Cours de Programmation Déclarative et Bases de Données

Prolog - arithmétique et listes

Nicolas Jouandeau

n@up8.edu

2022

arithmétique

- le prédicat `is` provoque l'évaluation d'une expression arithmétique
- le terme à droite de `is` définit l'expression à évaluer
- le terme à gauche de `is` reçoit la valeur correspondante
- `A is EXP` s'écrit également `is(A,EXP)`
- `is` est un prédicat infixe d'arité 2

opérations arithmétiques

```
?- X=2, Y is X+1.          %addition
X = 2,
Y = 3.

?- X is 10-1.              %soustraction
X = 9.

?- X is 4*(1+2).           %multiplication
X = 12.
```

opérations arithmétiques[a] (suite)

```
?- X is 10//3.        %division à l'entier inférieur
X = 3.

?- X is mod(-1, 3).   %modulo d'une valeur
X = 2.

?- X is rem(-1, 3).   %reste de la division à l'entier inférieur
X = -1.

?- X is 10/3.         %division flottante
X = 3.3333333333333335.

?- X is 2**3.         %puissance
X = 8.

?- Y is sqrt(2)-abs(2). %racine carrée et valeur absolue
Y = -0.5857864376269049.
```

[a] division à l'entier inférieur, division entière et modulo sont liés et sont représentés différemment selon les langages; en C, `//` représente la division entière et on a `(-10)//3=-3` et `(-10)%3=-1`; en Prolog (comme en Racket et en Python), `//` représente la division à l'entier inférieur et on a `(-10)//3=-4` et `(-10)%3=2`.

opérations arithmétiques (fin)

```
?- X is min(-1, 3).          % minimum
X = -1.

?- X is max(-1, 3).          % maximum
X = 3.

?- X is sign(-3).            % signe (0, 1 ou -1)
X = -1.

?- X is cos(0.5).            % cosinus
X = 0.877582561890372 8.

?- X is sin(0.5).            % sinus
X = 0.479425538604203.

?- X is tan(0.5).            % tangente
X = 0.546302489843790 5.

?- X is log(0.5).            % logarithme
X = -0.693147180559945 3.

?- X is exp(0.5).            % exponentiel
X = 1.648721270700128 2.
```

transtypage (i.e. cast)

```
?- X is integer(1.0).  %cast en entier
X = 1.

?- X is integer(1.6).
X = 2.

?- X is float(1).      %cast en float
X = 1.0.
```

opérateurs bit à bit

```
?- X is 1/\2.      %et bit à bit
X = 0.

?- X is 1/\3.
X = 1.

?- X is 1\/3.      %ou bit à bit
X = 3.

?- X is 2\/3.
X = 3.

?- X is 4\/3.
X = 7.

?- X is 1<<3.      %décalage à gauche
X = 8.

?- X is 8>>1.      %décalage à droite
X = 4.
```

différence entre unification et égalité

- l'opérateur = réalise l'unification
- l'opérateur == réalise un test d'égalité
  - une variable libre est unifiable avec tout terme (i.e. elle comprise)
  - une variable libre n'est pas égale de tout autre terme

exemple

```
?- ab=X.          % unification entre symbole et variable libre
X = ab.

?- X=Y.           % unification entre variables libres
X = Y.

?- X=ab, X==ab.   % unification puis test d'égalité
X = ab.

?- X==ab.         % test d'égalité entre symbole et variable libre
false.

?- X==Y.          % test d'égalité entre variables libres
false.
```

opérateurs de comparaison

- `X` est égal à `Y`
  `X == Y`
- `X` est différent de `Y`
  `X \= Y`
- `X` est inférieur strict à `Y`
  `X < Y`
- `X` est supérieur strict à `Y`
  `X > Y`
- `X` est inférieur ou égal à `Y`
  `X =< Y`
- `X` est supérieur ou égal à `Y`
  `X >= Y`

opérateurs de comparaison avec évaluation

- `X` est égal à `Y`
  `X =:= Y`
- `X` est différent de `Y`
  `X =\= Y`
- l'évaluation impose que `X` et `Y` soient des expressions arimthétiques

exemple

```
?- 2+2=:=5-1.
true.

?- 1+1=\=5-1.
true.
```

ordre des termes

- au delà des nombres, les termes Prolog sont ordonnés
- l'ordre du plus petit au plus grand est variables, réels, entiers, atomes par ordre alphabétique, termes composés
- les opérateurs sont `@<`, `@>`, `@=<`, `@>=`

exemple

```
?- @<(DEUX,1.0).
  true.

?- @<(1.0,1).
  true.

?- @<(1,blabla).
  true.

?- @<(bla,blabla).
  true.
```

prédicats de tests sur le type d'un terme

- `var(?term)` satisfait si le paramètre est non instancié
- `nonvar(?term)` satisfait si le paramètre est instancié
- `atom(?term)` satisfait si le paramètre est instancié par un atome
- `integer(?term)` satisfait si le paramètre est un entier
- `float(?term)` satisfait si le paramètre est un réel
- `number(?term)` satisfait si le paramètre est un entier ou un réel
- `atomic(?term)` satisfait si le paramètre est un nombre ou un atome
- `compound(?term)` satisfait si le paramètre est un terme composé (i.e. liste non vide ou structure)
- `callable(?term)` satisfait si le paramètre est un atome ou un terme composé
- `is_list(?term)` satisfait si le paramètre est une liste

syntaxe de déclaration des listes

- une liste se note entre crochets
- `[a,b,c]` est une liste de 3 symboles `a`, `b`, `c`
- tête et queue sont délimitées par le symbole "`|`"
- demander l'unification de `[a,b,c] = [T | Q]`
  est équivalent à demander les unifications de `T = a` et de `Q = [b,c]`
- `[a]` s'écrit également `[a|[]]`
- `[a,b]` s'écrit également `[a|[b]]`
- `[a,b,c]` s'écrit également `[a|[b|[c]]]`

exemple d'unifications de listes

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

?- [] = [T | Q].
false.

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

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

?- [a,b,c] = [A,B|C].
A=a , B=b , C=[c]
```

prédicats sur les listes

```
?- length([a,b,c], L).          %taille de liste
L = 3.

?- member(b, [a,b,c]).          %être un élément d'une liste
true .

?- is_list([a,b]).              %être une liste
true.

?- is_list(a).
false.

?- append([a],[b],C).           %concaténation de listes
C = [a, b].

?- prefix([a,b],[a,b,c]).       %être le début d'une liste
true.

?- prefix([b],[a,b,c]).
false.

?- nextto(b,c,[a,b,c,d]).           %successeur d'un élément
  true.
```

prédicats sur les listes (suite)

```
?- select(b, [a,b,c,b], X). %suppression de la première occurence
X = [a,c,b].

?- delete([a,b,c,b],b,C).   %suppression de toutes les occurences
C = [a,c].

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

?- nth0(1,[x,y,z],y).       %nième élément (indice partant de 0)
true.

?- nth1(2,[x,y,z],y).       %nième élément (indice partant de 1)
true.

?- last([a,b,c],c).         %dernier élément
true.

?- nth0(1,[0,a,0,a],a, X).  %suppression du nième élément
X = [0, 0, a].

?- nth1(4,[0,a,0,b],b, X).  %suppression du nième élément
X = [0, a, 0].
```

prédicats sur les listes (suite)

```
?- same_length([a,b,c],[a,b,c]).   %listes de même taille
true.

?- reverse([a,b,c],A).             %inversion
A = [c, b, a].

?- reverse([a,b,c],[c,b,a]).
true.

?- flatten([[a,b],[c,d]],L).       %aplatissement
L = [a, b, c, d].

%(ordre standard : variables<réels<entiers<atomes<complexes)

?- max_member(5,[1,5,3]).      %plus grand élément
true.                          %(dans l'ordre standard)

?- min_member(1,[1,5,3]).      %plus petit élément
true.                          %(dans l'ordre standard)
```

prédicats sur les listes (fin)

```
?- max_member(X,[X,1,PI]).
X = 1.

?- max_member(X,[1,a,11.0,(0,0)]).
X =  (0, 0).

?- sum_list([0,1,2,3],X).    %somme
X = 6.

?- max_list([0,1,2],2).      %max (pour des valeurs numériques)
true.

?- min_list([0,1,2],X).      %min (pour des valeurs numériques)
X=0.

?- numlist(1,5,X).           %générateur
X = [1, 2, 3, 4, 5].
```

mélange d'une liste

- le prédicat `permutation/2` permet d'énumérer les permutations d'une liste
- le prédicat `random_permutation/2` en prend une aléatoirement

permutations de liste

```
?- permutation([a,b,c],X).
X = [a, b, c] ;
X = [a, c, b] ;
X = [b, a, c] ;
X = [b, c, a] ;
X = [c, a, b] ;
X = [c, b, a] ;
false.

?- permutation([a,b,c],[c,a,b]).
true .

?- random_permutation([a,b,c],X).
X = [b, c, a].
```

syntaxe de déclaration des ensembles

- un ensemble est une liste sans doublon

prédicats sur les ensembles

```
?- list_to_set([1,2,3,3],X).          %ensemble
X = [1, 2, 3].

?- is_set([1,3,4]).                   %être un ensemble
true.

?- is_set([1,3,4,4]).
false.

?- intersection([1,2,3],[2,3,4],X). %intersection
X = [2, 3].

?- union([1,2,3],[2,3,4],X).          %union
X = [1, 2, 3, 4].

?- subset([1,4],[1,3,4,6]).           %appartenance
true.
```

tris des éléments d'une liste

```
?- sort([3,1,2],X).    %tri sans les doublons
X = [1, 2, 3].

?- sort([3,3,1,3,2,3],X).
X = [1, 2, 3].

?- msort([3,1,2,3],X). %tri avec les doublons
X = [1, 2, 3, 3].

?- keysort([3-[a],1-[b],2-[c]],X). %tri de listes
X = [1-[b], 2-[c], 3-[a]].         %indexées par des clés
```

spécification des prédicats dans la documentation

- les arguments des prédicats sont instanciés selon des modes
- les modes sont spécifiés par des caractères placés devant les noms des arguments
  - `+` signifie le terme est instancié à l'appel du prédicat
  - `-` signifie le terme est une variable instanciée si le prédicat réussit
  - `++` signifie le terme est remplacé par sa valeur à l'appel du prédicat
  - `-` signifie le terme est lié si le prédicat réussit
  - `?` signifie le terme est lié à un terme partiel à l'appel du prédicat
  - `:` signifie le terme est un meta-argument (i.e. argument d'argument)
  - `@` signifie le terme n'est pas instancié à l'appel du prédicat
  - `!` signifie le terme est une structure modifiable

exemple pour `min_list/1`

```
min_list(+List:list(number), -Min:number)
```

construire des listes de N termes

- définir le prédicat `mklist(T,N,L)` qui est vrai si `L` est une liste de `N` termes `T`

exécution

```
?- mklist(a,5,L).
L = [a, a, a, a, a] .

?- mklist([a,b],3,L).
L = [[a, b], [a, b], [a, b]] .
```

construire des listes de N termes

- définir le prédicat `mklist(T,N,L)` qui est vrai si `L` est une liste de `N` termes `T`

exécution

```
?- mklist(a,5,L).
L = [a, a, a, a, a] .

?- mklist([a,b],3,L).
L = [[a, b], [a, b], [a, b]] .
```

solution pour `mklist/3`

```
mklist(X,1,[X]).
mklist(X,N,[X|L]):-N>1,N1 is N-1,mklist(X,N1,L).
```

dupliquer les termes d'une liste

- définir le prédicat `duplicate(L1,N,L2)` qui est vrai si `L2` est une liste contenant `N` fois les termes de `L1`
- utiliser `mklist/3`

exécution

```
?- duplicate([1,2,3],2,X).
X = [1, 1, 2, 2, 3, 3] .

?- duplicate([1,2,3],3,X).
X = [1, 1, 1, 2, 2, 2, 3, 3, 3] .
```

dupliquer les termes d'une liste

- définir le prédicat `duplicate(L1,N,L2)` qui est vrai si `L2` est une liste contenant `N` fois les termes de `L1`
- utiliser `mklist/3`

exécution

```
?- duplicate([1,2,3],2,X).
X = [1, 1, 2, 2, 3, 3] .

?- duplicate([1,2,3],3,X).
X = [1, 1, 1, 2, 2, 2, 3, 3, 3] .
```

solution pour `duplicate/3`

```
duplicate([],_,[]).
duplicate([A|B],N,X):-duplicate(B,N,X1),
  mklist(A,N,X2),append(X2,X1,X).
```

rotation à gauche d'une liste

- définir le prédicat `rotate_left(L1,N,L2)` qui est vrai si `L2` correspond à `N` rotations vers la gauche de `L1`

exécution

```
?- rotate_left([1,2,3,4],1,X).
X = [2, 3, 4, 1] .

?- rotate_left([1,2,3,4],2,X).
X = [3, 4, 1, 2] .

?- rotate_left([1,2,3,4],5,X).
X = [2, 3, 4, 1] .
```

rotation à gauche d'une liste

- définir le prédicat `rotate_left(L1,N,L2)` qui est vrai si `L2` correspond à `N` rotations vers la gauche de `L1`

exécution

```
?- rotate_left([1,2,3,4],1,X).
X = [2, 3, 4, 1] .

?- rotate_left([1,2,3,4],2,X).
X = [3, 4, 1, 2] .

?- rotate_left([1,2,3,4],5,X).
X = [2, 3, 4, 1] .
```

solution pour `rotate_left/3`

```
rotate_left(A,0,A).
rotate_left([A|B],N,C):-N>0,append(B,[A],X),
  N1 is N-1,rotate_left(X,N1,C).
```

rotation à droite d'une liste

- définir le prédicat `rotate_right(L1,N,L2)` qui est vrai si `L2` correspond à `N` rotations vers la droite de `L1`
  - en utilisant un prédicat `list_and_last(L1,L2,A)` qui est vrai si `L1` est égale à `L2` en ajoutant `A` à la fin de `L2` (i.e. pour `L1 = {1,2,3}`, on a `L2 = {1,2}` et `A = 3`)
  - en utilisant `last/2` et `reverse/2`

exécution

```
?- rotate_right([1,2,3,4],1,X).
X = [4, 1, 2, 3] .

?- rotate_right([1,2,3,4],2,X).
X = [3, 4, 1, 2] .

?- rotate_right([1,2,3,4],5,X).
X = [4, 1, 2, 3] .
```

solution pour `rotate_right/3` avec un prédicat `list_and_last/3`

```
list_and_last([A],[],A).
list_and_last([A|B],[A|C],D):-list_and_last(B,C,D).

rotate_right(A,0,A).
rotate_right(B,N,C):-N>0,list_and_last(B,D,A),
  N1 is N-1,rotate_right([A|D],N1,C).
```

solution pour `rotate_right/3` avec `last/2` et `reverse/2`

```
rotate_right(A,0,A).
rotate_right(B,N,C):-N>0,N1 is N-1,
  last(B,B1),reverse(B,[_|Btmp]),reverse(Btmp,B2),
  rotate_right([B1|B2],N1,C).
```

être un palindrome

- définir le prédicat `palin(L)` qui est vrai si `L` est un palindrome
  - en utilisant `length/2`, `list_and_last/3`
  - en utilisant `length/2`, `last/2` et `reverse/2`
  - en utilisant `last/2` et `reverse/2` (i.e. sans `length/2`)

exécution

```
?- palin([1]).
true .

?- palin([1,1]).
true .

?- palin([1,2,1]).
true .

?- palin([1,2,3,2,1]).
true .
```

solution pour `palin/1` avec `length/2`, `list_and_last/3`

```
palin([]).
palin([_]).
palin(L):-length(L,X),X>1,L=[C|D],
  list_and_last(D,A,B),B==C,palin(A).
```

solution pour `palin/1` avec `length/2`, `last/2` et `reverse/2`

```
palin([]).
palin([_]).
palin(L):-length(L,X),X>1,L=[La|A],
  last(L,La),reverse(A,[_|B]),palin(B).
```

solution pour `palin/1` avec `last/2` et `reverse/2`

```
palin([]).
palin([_]).
palin([A,B|C]):-last([B|C],A),reverse([B|C],[_|D]),palin(D).
```

un élément aléatoire d'une liste

- définir le prédicat `random_elt(L,R)` qui est vrai si `R` est un élément aléatoire de `L`
  - en utilisant `length/2`, `random/3` et `nth0/3`

exécution

```
?- random_elt([],X).
false.

?- random_elt([1,2,3,4,5],X).
X = 1.

?- random_elt([1,2,3,4,5],X).
X = 4.
```

un élément aléatoire d'une liste

- définir le prédicat `random_elt(L,R)` qui est vrai si `R` est un élément aléatoire de `L`
  - en utilisant `length/2`, `random/3` et `nth0/3`

exécution

```
?- random_elt([],X).
false.

?- random_elt([1,2,3,4,5],X).
X = 1.

?- random_elt([1,2,3,4,5],X).
X = 4.
```

solution pour `random_elt/2`

```
random_elt(L,R):-length(L,X),X>0,random(0,X,A),nth0(A,L,R).
```

autre solution pour `random_elt/2`

```
random_elt([A|B],R):-length([A|B],X),random(0,X,C),
  nth0(C,[A|B],R).
```

couple aléatoire de 2 éléments d'une liste

- définir le prédicat `random_couple(L,R)` qui est vrai si `R` est un couple aléaoire de 2 éléments de `L`
  - en utilisant `random_elt/2` et `select/3`
  - en utilisant `random_permutation/2`

exécution

```
?- random_couple([1],X).
false.

?- random_couple([1,2],X).
X = [2, 1] .

?- random_couple([1,2,3,4],X).
X = [4, 3] .
```

solution avec `random_elt/2` et `select/3`

```
random_couple(A,[X,Y]):-random_elt(A,X),select(X,A,B),
  random_elt(B,Y).
```

solution avec `random_permutation/2`

```
random_couple([A,B|C],[A1,B1]):-
  random_permutation([A,B|C],[A1,B1|_]).
```