TP Prolog #3

Julien Reichert

LSV, ENS Cachan

Vendredi 10 février 2012

1 Coupe-choix

Avant de détailler le comportement du coupe-choix, essayez ces instructions :

```
| ?- !.
| ?- true,!;true.
| ?- !,true;true.
| ?- !,fail;true.
| ?- fail,!;true.
| ?- !,(true;true).
```

Dans l'avant-dernière ligne, l'échec empêche l'effet du coupe-choix de s'appliquer. Ainsi, l'autre option de la disjonction est étudiée. Le coupe-choix s'applique indifféremment si la suite de la branche réussit ou non, et il fait ignorer les branches autres que les descendantes. En particulier, l'endroit où il apparaît est très important si l'échec d'un but antérieur permet de continuer l'exploration. Dans la dernière ligne, il y a bien deux solutions données par Prolog [1] et c'est normal vu que la disjonction est cette fois après le coupe-choix.

Pour vérifier que vous avez bien compris : combien de couples seront retournés dans les trois cas ci-dessous ?

```
| ?- !, member(X,[1,2,3]), member(Y,[1,2,3]).
| ?- member(X,[1,2,3]), !, member(Y,[1,2,3]).
| ?- member(X,[1,2,3]), member(Y,[1,2,3]), !.
```

Moralité : si vous souhaitez obtenir tous les cas possibles [2], trouvez une solution pour ne presque jamais mettre de coupe-choix dans vos prédicats, ou alors à ce qui correspond à un nœud proche de la racine de l'arbre d'exécution.

1. regardez la différence en remplaçant le deuxième `true` par `fail`
2. et vous allez le souhaiter quasi systématiquement dans le projet

2 Entrées/sorties

Pour un peu d'interactivité et surtout pour surveiller la valeur des variables, il est possible d'écrire dans le terminal voire dans un fichier par le prédicat unaire (pour la sortie standard) ou binaire `write`. La sortie est le premier des arguments si deux sont précisés, elle doit être un fichier ouvert grâce au prédicat `open(Chemin,Mode,Flux).` (où `Mode` prend la valeur `read`, `write` ou `append` et `Flux` sera la variable utilisée pour écrire dans le fichier) puis fermé grâce au prédicat `close(Flux).` si on veut conserver les modifications. On peut également lire des instructions par `read` avec la même syntaxe, ces instructions doivent finir par un point et seront traitées comme des variables.

À propos, les chaînes de caractères sont mises entre apostrophes, les guillemets ne donneront pas le résultat escompté [3]. Vous pouvez également donner en argument à write des variables, dont des listes. Elles seront écrites sous la forme la plus simple et la plus instanciée possible dans l'état actuel de l'unification.

Pour aller à la ligne, en plus de l'ajout de '\n' dans la chaîne de caractère écrite, il existe un prédicat 0-aire `nl` (ou unaire si on écrit dans un flux comme avec `write`).

3 Quelques éléments supplémentaires qui peuvent être utiles

3.1 Débogage

Il est possible de suivre l'arbre des exécutions de l'interpréteur en utilisant `trace.` (ceci étant annulé par `notrace.`) et si seul un prédicat doit être suivi, il est intéressant de laisser de côté les autres en faisant `spy(predicat).` (`nospy(predicat).` pour arrêter). Attention cependant, `spy` trace également les sous-prédicats et en pratique tout l'arbre d'exécution sera imprimé à partir de la première occurrence d'un prédicat tracé.

3.2 Assertions dynamiques

Pour ne pas avoir à faire un `consult` à tout va quand un seul prédicat doit être ajouté, il est possible d'ajouter dynamiquement une clause à la base de données grâce aux prédicats unaires `asserta` (ajouter au début) et `assertz` (ajouter à la fin). Pour les retirer, on utilise `retract`.

3. sauf si vous voulez réviser le code ASCII

3.3 Dernier récapitulatif

Gardez cela à l'esprit en programmant : attention à la négation, attention au coupe-choix, attention à l'ordre des prédicats. Un manque de rigueur arrive très vite, et dès que votre programme ne retourne pas ce que vous voulez (un échec alors que vous avez une solution triviale, une boucle infinie, etc.) vous risquerez d'avoir du mal à corriger l'erreur sans faire une grosse déconstruction. Donc il vaut mieux tout de suite limiter le danger, en particulier vous aurez souvent besoin de la méthode «génère et teste». À l'instar de la négation qui, vous l'avez vu, est rarement utilisée autrement que pour exclure des candidats déjà générés, il faut toujours mettre d'abord les prédicats permettant de cerner les solutions potentielles. Ainsi, un prédicat qui reçoit des variables instanciées est plutôt un prédicat de test (exemple typique : [im]pair) et peut être plus libre dans son écriture qu'un prédicat de génération (exemple typique : findall). Il sera tout naturellement plus futé d'écrire les lignes du haut que celles du bas dans les instructions ci-dessous :

```
| ?- power2_inf_100(X) :- X #= 1.
| ?- power2_inf_100(X) :- X #> 0, X #< 100, XX #= X/2, power2_inf_100(XX).

| ?- power2_inf_100_bis(X) :- X #= 1.
| ?- power2_inf_100_bis(X) :- X #> 0, X #< 100, power2_inf_100_bis(XX), XX #= X/2.
```

3.4 Et maintenant, le sort en est jeté

Pour générer un entier aléatoire `Alea` compris entre `Min` et `Max - 1` :

`random(Min,Max,Alea).`

`Alea` doit être une variable, par exemple pour faire un pile ou face il ne faut pas écrire `random(0,2,0).` mais plutôt `random(0,2,X), X = 0.`

4 Exercices

4.1 Coupe-choix

Voici un bout de code issu de la correction du TP suivant. Les coupe-choix y figurant n'ont pas gêné mais on aurait pu écrire un code équivalent sans les y utiliser. Faites-le.

```
| ?- pasdansliste(_,[]).
| ?- pasdansliste(X,[X|_]) :- !,fail.
| ?- pasdansliste(X,[_|L]) :- pasdansliste(X,L).

| ?- retiredeliste(_,[],_) :- fail,!.
| ?- retiredeliste(X,[X|L],L) :- true,!.
| ?- retiredeliste(X,[Y|L],[Y|LL]) :- retiredeliste(X,L,LL).
```

```
| ?- intervalle(1,[1]) :- !.
| ?- intervalle(Max,[Max|L]) :- Max2 is Max-1, intervalle(Max2,L).
```

Appliquez le même principe à l'avenir, car le coût des modifications apportées est négligeable au regard de leurs avantages.

4.2 Entrées/sorties

Dessinez une jolie matrice grâce à write. C'est déjà plus agréable à lire qu'une liste de listes, non ?

4.3 Forall

L'un des exercices du TP suivant consistera à déterminer la région gagnante dans un jeu. Prolog est conçu pour répondre à la question $\exists x, p(x)$ naturellement, mais qu'en est-il de $\forall x, p(x)$? Naturellement, tester de manière exhaustive tous les entiers par exemple est une aberration, mais on peut avoir une version plus faible qui suffira largement dans tous les cas : $\forall x, propfinie(x) \Rightarrow p(x)$. Écrivez-le au moins une fois et gardez en tête la méthode.

4.4 Bon, puisqu'il reste un peu de temps...

... commencez le TP4 !