TP Prolog #1

Julien Reichert

LSV, ENS Cachan

Vendredi 27 janvier 2012

1 Introduction

Prolog (comprenez Programmer en Logique) est un langage basé sur la logique du premier ordre. Un programme en Prolog consiste en une suite de buts, vus comme des formules du premier ordre closes ou non, et l'interpréteur cherchera à les rendre vrais, en cherchant toutes les affectations des variables libres satisfaisant la formule. On peut de ce fait considérer Prolog comme un langage non-déterministe dans la mesure où l'interpréteur ne s'arrête pas une fois le but rempli. Cette séance de TD va mettre en lumière ces propriétés.

2 La base (de la base)*

On utilisera l'interpréteur GNU Prolog (commande `gprolog`). Une fois celui-ci lancé, vous aurez la main sur un prompt `| ?-` où vous allez saisir vos premiers buts. Les booléens de base sont `true` et `fail`, respectivement toujours et jamais rempli. La conjonction s'exprime avec une virgule et la disjonction par un point-virgule, en outre il faut toujours terminer par un point. Ainsi, découvrez ce que répond l'interpréteur sur les saisies suivantes :

```
| ?- true.
| ?- fail.
| ?- true, fail.
| ?- fail, true.
| ?- fail; true.
| ?- true; fail.
```

Le dernier cas est intéressant dans la mesure où on constate que l'interpréteur nous propose une solution et nous demande si on en veut d'autres. Demandez-vous pourquoi ce n'est pas le cas pour `| ?- fail; true.` où la réponse est uniquement `yes`. Moralité : l'ordre importe dans la disjonction et on ne peut pas parler d'évaluation paresseuse puisque Prolog continue les tests.

3 Variables et constantes

En prolog, un terme commençant par une majuscule est une variable et un terme commençant par une minuscule est une constante (les chaînes de caractères, dont les délimiteurs sont des apostrophes, et les entiers en sont aussi). Les noms de prédicats n-aires doivent également commencer par une minuscule. Le souligné représente une variable muette, il est notamment utilisé dans les disjonctions lorsqu'on définit une fonction au cas par cas et que l'un des arguments n'importe pas (variable inutilisée = warning).

Seules les variables peuvent être instanciées, par exemple :

```
| ?- a = b.
| ?- X = a.
| ?- X = Y.
| ?- a = 2.
| ?- X = 42.
| ?- _ = 16.
```

Les répétitions de l'interpréteur correspondent aux affectations des variables qui remplissent le but. Attention, l'opérateur infixe `=` sert à **unifier**, il ne correspond pas à une affectation et en aucun cas à l'égalité arithmétique. Regardez donc le résultat de cela :

```
| ?- X = 2 + 2.
| ?- 2 + 2 = 4.
| ?- 2 + 2 = 2 + 2.
| ?- 1 + 3 = 3 + 1.
| ?- X + 3 = 3 + Y.
| ?- a(X, Y) = a(b, 5).
```

Si vous voulez affecter une variable, l'opérateur infixe à utiliser est `is`, mais cela ne fonctionne pas pour les constantes, où on a recours à l'égalité dite contrainte #= (car utilisée ainsi que tous les autres opérateurs arithmétiques dans la programmation par contraintes) :

```
| ?- X is 2 + 2.
| ?- 2 + 2 is 4.
| ?- X #= 2 + 2.
| ?- 2 + 2 #= 4.
```

Parfois, les solutions seront exprimées sous forme d'intervalle, dans ce cas finir le prédicat par `fd_labeling()` permet de les trouver.

Attention également quand vous écrivez les opérateurs arithmétiques : il ne faut pas former de flèche d'implication, les inégalités larges seront donc `>=` et `=<`. Au passage, l'implication s'écrit `->` et elle échoue quand le but à gauche échoue (!!!).

4 Un petit mot sur le typage

Absent.

5 Définir des prédicats

Admettons que vous vouliez créer un prédicat `impair` d'arité 1 qui soit vrai si et seulement si son argument est pair. Déjà, vous choisissez les noms de prédicats n'importe comment et donc vous serez pénalisés. Ensuite, vous aimeriez savoir s'il existe une commande du genre `let` ou `def`. En fait, Prolog définit les conditions de validité d'un prédicat, en clair s'il est dans le cas favorable décrit, il sait que cela implique que le prédicat est vrai, sinon il ne peut pas conclure et par conséquent il échoue. Ceci s'exprime par l'opérateur infixe `:-` et dans notre exemple on aura donc

```
| ?- impair(X) :- X #= 2 * _.
```

et... Prolog retourne

```
uncaught exception: error(existence_error(procedure,(:-)/2),top_level/0).
```

C'est parce que vous ne pouvez pas créer ainsi des prédicats. Il faut demander à Prolog d'enrichir sa base de données en consultant une liste de telles assertions grâce à

```
| ?- consult('<lien vers un fichier>').
```

ou

```
| ?- consult('user').
```

pour une saisie manuelle (Ctrl+D à la fin). Cette fois, saisir le code annoncé permettra de demander `| ?- impair(2).` et en fait pour n'importe quel entier. Si on demande `| ?- impair(X).`, Prolog retourne une sorte d'ensemble grâce une autre utilisation du souligné. Cela vient de l'utilisation de contraintes avec le souligné dans la définition. Une autre solution (avec un nom correct cette fois) est de définir, toujours après avoir fait un `consult`,

```
| ?- pair(0).
| ?- pair(X) :- X #> 0, XX #= X - 2, pair(XX).
```

mais `pair(X)` décrira successivement toutes les solutions positives avec cette définition sans lister -2, -4, etc [1].

1. à ce propos, répondre 'a' à l'invite après une solution proposée est une MAUVAISE idée quand l'ensemble des solutions est infini, sauf si vous pouvez vite faire Ctrl+C

6 À vous de jouer

Maintenant que vous savez définir des prédicats, c'est le moment de s'exercer avec la généalogie. Créez des personnages par le prédicat `identite(Personne,Naissance,Sexe,Pere,Mere)`, déclarez l'année courante et créez les prédicats binaires `age`, `pere`, `mere`, `fils`, `fille`, `frere`, `soeur`, etc.