TP Prolog #2

Julien Reichert

LSV, ENS Cachan

Vendredi 3 février 2012

1 Récursion

Prolog n'est aucunement dérangé par la définition récursive de prédicats. En fait, l'ordre dans lequel ils sont définis n'importe même pas, il faut juste éviter de croiser des définitions de prédicats, les clauses sont ignorées en cas de discontinuité. Prenons pour exemple le code suivant :

```
| ?- pair(0).
| ?- pair(X) :- X #> 0, Y #= X-1, impair(Y).
| ?- impair(X) :- X #> 0, Y #= X-1, pair(Y).
```

Si on donne comme argument un entier naturel à `pair` ou `impair`, il retournera sa parité. Supposons qu'on ait interverti les deux dernières lignes. Alors Prolog aurait ignoré la dernière clause définissant `pair` et seul 0 aurait satisfait le prédicat.

Vu qu'un prédicat peut en utiliser un autre et vice-versa, on ne sera pas surpris qu'un prédicat peut faire appel à lui-même. Voici la fonction 91 de feu John McCarthy en Prolog :

```
| ?- mccarthy(X,Y) :- X > 100, Y is X-10.
| ?- mccarthy(X,Y) :- X =< 100, XX is X + 11,
mccarthy(XX,YY), mccarthy(YY,Y).
```

Rappel : puisqu'on utilise `is`, il faut que `X` soit totalement instancié. On peut donc trouver l'image par un entier de la fonction, mais pas les antécédents, ce qui peut encore une fois s'arranger par l'aménagement suivant, grâce auquel on peut connaître `mccarthy(X,91).`

```
| ?- mccarthy(X,Y) :- X #> 100, Y #= X-10.
| ?- mccarthy(X,Y) :- X #=< 100, XX #= X + 11,
mccarthy(XX,YY), mccarthy(YY,Y).
```

Remarque importante : il faut absolument définir les variables à part avant d'utiliser un prédicat, car Prolog refuse par exemple une clause de type

```
| ?- mccarthy(X,Y) :- X #=< 100, mccarthy(_!_X+11_!_,YY), mccarthy(YY,Y).
```

2 Les listes

Élément important de la récursion, une liste peut être construite selon des syntaxes diverses et contenir n'importe quoi, vu que Prolog n'est pas typé. Donc il n'y a pas d'inquiétude à ce qu'un élément de la liste soit un entier, une autre liste, etc. La liste vide est notée `[]`, la concaténation se marque par le délimiteur `|` et l'énumération par `,`. Concrètement, la liste contenant les éléments 1, 2 et 3 peut s'écrire indépendamment et entre autres :

```
[1,2,3]
[1,2,3|[]]
[1,2|[3]]
[1|[2,3]]
[1|[2|[3]]]
```

en ajoutant autant de `[<liste>|[]]` que nécessaire pour que le code devienne illisible. Les opérations se font essentiellement par unification : si on veut accéder à la queue de la liste `L`, il suffit de dire que `L = [_|Queue]` (la réponse est `no` si `L` était la liste vide). Pour recoller deux listes, on ne peut pas utiliser la concaténation : essayez ce code :

```
| ?- L = [1,2,3], LL = [4,5,6], [L|LL] = LLL.
```

En effet, `[L|LL]` est la liste `[[1,2,3],4,5,6]` (et `[L,LL]` est la liste `[[1,2,3],[4,5,6]]`). En fait pour obtenir la liste `[1,2,3,4,5,6]` à partir de `L` et `LL`, on va utiliser le prédicat ternaire `append` :

```
| ?- L = [1,2,3], LL = [4,5,6], append(L,LL,LLL).
```

Parmi les prédicats utiles pour les listes, le classique `member(X,L)` existe naturellement, mais on dispose également d'un bien sympathique `findall(X,P,L)` regroupant dans la liste `L` tous les `X` qui rendent le prédicat `P` (mis entre parenthèses s'il n'est pas réduit à une seule condition) valide. Notez qu'il peut y avoir d'autres variables libres dans `P`. Des prédicats analogues à `findall` sont `bagof` et `setof`, leur syntaxe est la même mais la façon de collecter les solutions varient : avec `findall` on obtient une liste ordonnée par les clauses définissant le prédicat, avec `bagof` elle est ordonnée selon les variables libres (qui ne sont pas mentionnées par `findall`) alors qu'avec `setof` les solutions sont triées et les doublons retirés.

3 Backtracking, nondéterminisme et autres joyeusetés

Nous avons déjà vu que si l'interpréteur échoue un test, il essaie les autres par backtracking. Mais à la différence d'autres langages, en cas de succès les tests se poursuivent quand même. Ainsi, l'innocent code suivant pour définir la factorielle fait planter Prolog si on demande à voir une autre solution (en faisant `a` ou `;`) :

```
| ?- fact(N,_) :- N < 0, fail.
| ?- fact(0,1).
| ?- fact(N,X) :- NN is N-1, fact(NN,XX), X is XX * N.
```

En fait, la première branche de l'arbre des possibilités explorée est celle où `N` $<$ `0`, elle échoue certes, mais on explore tout de même la branche récursive, conduisant à une boucle infinie... et boum ! De même, si `N` vaut 0, on explore tout de même la branche récursive, ce qui donne le même résultat. Il suffit donc d'ajouter à cette branche la condition `N` $>$ `0` afin que les cas soient partitionnés.

Une autre solution est d'utiliser le coupe-choix ou *cut*, mais nous verrons cela plus tard. Pour le moment, retenez juste que si vous mettez un `!` dans une clause, une fois le but traité plus aucun test n'est fait.

4 Appel de prédicats

Il est possible, et même parfois bien utile, de mettre un prédicat en argument d'un autre. Son utilisation se fait alors avec le prédicat n-aire `call`, dont le premier argument est le prédicat à appeler et les arguments suivants sont ceux dudit prédicat.

Voici un exemple d'utilisation :

```
| ?- recprint(Pred,N) :- NN #= N-1, recprint(Pred,NN),
call(Pred,N,X), write(X), nl.
```

`recprint(Pred,N)` imprime les valeurs entre 0 et `N` de la fonction `Pred` vue comme un prédicat binaire. Au passage, les fonctions d'entrée/sortie seront détaillées dans le TP suivant.

5 DO NOT WANT !

La négation en Prolog est un ÉNORME piège. Il faut absolument garder à l'esprit que l'interpréteur ne peut pas savoir autre chose que ce dont il dispose par défaut et ce qu'on lui a appris. Ainsi, quand il ne peut pas prouver, il échoue. `\+ predicat(X).` ne permet donc que de vérifier que `predicat(X).` échoue ou n'est pas prouvable, en aucun cas il ne génère les `X` pour lesquels `predicat(X).` échoue. Définissez par exemple `pair`, puis `impair` comme la négation de `pair`. Ensuite demandez deux fois en changeant l'ordre s'il existe un impair entre 1 et 10. Constatez au passage l'importance de l'ordre dans les conjonctions : un prédicat qui génère et teste est toujours mieux.

Grâce au coupe-choix ci-dessus, vous pouvez créer une autre version de la négation :

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

6 Exercices

6.1 Récursion

Simulez des boucles for et while en Prolog pour calculer respectivement la somme des entiers de 1 à un certain `N` et la somme des éléments d'une liste `L`.

6.2 Listes

Écrivez des prédicats pour retirer une fois un élément d'une liste (en échouant s'il n'y figure pas), retirer toutes les occurrences d'un élément d'une liste, retirer toutes les répétitions d'une liste.

Implémentez également le tri de votre choix.

6.3 Backtracking

Adaptez le code de `fact` avec et sans coupe-choix afin qu'il ne fasse plus de boucle infinie. Faites de même avec votre prédicat retirant une fois un élément d'une liste de sorte que l'occurrence retirée soit la première (si ce n'était pas déjà le cas).

Plus difficile maintenant : écrivez un prédicat décidant s'il existe un chemin d'un sommet à l'autre dans un graphe défini par ses arcs. Voici une mauvaise idée :

```
arc(a,b).
arc(b,c).
arc(c,a).
arc(c,d).

connexe(X,X).
connexe(X,Y) :- arc(X,Y).
connexe(X,Z) :- connexe(X,Y), connexe(Y,Z).
```

6.4 Négation

Voici un exercice qui vous fera passer l'envie d'utiliser la négation quand une autre possibilité existe. Demandez quelles sont, parmi les listes qui ne comprennent que des 0 et des 1, celles qui ne vérifient pas une propriété simple, par exemple contenir au moins un 0. Première solution : forcer qu'il n'y ait que des 0 et des 1 puis utiliser `not`. Deuxième solution : caractériser directement les listes qui ne vérifient pas la propriété. Quel prédicat donne un dépassement de pile dès qu'on demande une autre solution que la liste vide ?