Séance 1 du Projet 4IF ALIA – Octobre 2019
Découverte de Prolog

J-F. Boulicaut – S. Calabretto

Nous vous proposons de travailler des exercices pour maîtriser certains mécanismes assez particuliers de la programmation en Prolog. Votre effort d'appropriation des concepts du langage (explication du codage des prédicats, usage de l'unification, approche rigoureuse dans les tests avec notamment l'étude des différentes configurations d'appel) est indispensable pour s'engager dans la réalisation d'un projet de programmation sur les deux séances suivantes. La prise en main du langage peut également s'appuyer sur les exemples montrés pendant les cours et codés pour vous dans le fichier « intro_prolog.pl ». Il est important de savoir consulter la documentation en ligne de SWI-prolog pour comprendre les tâches réalisées par les prédicats fournis (Built-in Predicates).

1. Thème de la généalogie

Créer une base de connaissances dans laquelle on définit en extension certaines relations familiales (typiquement la parenté directe) et quelques autres informations comme, par exemple, le genre. On peut alors étudier le codage de nouvelles relations comme « ancêtre », « frère » ou « frère ou sœur » ou bien « oncle ou tante ». Ce domaine simple est très bien pour découvrir les mécanismes d'aide à la mise au point (« debug » avec activation des traces – trace, notrace, spy, nospy, …) et les prédicats prédéfinis indispensables (comme, par exemple, pour les E/S avec read, write, nl, etc).

2. Manipulation de listes

Coder un prédicat « element(Obj,List,Nlist) » (element/3) qui est vrai lorsque le troisième argument représente la liste qui est en second argument privée de toutes les occurrences de l'élément qui est en premier argument (c'est le travail réalisé par le BIP select/3).

?-element(b,[a,b,a,c],R). se prouverait avec R=[a,c] ;

?-element(X,[a,b,a, c],R). se prouverait avec X=a et R=[b,c] mais aussi X=b et R=[a,a,c], etc.

Tester tous les modes d'utilisation « convenables » et observer comment l'interprète Prolog procède. Comprendre notamment pourquoi un but comme ?-element(a,L,[b,c]) ne pourra pas être prouvé.

Ecrire la concaténation de listes (le prédicat « append » a été présenté en cours), tester et observer les modes de fonctionnement. En exploitant ce prédicat de concaténation, réaliser l'inversion d'une liste au moyen d'un prédicat inv\2 : on souhaite que la preuve de inv([a,b,c],X) réussisse avec X=[c,b,a]. Tester les divers modes d'appel et constater que certains posent des problèmes. Trouver une solution en testant à l'appel la nature des arguments (usage de « var » et « novar » qui testent si une variable est libre ou liée). Utiliser les outils de trace pour « mesurer » la différence entre un codage de type « naive reverse » et le renversement de liste efficace au moyen d'un accumulateur également vu en cours. Le renversement de listes est possible avec le BIP reverse/2.

On souhaite maintenant un prédicat « composante » permette un accès par le rang dans une liste (en comptant à partir de 1, c'est le travail réalisé par le BIP nth1/3).

?-composante(3,X,[a,b,c,d]). devrait réussir avec X=c.

On veut pouvoir aussi retrouver le rang :

?-composante(I,a,[a,b,a]). devrait fournir les réponses I=1 et I=3.

Il serait bien que le même prédicat puisse être utilisé pour ces différents modes d'appel.

Nous pouvons maintenant considérer la manipulation d'ensembles (définis en extension) représentés par des listes (sans répétition) d'objets.

Ecrire un prédicat list2ens/2 pour tester si une liste est bien un ensemble (pas de répétition). Ecrire un prédicat qui produise un ensemble (enlève les doublons). Ainsi « list2ens([a,b,c,b,a],E) » devrait réussir avec E=[a,b,c]. On s'interdira l'utilisation de « sort » et de « setof ». On peut ensuite décrire l'union, l'intersection, la différence, l'égalité (sans utiliser « sort ») de deux ensembles. Quel aurait été l'avantage, dans ce dernier cas, d'utiliser sort/2 (tri selon « @< ») ?

Ecrire un prédicat qui permette l'expression de substitutions dans une liste. Ainsi « subsAll(a,x,[a,b,a,c],R) » devrait réussir avec R=[x,b,x,c]. Modifier sa définition pour qu'il soit reprouvable en ne faisant chaque fois qu'une substitution.

3. Thème sur les chaînes de caractères et opérateurs

En prolog 'Bonjour' est un symbole et les « quotes » masquent le fait que ce devrait être pris pour une variable. Mais "bonjour" est considéré comme une liste de codes ASCII.

Ainsi ?-"bonjour"=[98,111,110,106,111,117,114] réussit.

Le prédicat « name(Nom,Chaine) » permet d'éclater un nom en la liste des codes ASCII et réciproquement. Par exemple, on peut concaténer deux noms ainsi :

?- name('jean',N1),name('Paul',N2),append(N1,N2,N3),name(Result,N3),
Result==jeanPaul. réussit.

Ecrire un prédicat qui teste si un nom est avant l'autre dans l'ordre alphabétique. Ainsi, « ?- avant(adam,eve). » doit réussir, mais « ?- avant(adam,ada). » doit échouer. Faire que « avant » soit reconnu comme opérateur, c'est-à-dire que l'on puisse écrire indifféremment « avant(adam,eve) » ou « adam avant eve » (voir le prédicat prédéfini « op/3 »).

NB. Pour cette phase d'appropriation, cette liste de thèmes n'est pas exhaustive, vous pouvez aussi regarder les prédicats de parcours de graphes qui ont été présentés, essayer de construire et d'exploiter une table arborescente pour gérer efficacement un dictionnaire, développer l'arborescence des coups possibles dans un jeu à 2 joueurs simple comme un morpion, ou encore regarder des problèmes typiques de programmation sous contraintes (colorations de cartes/graphes et ordonnancements, puzzle logiques comme les sudokus, etc).