Prolog

- Objets
- Opérateurs
- Listes

Objets en prolog

- **Alphabet** :
  - A,B,..,Z, a,b,...,z
  - 0,1,...,9
  - + - * / < > = : . & _ ~

En prolog tout est un terme

- **variable**: X_25, _resultat
- **constante** :
  - atome (chaîne alphabétique): tom, x_25, ‘racing club de lens’
  - nombre : 3.14, -0.573, 23E-5 (réels) 23, 5753, -42 (entiers)
- **structure** :
  - date(10, mars, 2003),
  - eleve(robert, maitrise, info, adresse(10, ‘ bvd Bazly ’, 62300, ‘ Lens ’))

Arithmétique et opérateurs

- Chaque Prolog dispose d'opérateurs infixés, préfixés et postfixés :

  +, -, *, /

- L'interprète les considère comme foncteurs et transforme les expressions en termes :

  2 * 3 + 4 * 2 est un terme identique à +(*(2,3), *(4,2))

Arithmétique et opérateurs

Règles de précédence et d'associativité

- **La précédence** est la détermination récursive de foncteur principal par une priorité de 0 à 1200
- **L'associativité** détermine le parenthésage de A op B op C :
  - Si elle est à **gauche**, on a **(A op B) op C**
  - si elle est à **droite**, on a **A op (B op C)**

Arithmétique et opérateurs

Règles de précédence et d'associativité

- On peut définir de nouveaux opérateurs par :

  :- op(précédence, associativité, nom)

  - **nom** est un atome.
  - **précédence** est un entier [0..1200](norme Edimbourg)
  - **associativité** : On distingue trois types :

    xfx xfy yfx - opérateurs infixés

    fx fy - opérateurs préfixés

    xf yf - opérateurs postfixés

    f représente l 'opérateur, x & y représentent les arguments

Arithmétique et opérateurs

Règles de précédence et d'associativité

	non assoc.	*droite*	*gauche*
Infixé	xfx	xfy	yfx
préfixé	fx	fy	
postfixé	xf		yf

- **Exemple:**

```
:- op(600, xfx, aime).
On peut écrire : toto aime tata.
Mais pas : toto aime tata aime titi
```

Arithmétique et opérateurs

Règles de précédence et d'associativité

- La précédence d'un atome est 0. La précédence d 'une structure celle de son foncteur principal
- **X** représente un argument dont la précédence doit être strictement inférieur à celle de l'opérateur.
- **Y** représente un argument dont la précédence doit être inférieur ou égale à celle de l'opérateur
- **Exemples:**
  - **a - b - c** est comprise comme **(a - b) - c**

    => l'opérateur '-' doit être défini comme **y f x.**
  - Si l 'opérateur **not** est défini comme **fy** alors l 'expression **not not p** est légale. S'il est défini comme **fx** alors elle est illégale, et elle doit être écrite comme : **not (not p).**

Arithmétique et opérateurs

- **Opérateur prédéfini en Prolog**

```
:- op(1200, xfx, `:-').
:- op(1200, fx, [:-, ?-]).
:- op(1100, xfy, `;').
:- op(1000, xfy, `,').
:- op(700, xfx, [=,is,<,>,=<,>=,==,=\=,\==,=:=]).
:- op(500, yfx, [+,-]).
:- op(500, fx, [+, -, not]).
:- op(400, yfx, [*, /, div]).
:- op(300, xfx, mod).
```

Arithmétique et opérateurs

- **Opérateurs prédéfinis :**
  - +, -, *, /, mod
- Une opération est effectué 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 >= Y
  - X =< Y
  - X =:= Y les valeurs de X et Y sont identiques
  - X =\= 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

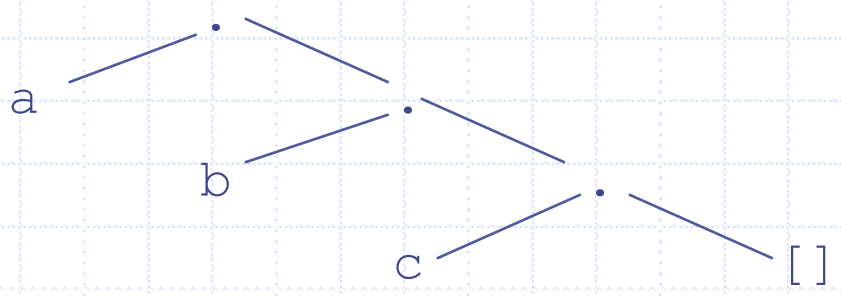
- Le PGCD D de X et Y peut être calculé par :
  - (1) Si X et Y sont égaux alors D est égale à X.
  - (2) si X < Y alors D est égale au PGCD de X et (Y-X).
  - (3) si Y < X alors idem que pour (2) en échangeant X et Y

```
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).
```

Listes

- Une liste est une séquence de nombres ou d 'atomes
- Structure de liste: **.(Car, Cdr)**

  .(a, .(b,[ ])) ⇔



- [paul, dupont] ⇔ .(paul, .(dupont, []))
- [a | Cdr] ⇔ .(a, Cdr)
- [a,b,c] = [a | [b,c]] = [a,b | [c]] = [a,b,c|[]]
- Les éléments peuvent être des listes et des structures:
- [a, [1, 2, 3], tom, 1995, date(10,mars,2003)]

Opérations sur les listes

- **Appartient :**
  - **member(X, L) si X ∈ L.**

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

- **Concaténation :**
  - **conc(L1, L2, L3) si L3 = L1.L2**

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
            conc(L1, L2, L3).
```

Opérations sur les listes

```
?- conc( [a,b,c], [1,2,3], L).
   > L = [a,b,c,1,2,3]
?- conc( L1, L2, [a,b,c] ).
   > L1 = [] , L2 = [a,b,c];
   > L1 = [a], L2 = [b,c];
   > L1 = [a,b], L2 = [c];
   > L1 = [a,b,c], L2 = [];
   > no
?- conc( Avant, [4|Apres], [1,2,3,4,5,6,7]).
   > Avant = [1,2,3], Apres = [5,6,7]
?- conc(_, [Pred, 4, Succ |_], [1,2,3,4,5,6,7]).
   > Pred = 3, Succ = 5
```

Opérations sur les listes

- **Redéfinition de member en utilisant conc:**

```
member1(X, L) :-
conc(_, [X|_], L).
```

- **ajout en tête :**

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

- **Suppression de toutes les occurrences d 'un élément**

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

- **Suppression d'un élément**

```
del1(X, [X|L], L).
del1(X, [Y|L], [Y|L1]) :-
    del1(X, L, L1).
```

- Si X apparaît plusieurs fois, toutes les occurrences de X sont supprimés.

Opérations sur les listes

- **Pour insérer un élément dans une liste:**

```
?- del1(a, L, [1,2,3]).
> L = [a,1,2,3]; > L = [1,a,2,3]; > L = [1,2,a,3];
> L = [1,2,3,a]; > no
```

- **On peut définir insert en utilisant del1:**

```
insert(X,L1,L2) :-
   del1(X, L2, L1).
```

- **La relation sous-liste**

```
sublist(S, L) :-
   conc(L1, L2, L),
   conc(S, L3, L2).

?- sublist(S, [a,b,c]).
> S = []; S = [a]; ... S = [b,c];> S = [a, b,c];
> no
```

Opérations sur les listes

- **Permutations:**

```
permutation([], []).
permutation([X|L], P) :-
    permutation(L, L1),
    insert(X, L1, P).

insert(X, L, [X|L]).
insert(X, [Y|L1], [Y|L2]):-
   insert(X,L1, L2).
?- permutation( [a,b,c], P).
> P = [a,b,c];
> P = [a,c,b];
> P = [b,a,c];
...
permutation2([], []).
permutation2(L, [X|P]) :-
   del(X, L, L1),
   permutation2(L1, P).
```

Opérations sur les listes

- La longueur d ’une liste peut être calculé da la manière suivante :

```
length([], 0).
length([_|L],N) :-
   length(L, N1),
   N is 1 + N1.
?- length([a,b,[c,d],e], N).
   > N = 4
?- length(L,4).
   > [_5, _10, _15, _20] ;
   ..... ?
```

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)** <=> X est un entier
- **var(X)** <=> X est une variable non instancié
- **nonvar(X)** <=> X est un terme autre qu'une variable, ou une variable instancié.
- **atom(X)** <=> X est un atome
- **atomic(X)** <=> 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) <=> 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
```

Prédicats de manipulation de la BD

- **Ajout et suppression de clauses (règles) en cours d'exécution :**
- **assert(C) :**

  ajout de la clause C à la base de données.
- **retract(C) :**

  Suppression des clauses unifiable avec C.
- **asserta(C) :**

  ajout de C au début de la base de données.
- **assertz(C) :**

  ajout de C en fin de la base de données.

Prédicats repeat

- **Repeat**

  Le prédicat repeat est toujours vrai (succès) et à chaque fois qu 'il est rencontré, une nouvelle branche d 'exécution est générée.

- Repeat peut être définie par :

```
repeat.
repeat :- repeat.
```

- **Exemple d'utilisation:**

```
dosquares :-
    repeat,
    read(X),
    (X = stop, !; Y is X*X, write(Y), fail ).
```

Prédicats bagof & setof

- **bagof and setof**
  - La résolution Prolog peut trouver toutes les solutions satisfaisant un ensemble de buts.
  - Mais lorsqu 'une nouvelle solution est générée, la solution précédente est perdue.
  - Les prédicats bagof,setof et findall permettent de collecter ses solutions dans une liste.
- **bagof(X,P,L):**
  - permet de produire la liste L de tout les objets X vérifiant P.
  - Utile si X et P admettent des variable en commun.
- **setof(X,P,L):**
  - idem que bagof. Les éléments de la liste sont ordonnées et sans répétitions.

Prédicats bagof & setof

Exemples:

```
class( f, con).
class( e, vow).
class( d, con).
class( c, con).
class( b, con).
class( a, vow).
?- bagof(Letter, class(Letter, con), List).
    > List = [f,d,c,b]
?- bagof(Letter, class(Letter,Class), List).
    > Class = con
      List = [f,d,c,b];
    > Class = vow
      List = [e,a]
```

Prédicats bagof & setof

```
?- setof(Letter, class(Letter,con), List).
    > Letter = _0
      List = [b,c,d,f]
?- setof((C1,Let), class(Let,C1), List).
    > C1 = _0
      Let = _1
      List = [(con,b),(con,c),(con,d),(con,f),(vow,a),
      (vow,e)]
```