fleche(X,Y).
```

```
chemin_s_boucle(X,Y,M,L) :-  
    fleche(X,Z),  
    hors_de(Z,M),  
    chemin_s_boucle(Z,Y, [Z|M],L).
```



Algorithme de tri rapide

- ◆ **Algorithme de tri rapide :**

```
triRapide([], []).
```

```
triRapide([P|L], T) :-
```

```
    partage(P, L, L1, L2),
```

```
    triRapide(L1, T1),
```

```
    triRapide(L2, T2),
```

```
    append(T1, [P|T2], T).
```

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

```
partage(P, [X|T], [X|U1], U2) :-
```

```
    P > X,
```

```
    partage(P, T, U1, U2).
```

```
partage(P, [X|T], U1, [X|U2]) :-
```

```
    P <= X,
```

```
    partage(P, T, U1, U2).
```

Algorithme de tri rapide

- ◆ **Algorithme de tri rapide sans append:**

```
triRapideAcc(L,T):-
```

```
    triRap(L,[],T).
```

```
triRap([],L,L).
```

```
triRap([P|L],Acc,T):-
```

```
    partage(P,L,L1,L2),
```

```
    triRap(L2,Acc,T1),
```

```
    triRap(L1,[P|T1],T).
```

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

```
partage(P,[X|T],[X|U1],U2):-
```

```
    P>X,
```

```
    partage(P,T,U1,U2).
```

```
partage(P,[X|T],U1,[X|U2]):-
```

```
    P<X,
```

```
    partage(P,T,U1,U2).
```

Algorithme de tri fusion

- ◆ **Algorithme de tri rapide sans append:**

```
triFusion([], []).
```

```
triFusion([X], [X]).
```

```
triFusion([A,B|R], S) :-
```

```
    diviser([A,B|R], L1, L2),
```

```
    triFusion(L1, S1),
```

```
    triFusion(L2, S2),
```

```
    fusionner(S1, S2, S).
```

```
diviser([], [], []).
```

```
diviser([A], [A], []).
```

```
diviser([A,B|R], [A|Ra], [B|Rb]) :-
```

```
    diviser(R, Ra, Rb).
```


Algorithme de tri fusion

- ◆ Algorithme de tri rapide sans append:

```
fusionner(A, [], A) .
```

```
fusionner([], B, B) .
```

```
fusionner(A|Ra, [B|Rb], [A|M]) :-
```

```
  A=<B,
```

```
  Fusionner(Ra, [B|Rb], M) .
```

```
fusionner(A|Ra, [B|Rb], [B|M]) :-
```

```
  A>B,
```

```
  fusionner([A|Ra], Rb, M) .
```

Algorithme de tri sélection

```
triSelection([], []).
triSelection([X], [X]).
triSelection([X,Y|L], [Z|T]) :-
    minimum([X,Y|L], Z),
    retirer(Z, [X,Y|L], S),
    triSelection(S, T).
```

```
minimum([X], X).
minimum([X,Y|L], X) :-
    minimum([Y|L], M),
    X <= M.
minimum([X,Y|L], M) :-
    minimum([Y|L], M),
    X > M.
```

```
retirer(X, [], []).
retirer(X, [X|L], L).
retirer(X, [U|L], [U|M]) :-
    X \== U,
    retirer(X, L, M).
```

Algorithme de tri à bulle

```
triBulle(Liste, Trie) :-  
    echange(Liste, List1),!,  
    triBulle(List1, Trie).
```

```
triBulle(Trie,Trie).
```

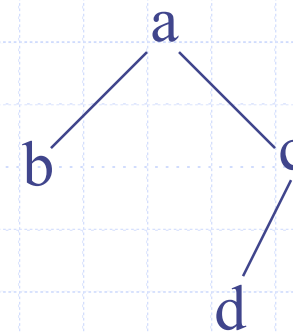
```
echange([X,Y|Reste], [Y,X|Reste]) :-  
    X>Y.
```

```
echange([Z|Reste], [Z|Reste1]) :-  
    echange(Reste,Reste1).
```

Arbres en prolog

- ◆ Arbres binaires :

- représentation ?
 $a(b, c(d))$



- ◆ méthode la plus réponde :

- l'atome nil représente l'arbre vide
- t sera le foncteur tel que l'arbre de racine X ayant un sous-arbre gauche G et un sous-arbre droit D sera représenté par :
 $t(G, X, D)$

- ◆ $a(b, c(d))$ sera représenté par :

$t(t(\text{nil}, b, \text{nil}), a, t(t(\text{nil}, d, \text{nil}), c, \text{nil}))$

Arbres en prolog

- ◆ Arbres binaires :
 - tester l'appartenance d'un élément
 - Afficher un arbre binaire (préfixé, infixé, postfixé)
- ◆ Arbres binaires ordonnés :
 - tester l'appartenance d'un élément
 - ajouter un élément
 - supprimer un élément

Arbre en prolog

- Recherche dans un arbre binaire

```
dans(X,t(_,X,_)):-!.
```

```
dans(X,t(G,_,_)):-
```

```
    dans(X,G).
```

```
dans(X,t(_,_,D)):-
```

```
    dans(X,D).
```

* il est évident que : dans(X,nil) échouera.

- Recherche dans un arbre binaire ordonnée

```
dans(X,t(_,X,_)).
```

```
dans(X,t(G, Racine,_)):-
```

```
    X < Racine, dans(X,G).
```

```
dans(X,t(_,_,D)):-
```

```
    X > Racine, dans(X,D).
```

Arbre en prolog

- **Afficher un arbre binaire :**

```
afficher(nil).
```

```
afficher(t(G,X,D)):-
```

```
    afficher(G),
```

```
    write(X), tab(4),
```

```
    afficher(D).
```

```
* etc.
```

Arbre en prolog

- Ajouter un élément dans un arbre binaire ordonné:

ajout(A, X, A1) : insérer X dans A donne A1

ajout(nil, X, t(nil,X,nil)).

ajout ((t(G, X, D), X, t(G, X, D)).

ajout(t(G, R, D), X, t(Ag, R, D)):-

X<R,

ajout(G, X, Ag).

ajout(t(G, R, D), X, t(G,R,Ad)):-

X>R,

ajout(D, X, Ad).

Arbre en prolog

- Suppression d'un élément dans un arbre binaire ordonné:

suppr(t(nil, X, D), X, D).

suppr(t(G, X, nil), X, G).

suppr(t(G, X, D), X, t(G, Y, D1)) :-

effMin(D, Y, D1).

suppr(t(G, Racine, D), X, t(G1, Racine, D)):-

Racine > X,

suppr(G, X, G1).

suppr(t(G, Racine, D), X, t(G, Racine, D1)):-

Racine < X,

suppr(D, X, D1).

effMin(t(nil, Y, D), Y, D).

effMin(t(G, Racine, D), Y, t(G1, Racine, D)):-

effMin(G, Y, G1).

Utilisations des structures

- ◆ **Extraction d 'informations structurées d 'une base de données**
 - Une base de données en prolog est représenté par un ensemble de fait.
 - **Exemple :**

une famille est composé de trois éléments suivants :

 - ◆ le mari, l 'épouse et les enfants. Les enfants seront représenté par une liste (nombre est variable)

chaque personne est décrite par quatre composants :

 - ◆ le prénom, le nom, la date de naissance, et l 'emploi. Ce dernier peut prendre la valeur « inactif » , « etudiant(e) » ou spécifier l 'employeur et le salaire.
 - Famille(
individu(jean , dupont, date(7, mai, 1950), travail(univ, 1200)) ,
individu(anne, dupont, date(9 mai, 1951), travail(hopital, 1500)) ,
[enfant(rose, dupont, date(5, mai, 1973), etudiante),
enfant(éric, dupont, date(10, octobre, 1978), etudiant)]).

Utilisations des structures

- ◆ Extraction d'informations structurées d'une base de données

?- famille(individu(_ , dupont, _ , _) , _ , _).

Toutes les familles ayant dupont pour nom

?- famille(M, E, [_ , _ , _]).

Les familles ayant trois enfants

?- famille(M, E, [_ , _ , _|_]).

Les familles ayant au moins trois enfants

Quelques prédicats pour faciliter le dialogue avec la BD :

mari(X):-

famille(X,_,_).

epouse(X) :-

famille(_X,_).

Utilisations des structures

- ◆ Extraction d'informations structurées d'une base de données

```
enfant(X):-
```

```
    famille(_,_ , Enfants),
```

```
    member(X, Enfants).
```

```
existe(Individu) :-
```

```
    mari(Individu);
```

```
    epouse(Individu);
```

```
    enfant(Individu).
```

```
dateNaissance(individu(_,_ ,Date,_), Date).
```

```
salaire(individu(_,_ ,_, travail(_ ,S) ), S).
```

```
salaire(individu(_,_ ,_, inactif ), 0).
```

```
?- existe(individu(Prenom, Nom, _ , _)).
```

```
?- enfant(X), dateNaissance(X, date(_,_ ,2000) ).
```

Utilisations des structures

- ◆ Extraction d 'informations structurées d 'une base de données
écrire le prédicat : total(ListeIndividus, SommeSalaire)?

Total([], 0).

Total([Individu|Liste], Somme):-

 salaire(Individu, S),

 total(Liste, Reste),

 Somme is S+Reste.

Poser la question permettant de connaître le salaire total d 'une famille?

?- famille(Mari, Epouse, Enfants), total([Mari, Epouse|Enfants], revenus).

Arithmétique et opérateurs

- ◆ **Opérateurs prédéfinis :**

- `+, -, *, /, mod`

- ◆ Une opération est effectuée uniquement si on la explicitement indiquée.

- ◆ Exemple:

- `x = 1 + 2.`

- `x=1+2`

- ◆ L'opérateur prédéfini `'is'` force l'évaluation.

- `x is 1 + 2.`

- `x=3`

- ◆ Les opérateurs de comparaison force aussi l'évaluation.

- `145 * 34 > 100.`

- `yes`

Arithmétique et opérateurs

◆ Opérateurs de comparaison:

- $X > Y$

- $X < Y$

- $X \geq Y$

- $X \leq Y$

- $X ::= Y$ les valeurs de X et Y sont identiques

- $X \neq Y$ les valeurs de X et Y sont différentes.

Arithmétique et opérateurs

- ◆ **Opérateur (=)**

`X = Y` permet d'unifier X et Y (possibilité d'instanciation de variables).

- ◆ **Exemples:**

- `1 + 2 = 2 + 1.`

- > no

- `1 + 2 ::= 2 + 1.`

- > yes

- `1 + A = B + 2.`

- > A = 2

- > B = 1

Arithmétique et opérateurs

- ◆ **T1 == T2** si les termes T1 et T2 sont littéralement égaux .
i.e. ont la même structure et leur composants sont les mêmes.

- ◆ **T1 \== T2** est vrai si T1 == T2 est faux

- ◆ **Exemples:**

```
?- f(a,b) == f(a,b).
```

```
> yes
```

```
?- f(a,b) == f(a,X).
```

```
> no
```

```
?- f(a,X) == f(a,Y).
```

```
> no
```

```
?- X \== Y
```

```
> yes
```

```
?- t(X, f(a,Y)) == t(X, f(a,Y)).
```

```
> yes
```

Exemple : calcul du PGCD

- ◆ Le PGCD D de X et Y peut être calculé par :
 - (1) Si X et Y sont égaux alors D est égale à X .
 - (2) si $X < Y$ alors D est égale au PGCD de X et $(Y-X)$.
 - (3) si $Y < X$ alors idem que pour (2) en échangeant X et Y

```
pgcd(X, X, X) .
```

```
pgcd(X, Y, D) :-
```

```
    X < Y,
```

```
    Y1 is Y - X,
```

```
    pgcd(X, Y1, D) .
```

```
pgcd(X, Y, D) :-
```

```
    Y < X,
```

```
    pgcd(Y, X, D) .
```

Prédicats prédéfinies

Tester le type d'un terme :

- ◆ Le type d'un terme peut être une variable, une constante (atome, nombre), une structure.
- ◆ Un terme de type variable peut être instancié ou non.

Built-in predicates:

- ◆ **integer(X)** \Leftrightarrow X est un entier
- ◆ **var(X)** \Leftrightarrow X est une variable non instancié
- ◆ **nonvar(X)** \Leftrightarrow X est un terme autre qu'une variable, ou une variable instancié.
- ◆ **atom(X)** \Leftrightarrow X est un atome
- ◆ **atomic(X)** \Leftrightarrow X est entier ou un atome.

Prédicats prédéfinies

◆ Exemples :

```
■ ?- var(Z), Z=2.  
> Z=2  
■ ?- Z=2, var(Z).  
> no  
■ ?- integer(Z), Z=2.  
> no  
■ ?- var(Z), Z=2, integer(Z), nonvar(Z).  
> Z=2  
■ ?- atom(22).  
> no  
■ ?- atomic(22).  
> yes  
■ ?- atom(=>).  
>yes  
■ ?- atom( date(1, mars, 2003) ).  
> no
```

Prédicats prédéfinies

Utilisation: `integer(X), integer(Y), Z is X+Y;`

Que faire en cas d'échecs ...

- ◆ **count(A,L,N) \Leftrightarrow A apparaît N fois dans L**

```
count(_, [], 0).
```

```
count(A, [A|L], N) :- !,
```

```
    count(A, L, N1),
```

```
    N is N1 + 1.
```

```
count(A, [_|L], N) :-
```

```
    count(A, L, N).
```

- ◆ **Mais alors:**

```
?- count(a, [a,b,X,Y], N).
```

```
> N = 3
```

```
?-count(b, [a,b,X,Y], N).
```

```
> N = 3
```

- X et Y ont été instancié à a (b)

Prédicats prédéfinie

- ◆ **Nouvelle solution:**

```
count(_, [], 0).
```

```
count(A, [B|L], N) :-
```

```
    atom(B), A = B, !,
```

```
%B est un atome A?
```

```
    count(A, L, N1),
```

```
%compter nombre de A dans L
```

```
    N is N1 + 1;
```

```
    count(A, L, N).
```

```
%sinon - compter dans L
```