NFP120 --- an 8

Spécification Logique
et
Validation des Programmes Séquentiels

COURS n° 8

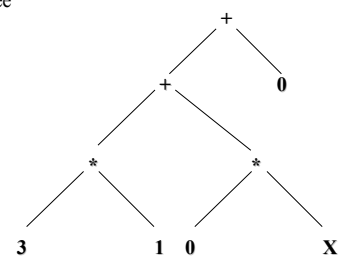
PROLOG

opérateurs

Structure de donnée

3 * 1 + 0 * x + 0

Arbre de représentation



Structures de données : termes fonctionnels T

DATALOG utilise des constantes simples.
Prolog va nous permettre d'utiliser des données à la structure complexe : les termes fonctionnels

```
%exemple :  DATALOG
cours(informatique, mardi,18,20, dewez,jl, amphi, v).

%peut devenir par exemple : Prolog
cours(informatique,
        horaire(mardi,18,20), % Terme fonctionnel
        enseignant(dewez,jl), % Terme fonctionnel
      lieu(amphi, v) % Terme fonctionnel
).
...
```


Structures de données$_2$

```
%et alors
enseigneCours(Cours, Enseignant) :-
                lieu(Cours, _, Enseignant, _).
durée(Cours, Durée) : -
          cours (Cours, horaire(Jour, D, F), _, _),
          Durée is F - D.

%Alors :
? - durée(informatique, D)
            D = 2
```

ATTENTION : terme fonctionnel ≠ prédicat

Exemple : manipulation formelle
Dérivation symbolique des polynomes

```
derivée(N, 0) :- number(N).
derivée(x, 1).
derivée(F + G, DF  + DG) :-
        derivée(F,   DF),
        derivée(G,  DG).
derivée(F * G,   F *  DG + DF * G) :-
        derivée(F, DF),
        derivée(G, DG).
derivée(F / G,  ( DF *  G - F * DG)/(G*G)) :-
        derivée(F, DF),
        derivée(G, DG).
```


Syntaxe de Prolog$_1$

- Constantes {a, b, c, ...}

- Symboles fonctionnels {f/2, g/1, ...}

- Variables {X, X1, ... , Y, ...}

- Termes fonctionnels (définis inductivement).
  - (i) - une variable est un terme
  - (ii) - une constante est un terme
  - (iv) - si f est un symbole fonctionnel n-aire et si t_1,..., t_n sont des termes alors f (t_1,..., t_n) est un terme.
  - (v) - il n'y a pas d'autres termes

Syntaxe de Prolog (suite)

- Symboles de prédicats (associés à leurs arités)
  { p/2, q/3, r/0, ...}

- Atome :
  si p est un symbole de prédicat n-aire
  si $t_1,..., t_n$ sont des termes
  alors p ($t_1,..., t_n$) est un atome.

- Littéral : atome ou négation d'un atome.

- Clause : fait ou règle.

Univers Herbrand

exemple de programme Prolog : les entiers de Peano

```
nat(0).
nat(s(X)) :- nat(X).
```

Univers de Herbrand (Up) :
Ensemble des termes clos formés à partir
des symboles de constantes et de fonctions du programme
Up = {o, s(o), s(s(o)),...}

ATTENTION : Up est infini dès qu'il y a un symbole de fonction
*Remarque : si aucun symbole de constante, on en choisit un arbitrairement (**a** par exemple)*

BASE de Herbrand : Bp

La Base de Herbrand Bp est Ensemble des atomes clos par des éléments de Up
Bp = {nat(o), nat(s(o)), nat(s(s(o))),...}

Bp est infinie dès qu'il y a un symbole de fonction.

Interprétation de Herbrand : Sous-ensemble de Bp.

Rappel : Un atome est satisfait par une interprétation I s'il figure dans I.

Modèle de Herbrand : Interprétation qui satisfait le programme.

Unification :
Ensemble de discordance

Ensemble de discordance d'un ensemble E = $\{e_i\}$ d'expressions (atomes ou termes)

Calcul :
1) parcourir chaque élément de E de gauche à droite jusqu'aux premiers des caractères qui diffèrent.

2) extraire de chaque élément de E la sous-expression qui débute à cette position.

Ensemble de discordance (Unification)

%exemples :
E_1 = *{p(X, f(a,X), g(X, b), p(X, f(a,g(Z))}*
D_1 = *{f(a,X), b, f(a, g(Z))} %NON unifiable*

E_2 = *{p(a,X, f(g(Y))), p(Z, f(Z), f(V))}*
D_1 = *{ a, Z} % unifiable ; {Z --->a}*
E_2 . *{Z --->a}* = *{p(a,X, f(g(Y))), p(a, f(a), f(V))}*
D_2 = *{ X, f (a)} % unifiable {X ---> f(a)}*
E_2 . *{Z --->a , X ---> f(a)}* =
{p(a,f(a), f(g(Y))), p(a, f(a), f(V))}
D_3 = *{g(Y) , V} % unifiable { V ---> g(Y)}*
E_2 . *{Z --->a , X ---> f(a) , V ---> g(Y)}* =
{p(a,f(a), f(g(Y)))}

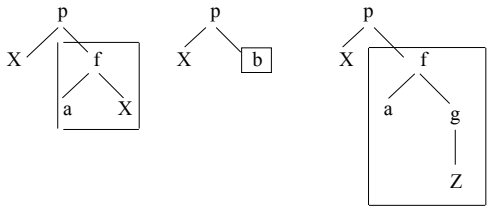
ALGORITHME D'UNIFICATION

E un ensemble de termes de même arité
σ = mgu (E) ou Echec
1) σ = {}
2) si E.σ est un singleton alors
   arrêt et retourner σ
3) sinon Former D l'ensemble de discordance de E.σ
4) Si il existe 2 éléments V et t de D
   avec V une variable
   et t un terme
   alors σ = σ ° {V ---> t}
   reprendre en (2)

5) sinon ECHEC : E n'est pas unifiable

Exemple d'unification

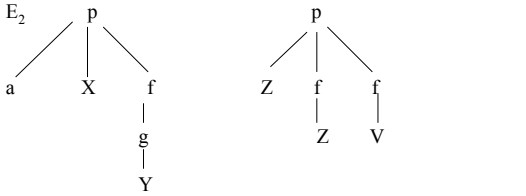
E_1



Echec de l'unification

Exemple d'Unification 2

E_2



D_1 = {a, Z} σ_1 = {Z }
D_2 = {X, f (a)} σ_2 = {Z --> a, X --> f (a)}
D_3 = {g (Y), V} σ_3 = {Z --> a, X --> f (a), V --> g(Y)}

E_4 = {p (a, f (a), f (g (Y))} : Succès de l'unification

Occur check/test d'occurrence

ATTENTION : f (X, X) = f (Y, g (Y))
conduit à

D_1 = {X, Y} σ_1 = {X ---> Y}

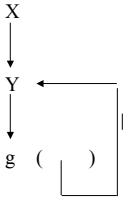
D_2 = {Y, g(Y)} $\sigma_2 = \{Y \text{ ---> } g(Y)\}$!

{X ---> g(Y) , Y ---> g(Y)} mgu ??

D'où les termes :
`f(g(Y), g(Y))` et `f(g(Y), g(g(Y)))`
etc ...

Test d'occurrence suite

D_3 = {Y, g (Y)} l'algorithme génère donc un terme infini



Autre exemple : ?- X=f(X) , write(X).

Programmation Prolog

programmation récursive

l'arithmétique de Prolog (is et les opérateurs)

les listes

contrôler la résolution : le CUT (!)

Premiers Programmes (i)

```
/*si, les entiers naturels de Peano {0 , s/1} sont représentés par
des termes du genre  s(s(s(s(s(s(0))))))
On vérifie que un terme est un entier de Peano par nat/1 :
*/
nat(0).
nat(s(A)) :-  nat(A).

?- nat(s(s(s(s(s(s(0)))))))).
true.
?- nat(X).
X = 0 ;
X = s(0) ;
X = s(s(0))
true.
```

Premiers Programmes (ii)

```
/* l'addition de 2 entiers de peano : add/3
*/
add(0,X,X) :- nat(X). % 0 + x = x
add(s(X),Y,s(Z)):- add(X,Y,Z). % x+1  + y = (x+y) + 1

?- add(s(s(s(s(s(s(0))))))) ,  s(s(s(0))) , M).
M = s(s(s(s(s(s(s(s(s(0)))))))))) ;
fail.
```

Mais aussi : ?- add(A , B , s(s(s(0)))).

A = 0 B = s(s(s(0))) ;
A = s(0) B = s(s(0)) ;
A = s(s(0)) B = s(0) ;
A = s(s(s(0))) B = 0 ;
fail.

Test d'occurrence suite2

?- trace , add(s(s(s(0))) , C , C).
Call: (8) add(s(s(s(0))), _G292, _G292) ? creep
Call: (9) add(s(s(0)), s(_G397), _G397) ? creep
Call: (10) add(s(0), s(s(_G399)), _G399) ? creep
Call: (11) add(0, s(s(s(_G401))), _G401) ?

Avec add(0, N1 , N1)

Arithmétique et Prolog

On dispose de Constantes numériques : Entiers et Flottants (SWI)
Contrôlés par number/1 , integer/1 , float/1

On dispose d'opérateurs de comparaison entre nombres :
< , <= , == , =\= , => , > , ...
On dispose des opérateurs arithmétiques (infixés) :
+ , - , * , /

MAIS Rappel :

?- X = 2+3*5 , display(X).
+(2, *(3, 5)) arbre parce que Unification
X = 2+3*5
true.

Le prédicat d'affection : is/2

?- X is 2+3*5.
17 ouf !!!

En général : -Number is +Expr
ATTENTION : à droite de 'is' l'expression doit être
COMPLETEMENT instanciée au moment de l'appel
sinon 'Exception'.
On parle de prédicats exécutables
à gauche on trouve soit une variable qui sera « affectée »
soit une constante dont la valeur sera comparée à la valeur
de l'expression de droite

Opérateurs Prolog

Déclaration d'opérateur : op(1000,xfy, (,))

1000 : precedence (de 0 à 1200) :
niveau 1200 : ponctuation générale des clauses
op(1200,fx, (:-)), op(1200,fx, (?-)), op(1200,xfx, (:-))
niveau 1100 : op(1100,xfy, (;)) le 'ou'
niveau 1000 : op(1000,xfy, (,)) le 'et '

xfy : associativité : xfx : non-associatif , xfy associatif à droite ,
yfx associatif à gauche
fx non répétable , fy répétable,
xf non répétable , yf répétable.

(,) : nom de l 'opérateur

?- setof(op(X,Y,Z) , current_op(X,Y,Z) , L).

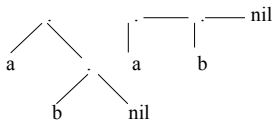
op(400, yfx, *), op(400, yfx, /), op(400, yfx, //),
…
op(500, yfx, +), op(500, yfx, -), op(500, yfx, /\), op(500, yfx, \/),
…
op(700, xfx, =), op(700, xfx, is), op(700, xfx, =..),
…
op(900, fy, \+), op(900, fy, not),
…
op(1000, xfy, (,)),
…
op(1100, xfy, (;)), op(1100, xfy, ('|')),

op(1200, fx, (:-)), op(1200, fx, (?-)), op(1200, xfx, (:-))

Les Listes

Structure de données privilégiée (poubelle) par les interpréteurs :

'.' (a, '.' (b, nil))



Notation pointée infixée : a . b . nil

le '.' / 2 sépare l'élément en tête de liste de la liste à droite.

Notation des listes dans Prolog

a . b . c . nil est notée plus simplement [a, b, c]
nil est notée [ ]
a . Queue devient [a | Queue]

```
? -  [T | Q]  =  [a, b, c,]
       T = a  ,  Q =    [b, c]
       true.
? -  [T | Q]  =  [a]
       T = a  ,  Q = [ ]
true.
```


Notation des listes (suite)

en fait [a] = [a | [ ]]

et plus généralement [a, b, c] peut se noter indifféremment
[a, b, c | [ ]] ou [a, b | [c]] ou [a, b | [c | [ ]]] ou ...

```
%mais bien sur :
? - [T | Q] = [ ].
       No
%et aussi :
? - [[]] = [ ].
       No
```


Manipulation des listes (i)

```
/*La longueur d'une liste « avec entiers naturels de
Peano »*/
%longueur / 2 :
longueur([], 0).
longueur( [T| Q], s(N)) :- longueur( Q, N).

?- longueur([a,b,c,f] , L).
L = s(s(s(s(0)))) ;
fail.
?- longueur( L , s(s(s(s(0))))).
L = [_G374, _G377, _G380, _G383] ;
fail.
```

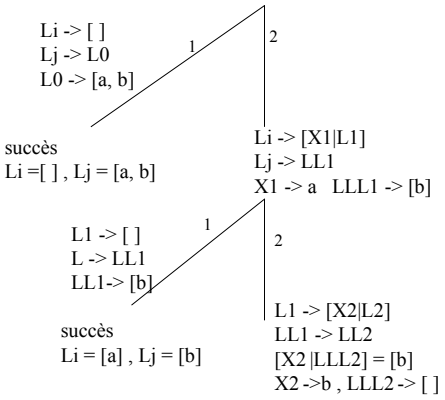

longueur/2 *(revisité)*

```
%longueur / 2 : « du temps de Peano »
        longueur([], 0).
        longueur( [T| Q], s(N)) : - longueur( Q, N).
%du temps de l'arithmétique : %lg / 2
        lg([], 0).
        lg( [T| Q], N) : - lg( Q, NN), N is NN+1.
%attention à la place du but N is NN+1
?- lg([a,b,c] , N).
N = 3 ;
fail.
?- lg(L , 3).
L = [_G236, _G239, _G242]
%ATTENTION « pas de ; »
```

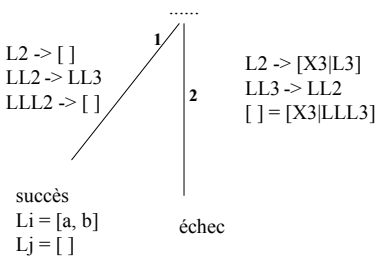

Concaténation de 2 listes

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

    ? - conc( [a, b], [c], L).
          L = [a, b, c]

% mais aussi
    ? - conc( Li, Lj, [a, b] )
          Li = [ ] ,  Lj=[a, b] ;
          Li = [a] ,  Lj=[b] ;
          Li = [a,b ] ,  Lj=[] ;
          fail.
% ...
```

Exemple : ?- conc (Li, Lj, [a, b])



?- conc (Li, Lj , [a,b]) *suite*



Sémantique de conc

Up = {**a**, [ ], [**a**], [[ ]], [[**a**], []], ...} // **a** constante arbitraire

Bp = {conc([ ], [ ], [ ]),
conc([**a**], [ ], [**a**]),
conc([**a**], [**a**], [**a**, **a**]),...}

mais aussi

conc([ ], a,a) !

En effet conc([ ], **a**, **a**) est dans la dénotation de conc :

? - conc ([ ], **a**, **a**)
yes

sémantique de conc (*suite)*

donc attention : Problème de typage ! Première solution

```
conc1( [ ], L, L) :- liste(L).
conc1( [X|L1], L2, [X|L3]) :- conc1(L1, L2, L3).

liste( [ ] ).
liste( [_ | L] ) :- liste(L)
```

Problème de typage , autre solution :

```
conc2( [ ], [ ], [ ] )
conc2( [ ], [ T | Q], [ T | Q] )
conc2( [X | L1], L2, [X | L3] ) :- conc2(L1, L2, L3).
```

Mais append/3 des Prologs est défini comme conc/3

Spécification --- ordre des clauses $_1$

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

?- membre(r , [a,z,e,r,t,y,u,i,o,p]).
Yes
?- membre(X , [a,z,e,r,t,y,u,i,o,p]).
X = a ;
X = z ;
X = e ;
…

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


Spécification --- ordre des clauses $_2$

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

?- membr(r , [a,z,e,r,t,y,u,i,o,p]).
Yes

?- membr(X , [a,z,e,r,t,y,u,i,o,p]).
X = p ;
X = o ;
X = i ;
…
```

Même comportement mais ordres différents des réponses…

Prolog et Spécification : Tri (lent)

```
tri(T, Trie) : - permutation(T, Trie),  ordonnée(Trie).

permutation ( [ ], [ ] )
permutation ( [T1 | RT], TS) :-
        permutation ( RT, RTp),
         append(RTp1, RTp2, RTp), append(RTp1, [T1 | RTp2], TS).

ordonnée( [ ] )
ordonnée( [ X ] )
ordonnée( [ X, Y | Reste ] ) :-
          compare(< , X , Y), ordonnée( [Y | Reste] ).
ordonnée( [ X, Y | Reste ] ) :-
          compare(= , X , Y), ordonnée( [Y | Reste] ).
```


compare/3 : comparer deux termes

?- help(compare). %prédéfini de SWI

compare(?Order, +Term1, +Term2)

Determine or test the Order between two terms in the standard order of terms.

Order is one of <, > or =, with the obvious meaning.

?-

Prolog & Algorithme : Tri (Rapide)

```
quicksort([ ], [ ]).
quicksort([X|Xs], Ys) :- partition(Xs, X, INF, SUP),
        quicksort(INF , Ls), quicksort(SUP , Bs),
        append(Ls, [X|Bs], Ys).

partition([X|Xs], Y, [X|L], Bs) :-
        compare(< , X , Y), partition(Xs ,Y, M, Bs).
partition([X|Xs], Y, [X|L], Bs) :-
        compare(= , X , Y), partition(Xs, Y, M, Bs).
partition([X|Xs], Y,Ls, [X|B]) :-
        compare(> , X , Y), partition(Xs, Y,Ls, B).
partition([ ], _, [ ], [ ]).
```


Contrôle de la résolution : Cut (!)

Spécification --- ordre des clauses 1

Rappel :

```
?- trace , membre(X , [a,z,e,r,t,y,u,i,o,p]).
   Call:  ( 8) membre(_G501, [a, z, e, r, t, y, u, i, o, p]) ? creep
   Exit:  ( 8) membre(a, [a, z, e, r, t, y, u, i, o, p]) ? creep
X = a ;
   Redo:  ( 8) membre(_G501, [a, z, e, r, t, y, u, i, o, p]) ? creep
   Call:  ( 9) membre(_G501, [z, e, r, t, y, u, i, o, p]) ?
…
```


Spécification --- ordre des clauses 2

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

?- trace , membrePred(X , [a,z,e,r,t,y,u,i,o,p]).
   Call:  ( 8) membrePred(_G399, [a, z, i, r, t, y, u, i, o, p]) ? creep
   Exit:  ( 8) membrePred(a, [a, z, i, r, t, y, u, i, o, p]) ? creep
X = a ;
fail
?-
```

« contrôle de la résolution »

Cut !

Soit représenter un instant de la résolution par :

$[C_1 , ... , C_p] , B_1 , B_2 , ... , B_n$

C_i : les choix restants à exploiter en cas de rebroussement.

Ils ont rangés du plus ancien C_1 au plus récent C_p.

B_i : la résolvante courante : B_1 le prochain but , B_n le dernier.

Cut !!

Un pas de résolution :

(1)- unification réussie de B1 avec la tête T de la clause

T :- $Q_1 , ..., !, Q_i , ... , Q_q$.

on note le choix : C_{B1}

(2)-nouvelle résolvante et nouvel instant de la résolution :

$[C_1 , ... , C_p, C_{B1}] , Q_1 , ..., ! , Q_i , ... , Q_q , B_2 , ... B_n$

Cut !!!

(3)-TOUT va bien, la résolution progresse jusqu'à !:

$[C_1 , ... , C_p, C_{B1} , ... , C_{Qj}] , !, Q_i,...,Q_q , B_2 , ... , B_n$

(4)- le but ! réussit MAIS : l'instant suivant de la résolution est

$[C_1 , ... , C_p] , Q_i , ... , Q_q , B_2 , ... , B_n$

les choix pour les premiers Q et ceux pour B_1 ont été effacés, coupés !!!

Cut !!!!

(5)-OUBIEN la résolution termine.

OUBIEN le rebroussement nous ramène en ce point :

les choix suivants pour B1 n'existent plus et on a épuisé les choix pour Q_i, alors on prend le choix dans C_p.

D'où le nouvel instant de la résolution :

$[C_1 , ... , C_{p'}] , Av_1 , ... , Av_l , B_2 , ... , B_n$

Avantage du Cut

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X) :- X >= Y.

? - max(3, 5, Z), q(Z),...
```

En cas d'échec de q(5), on n'essaiera pas l'alternative pour max

Exemple d'utilisation « consciente » de Cut

```
fusion([X|Xs], [Y|Ys], [X|Zs]) :-
        X < Y, !, fusion(Xs, [Y|Ys], Zs).
fusion([X|Xs], [Y|Ys], [X, Y|Zs]) :-
        X = Y, ! , fusion(Xs, Ys, Zs).
fusion([X|Xs], [Ys], [Y|Zs]) :-
        X > Y, !, fusion ([X|Xs], Ys, Zs).
fusion(Xs, [ ], Xs)  :- !.
fusion([ ], Ys, Ys).
```

Ici chaque clause est exclusive.
Le programme devient déterministe d'où l'intérêt du cut.

Inconvénients du Cut

Ce programme

```
max (X,Y,Y) :- X =< Y, !.
max (X,Y,X).
```

a la même sémantique opérationnelle (et est encore plus efficace que le précédent). Mais pas de sémantique déclarative ou de sémantique du point fixe… ET

```
?- maxRed(10 , 40 , 10).
Yes
```

Prolog cesse alors d'être un langage déclaratif ! (un mauvais Pascal ?)

Du bon usage du Cut : Il doit pouvoir être ignoré dans la lecture déclarative du programme :

cut vert sinon : cut rouge

Négation, cut

call(P) : but variable
P est un prédicat
lance le but ?- P

not : Négation par l'échec

```
not(P) :- call(P), !, fail.
not(P).
```