Prolog

Opérateurs et expressions arithmétiques

Listes

Expressions arithmétiques

- Prolog connaît les entiers et les nombres flottants.
- Syntaxe habituelle pour les opérateurs classiques +, - , *, division entière (symbole //), division flottante (symbole /).

- Différence : opérateur infixe **is** permet d'évaluer les expressions :
  - ?- X is 3+2. ➔ X=5;
  - ?- X is 8 // 3. ➔ X=2;
  - ?- X is 4 * (3+2). ➔ X=20;

Représentation des expressions

- Expressions représentées par des **arbres** Prolog ;
  - expression (X+Y) représentée par l'arbre +(X,Y)
- Donc :

?- 3+2 = 2+3. ➔ no, 2 arbres différents

?- 3+2 is 2+3. ➔ no, partie droite évaluée et partie gauche est un arbre

?- 3+2 = 3+2. ➔ yes, 2 arbres identiques

?- 3+2 is 3+2. ➔ no, partie droite est évaluée et partie gauche est un arbre

Expressions arithmétiques

- attention à certaines tentatives d ’unification
  - la tentative d’unification entre *3+2* et *5* échouera :
    - l’expression *3+2* est un arbre
    - 5 est un nombre.
  - *<?Terme>* ***is*** *<+Expression>* s'efface si Terme est unifiable avec le résultat de l'évaluation de Expression
    - **?- 5 is 2 + 3. s'efface (yes)**
    - **?- N is 2 + 3. s'efface et donne: N = 5**
  - L'évaluation des expressions ne fait pas partie de l’algorithme d’unification => ce n’est plus de la logique

Prédicats de comparaison

- Comparaison pour les expressions arithmétiques:
  - Prédicats binaires et infixés;
  - Évaluation des expressions à gauche et à droite de l'opérateur;
  - X **=:=** Y ➔ X est égal à Y
  - X **=\=** Y ➔ X est différent de Y
  - X **<** Y ➔ X est strictement inférieur à Y
  - X **=<** Y ➔ X est inférieur ou égal à Y

- Exemples : 5+2 =< 5+3. ➔ yes

  5+2 =:= 5+3. ➔ no

  5+2 =\= 5+3. ➔ yes

Évaluation, identité formelle, unification

- **Évaluation des expressions arithmétiques**, puis comparaison :
  5+3 =:= 3+5. ➔ yes
  5+3 =:= 5+3. ➔ yes
  5+X =:= 5+3. ➔ erreur, car X n'est pas connu
  entree(crudites) =:= entree(crudites). ➔ erreur

- **Identités formelles :** égalités des termes au nom des variables près
  5+3 == 3+5. ➔ no
  5+3 == 5+3. ➔ yes (identité des 2 arbres)
  5+X == 5+3. ➔ no
  entree(crudites) == entree(crudites). ➔ yes

- **Unification** (il existe une substitution)
  5+3 = 3+5. ➔ no
  5+3 = 5+3. ➔ yes
  5+X = 5+3. ➔ X=3 (il existe une substitution permettant l'unification)
  entree(crudites) = entree(crudites). ➔ yes

Évaluation, identité formelle, unification

- ?- A=3.

  ➔ A = 3 ;

  il existe une substitution {A:3}

- ?- A==3.

  ➔no

  la variable A n'est pas connue, donc ne peut pas être identique à 3

- ?- A=3, A==3.

  ➔ A = 3 ; no

- ?- A==3, A=3.

  ➔ no

Évaluation, identité formelle, unification

- Négation :
  - évaluation =:=

    négation =\= ➔ résultat de l'évaluation différente

  - identité formelle ==

    négation \== ➔ termes non identiques

  - unification =

    négation \= ➔ termes non unifiables

Coupure et contrôle de l'interpréteur

Notion de coupure

- Différents noms : coupure, cut ou coupe-choix
- Introduit un contrôle sur l'exécution du programme
  - en élaguant les branches de l'arbre de recherche
  - en rendant les programmes plus simples et efficaces
- Différentes notations : ! ou / sont les plus courantes

Le coupe-choix permet de signifier à Prolog que l'on ne désire pas conserver les points de choix en attente.

Notion de coupure

- Le coupe-choix permet :
  - d'éliminer des points de choix
  - d'éliminer des tests conditionnels que l'on sait inutile

- Quand Prolog démontre un coupe-choix, il détruit tous les points de choix créés depuis le début de l'exploitation du paquet des clauses du prédicat où le coupe-choix figure.

Exemples

- menu_simple(E,P,D).
- menu_simple(E,P,D),!.
- menu_simple(E,P,D),poisson(P).
- menu_simple(E,P,D),poisson(P),!.
- menu_simple(E,P,D),!,poisson(P).

  - Que se passe-t-il dans chacun de ces cas ?

Repeat et fail

- Le prédicat *fail/0* est un prédicat qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure.
- Le prédicat *repeat/0* est une prédicat prédéfini qui est toujours démontrable mais laisse systématiquement un point de choix derrière lui. Il a une infinité de solutions.
- L'utilisation conjointe de *repeat/0*, *fail/0* et du coupe-choix permet de réaliser des boucles.

Fail : exemple

```
/* est le chien de */
chien(medor, pierre).
chien(medor, julie).
chien(medor, arthur).

/* avoir un chien */
a_un_maitre1(C) :- chien(C,E).

a_un_maitre2(C) :- chien(C,E), write(E).

a_un_maitre3(C) :- chien(C,E), write(E),
                   tab(1), fail.
```

```
/* requêtes */
?- a_un_maitre1(medor).
Yes

?- a_un_maitre2(medor).
pierre
Yes

?- a_un_maitre3(medor).
pierre julie arthur
```

La négation en Prolog

L'argument not/1 est défini par

```
not(X):-X,!,fail.
not(X).
```

Si X s'efface alors not(X) échoue,

sinon not(X) réussit.

- Négation par l'échec (raisonnement en monde fermé):
  - Un but not(P) est réussi lorsque la résolution du but P échoue
  - Ce qui ne peut pas être prouvé est considéré comme étant faux

La négation en Prolog

- Deux contraintes :
  - Un not ne peut intervenir que dans le corps d'une clause
  - Pour satisfaire not(P), il faut que toutes les variables de P soient liées
- Exemples:

```
sportif(X) :- fort(X),not(petit(X)).
moyen(X) :- personne(X), not(grand(X)),
not(petit(X)).
```

- Contre Exemples :

```
not(grand(X)) :- petit(X).
moyen(X) :- not(grand(X)),not(petit(X)).
```

Listes

Termes simples et Termes composés

- La notion de ‘termes’ est plus générale en Prolog qu’en logique des prédicats du 1er ordre

- Termes simples
  - constantes
  - variables

- Termes composés
  - arbres binaires
  - listes

Arbres binaires

- Un arbre binaire est :
  - soit vide : **abv/0**
  - soit composé d'une racine et de 2 sous-arbres binaires

    ab(Rac, FG, FD)

- En IA, la structure d'arbre :
  - A*
  - minimax
  - alpha-beta (notamment dans des jeux)

Listes

- Structure de données
  - traitements récursifs
  - très utilisées

- Définition récursive :
  - la liste vide, représentée par **[ ],** est une liste,
  - si T est un terme et L une liste, le terme **.(T, L)** représente la liste de premier élément T (ou "tête de liste"), et L est la liste privée du premier élément (ou "queue de liste").
  - • est l'opérateur binaire de séquence.

Listes

- Notations
  - **.(T, L)**
  - **[T | L] ,** avec l'opérateur **|** (« cons ») de construction de liste.
- Exemples

.(a, [ ]) est représentée aussi par **[a],** liste d'un élément

.(a, .(b, [ ])) équivalent à **[a, b]** ou **[a | [b]]**

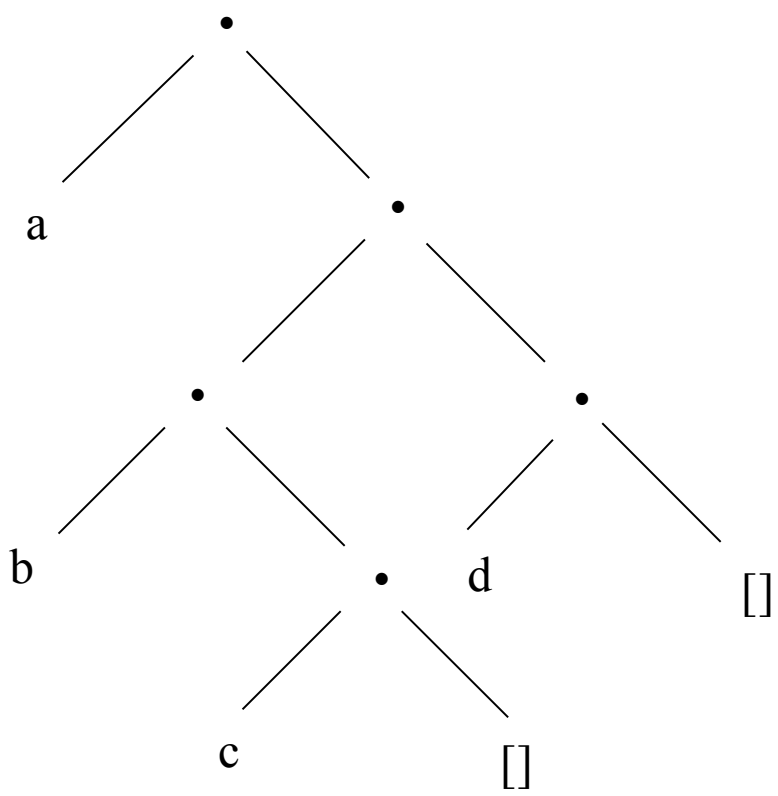
.(a, .(b, .(c, [ ])) équivalent à **[a, b, c]** ou **[a | [b,c]]** ou **[a, b | [c]]**

Exemples

- Plusieurs représentations de la même structure de liste :

  .(a, .(.(b, .(c, [ ])), .(d, [ ])))
  [a, [b,c], d]
  [a | [[b,c], d]]
  [a, [b,c] | [d]]

- dont l'arbre binaire est le suivant :



Unification sur les listes

- Une liste non vide est représentée par [X | Ls]

- ?- [[il, fait], beau,[a,paris]] = [X,Y].
  - no, une liste de 3 éléments ne peut s'unifier à une liste de 2 éléments
- ?- [a, [b, c], d] = [X, Y, Z] .
  - X = a; Y = [b,c]; Z = d ;no
- ?- [a, b, c, d] = [a, b | L] .
  - L = [c,d] ;no
- ?- [a, [b, c], d]=[a, b | L].
  - no

Prédicats sur les listes

- appartenance d'un terme à une liste:

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

- concaténation de deux listes

/* concat(L1,L2,LR) : "LR est la liste résultant de la concaténation des deux listes L1 et L2" */

```
concat([],L,L).
concat([X|Ls],L2,[X|Lc]):- concat(Ls,L2,Lc).
```

Prédicats sur les listes

- Longueur d'une liste

```
long([ ], 0) .
long([X|Ls],N) :- long(Ls, N1), N is N1 + 1.
```

- Somme des éléments d'une liste

```
somme([ ],0).
somme([A|B],C) :- somme (B,D), C is D+A.
```

supprimer(+X,+Xs,?Ys)

version avec coupure :

```
supprimer(X,[],[]).
supprimer(X,[X|Xs],Ys) :- !,supprimer(X,Xs,Ys).
supprimer(X,[Y|Ys],[Y|Zs]):-supprimer(X,Ys,Zs).
```

exemple de requêtes :

```
?- supprimer(a,[b,a,c,d],Y).
Y = [b, c, d] ;
No
```

- si X est en tête dans la liste, on ne regardera pas la 3ième règle
- on peut aussi écrire `supprimer(X,[],[]):-!.` pour la première règle