N7PD : Miniprojet Prolog

Author : Christophe Garion <garion@isae-supaero.fr>
Public : N7 2SN
Date : 03/03/25

Vous ne devez utiliser que les constructions vues en cours ou celles introduites explicitement dans le sujet. En particulier, les constructions pouvant être utilisées en programmation par contraintes comme `fd_all_different` ne sont pas autorisées.
Les réponses aux questions autres que du code seront insérées sous forme de commentaires dans le code Prolog (utilisation de `/**/` comme en C). N'hésitez pas également à mettre sous forme de commentaires les résultats de vos tests.

Résumé

Le but de ce BE est d'utiliser la programmation logique et le langage Prolog pour résoudre un problème.

1 Introduction de la problématique

Nous nous intéressons ici à un problème classique en Intelligence Artificielle, à savoir la génération de plans. Un plan est une suite d'actions qui permettent d'atteindre un but. Par exemple, pour prendre un café à l'ENSEEIHT, je sais qu'il faut que je me rende à la machine à café au bâtiment C, que je mette suffisamment d'argent dans la machine, que je sélectionne une boisson, que j'attende qu'elle soit prête, que je la prenne et que je récupère ma monnaie.

La principale difficulté dans l'établissement d'un plan est que l'on doit représenter l'évolution de l'environnement lors de l'exécution du plan. En particulier, des faits qui étaient vrais avant une action (par exemple « le cube a est au dessus du cube b ») vont devenir faux après que l'action a été effectuée et d'autres vont devenir vrais.

Dans un langage impératif, on peut modéliser l'environnement avec un état mutable. Dans le langage Prolog, nous avons seulement la possibilité de représenter des *faits* qui représentent la connaissance que l'on a de l'environnement. Ces faits sont stockés dans une *base de faits* que l'on peut faire évoluer.

Dans ce qui suit, la section 2 présente les prédicats **`assertz`**`/1` et **`retract`**`/1` qui permettent de faire évoluer la base de faits. La section 3 présente le problème de planification auquel on s'intéresse et la section 4 contient les questions auxquelles vous devez répondre.

2 Préliminaires : gestion des faits en Prolog

Prolog fournit deux prédicats pour gérer la base de faits :
- **`assertz`**`/1` qui permet d'ajouter un fait à la base
- **`retract`**`/1` qui permet de retirer un fait de la base

Les prédicats **`listing`**`/0` et **`listing`**`/1` permettent respectivement d'examiner les faits et les définitions de tous les prédicats définis dans l'environnement ou d'un prédicat en particulier.

2.1 Un premier exemple interactif

Utilisons l'interpréteur Prolog pour ajouter des faits concernant un prédicat `smart/1` qui indique qu'une personne est intelligente.

Dans un premier temps, on peut vérifier qu'il n'y a aucun fait ou définition concernant `smart/1` :

```
?- listing(smart).

yes
```

On peut ajouter des faits concernant `smart/1` :

```
?- assertz(smart(christophe)).

yes
?- listing(smart).

% file: user_input

smart(christophe).
```

```
yes
```

La sortie de **listing**/1 nous indique d'où proviennent les faits : ici, le seul fait connu sur smart/1 provient de l'interpréteur. On verra par la suite qu'on peut ajouter des faits et des définitions via un fichier.

On peut également ajouter plusieurs faits et en enlever via une seule requête :

```
?- assertz(smart(guillaume)), assertz(smart(aurelie)), retract(smart(christophe)).

yes
?- listing(smart).

% file: user_input

smart(guillaume).
smart(aurelie).

yes
```

2.2 Sémantique de assertz/1 et retract/1

Nous donnons ici une sémantique informelle aux deux prédicats **assertz**/1 et **retract**/1 :

- **assertz**(f) est toujours valide et ajoute une instance du fait f dans la base. On peut donc avoir plusieurs instances du même fait dans la base.
- **retract**(f) est valide s'il existe au moins une instance du fait f dans la base. Dans ce cas, une instance du fait est retirée de la base (il peut donc rester des instances du même fait). Si le fait f n'est pas dans la base, **retract**(f) est évalué à faux.

Le prédicat rectractall/1 permet de retirer toutes les instances d'un fait. Il peut vous servir en particulier en mode interactif, car le chargement d'un fichier ne réinitialise pas la base de faits (on ne fait qu'ajouter des faits à l'environnement actuel).

2.3 Utilisation d'assert/1 et de retract/1 dans le corps d'une clause

Prenons un exemple en utilisant trois prédicats : smart/1 qui indique qu'une personne est intelligente, watch/2 qui indique qu'une personne regarde une vidéo sur un média donné.

On définit un programme dans un fichier watch.pl comme suit :

```
/* on indique que les faits concernant smart/1 évoluent */
:- dynamic(smart/1).

/* on établit quelques faits */
smart(christophe).
smart(guillaume).
smart(aurelie).

/* on définit watch/2 sur deux cas */
watch(X, college_de_france) :- assertz(smart(X)).
watch(X, tiktok)            :- retract(smart(X)).
```

Dans ce fichier :

- la directive :- dynamic(...) permet d'indiquer que les clauses d'un prédicat peuvent changer, en particulier les faits le caractérisant ;
- on définit des faits pour smart/1 ;
- on définit watch/2 en utilisant **assertz**/1 et **retract**/1. Si on demande par exemple à Prolog d'évaluer la requête

```
watch(guillaume, tiktok).
```

il va essayer de satisfaire le but **retract**(smart(guillaume)).

Voici une session interactive utilisant le prédicat watch/2 :

```
?- ['./watch.pl'].
compiling /home/.../watch.pl for byte code...
/home/.../watch.pl compiled, 11 lines read - 922 bytes written, 7 ms

yes
?- listing(smart).
```

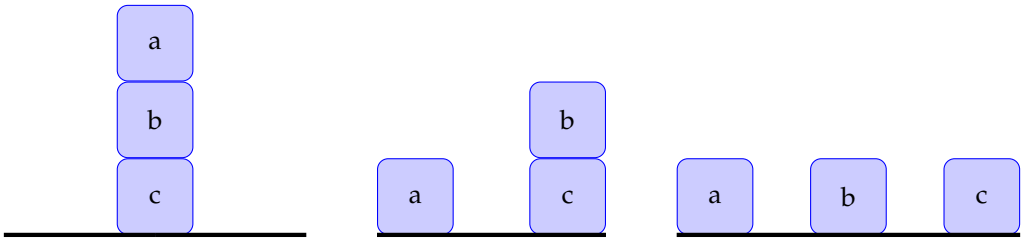
```
% file: /home/.../watch.pl

smart(christophe).
smart(guillaume).
smart(aurelie).

yes
?- watch(guillaume, tiktok).

yes
?- listing(smart).

% file: /home/.../watch.pl

smart(christophe).
smart(aurelie).

(1 ms) yes
?- watch(christophe, college_de_france).

true ? ;

no
?- listing(smart).

% file: /home/.../watch.pl

smart(christophe).
smart(aurelie).
smart(christophe).

yes
```

Remarquez qu'il y a maintenant deux faits `smart(christophe)`.

3 Problème posé

On suppose ici qu'un robot doit déplacer trois cubes. Le robot ne peut prendre qu'un seul cube à la fois, donc si des cubes sont empilés sur un cube à déplacer, le robot doit d'abord les enlever. On suppose que plusieurs cubes peuvent être posés sur la table.

La situation initiale est la suivante (cf. fig. 1) : le cube a est sur le cube b, le cube b sur le cube c et le cube c sur la table. Le robot ne peut pas déplacer directement les cubes b et c à partir de cette situation.



(a) La situation initiale (b) Une situation possible (c) Une autre situation possible

FIGURE 1 – Quelques situations

4 Questions

Dans ce qui suit, on va modéliser la situation du monde par des faits Prolog que l'on va faire évoluer dynamiquement.

1. représenter la situation initiale grâce à un prédicat.

2. on souhaite maintenant représenter l'action *atomique* `put_on` qui consiste à placer directement un objet sur un autre. Pour cela, il faut spécifier l'action. Une action peut se spécifier par :
   – ses préconditions, qui sont les formules qui doivent être vraies pour que l'on puisse réaliser l'action ;
   – ses postconditions, qui sont les formules qui seront vraies après l'exécution de l'action.
   Écrire en français dans un commentaire dans votre fichier les préconditions et les postconditions de l'action `put_on/2` lorsque l'on veut placer un objet a sur un objet b.
3. écrire si nécessaire des prédicats intermédaires pour exprimer les préconditions et les postconditions de `put_on/2`.
4. écrire le prédicat `put_on/2` avec les préconditions et postconditions énoncées précédemment. À noter qu'assurer les postconditions revient à faire évoluer la base de faits.
5. le but de ce projet est de trouver un plan, i.e. une suite d'actions. Il faut donc pouvoir « enregistrer » les mouvements qu'effectue le robot. Modifier la définition de `put_on/2` pour que lorsque l'on appelle `put_on(a,b)` on enregistre un fait `move(a,b)`.
6. poser les requêtes suivante une par une dans l'interpréteur :
   – `put_on(a,table)`
   – `put_on(c,a)`
   – `put_on(b,table), put_on(c,a)`
   Utiliser régulièrement `listing(move)` pour observer l'évolution de la base de faits.
7. `put_on` est une action atomique permet au robot créer des plans plus complexes. Afin de planifier plusieurs actions à la fois, spécifier les deux prédicats suivants :
   – `clear/1` qui permet de libérer un objet, i.e. d'enlever tous les objets situés dessus en les plaçant sur la table.
   – `r_put_on/2` permet de placer un objet sur un autre même s'ils ne sont pas libres initialement.

On peut être amené à vouloir empêcher Prolog de revenir en arrière lorsqu'il a satisfait un but. Cela peut être fait en utilisant le prédicat ! appelé *cut*. ! est toujours évalué à vrai et empêche Prolog d'essayer de « trouver » des solutions supplémentaires. Par exemple, si on définit le prédicat `ancestor` vu en cours comme suit :

```
ancestor(X, Y) :- parent(X, Y), !.
ancestor(X, Y) : parent(Z, Y), ancestor(X, Z).
```

alors on aura une seule réponse à la requête `ancestor(W, mary)`.

8. on cherche maintenant à construire un plan pour un ensemble de buts à atteindre.
   (a) dans un premier temps, on introduit un prédicat `achieve/1` qui prend en paramètre une liste de termes représentant des buts à atteindre et affiche les actions nécessaires à la satisfaction de ces buts. Par exemple, `achieve([on(a,c), on(c,b)])` affichera la liste de mouvements nécessaires à l'obtention de la situation dans laquelle le cube a est sur le cube c et le cube c sur le cube b.
   On ne s'intéressera ici qu'à des buts du type « l'objet x est sur l'objet y », mais le prédicat `achieve` pourrait également servir pour d'autres buts/actions que le robot pourrait effectuer (se déplacer d'une pièce à l'autre etc).
   (b) que se passe-t-il si on demande l'évaluation de `achieve([on(a,c), on(c,b)])` ?
   (c) pour pallier le problème précédent, il faut pouvoir trier correctement la liste des buts à atteindre. Proposer une solution simple permettant de réaliser ce tri *en supposant que la liste de buts est cohérente*.

5 Documents à rendre

Vous devez déposer pour le **4 avril 2025 à 23h00** sur le dépôt Moodle du cours le source d'un programme Prolog répondant aux questions ci-dessus. **Le fichier source devra impérativement s'appeler `cube-login1-login2.pl` où `login1` et `login2` sont les logins de votre binôme.** Les réponses aux questions autres que du code devront y être insérées sous forme de commentaires.

License CC BY-NC-SA 3.0