Séance 1 du Projet 4IF ALIA – Octobre 2021
Découverte de Prolog

Jean-François Boulicaut

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 s'appuyer sur les exemples montrés pendant les cours et codés pour vous dans le fichier « intro_prolog.pl ». Il faut consulter la documentation SWI-prolog pour comprendre 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 comme la parenté directe avec un prédicat parent(X,Y) pour dire que X est le parent de Y et, au minimum, deux autres prédicats désignant le genre, désignons les comme masculin(X) et feminin(X).

(a) Programmer la définition du prédicat grand_parent(X,Y) qui réussit quand X est le grand-père ou la grand-mère de Y. Tester le prédicat et découvrir les mécanismes d'aide à la mise au point (« debug » avec activation des traces – trace, notrace, spy, nospy, …). On peut notamment écrire un prédicat qui va assurer la lecture du nom d'un individu et afficher l'ensemble de tous ses petits-enfants. On utilisera ici les prédicats E/S prédéfinis indispensables (comme, par exemple, read, write, nl).

(b) Programmer la définition du prédicat ancetre(X,Y) qui réussit quand X est un ou une ancêtre de Y. Tracer la preuve de ce prédicat sur des exemples et étudier les variantes de codage possibles vues en cours.

(c) Réaliser le codage d'un prédicat liste_des_ancetres(-L,+X) qui réussit quand L est la liste de tous les ancêtres de X (on suppose que le second argument est instancié).

(d) Programmer la définition du prédicat frere(X,Y) qui réussit quand X est le frère de Y. Tester ce prédicat et s'intéresser à l'ensemble de toutes les solutions.

(e) On peut également écrire décrire d'autres relations comme les

prédicats cousin(X,Y) ou oncle_ou_tante(X,Y) qui réussissent respectivement quand X est le cousin de Y et quand X est l'oncle ou la tante de Y.

2. Manipulation de listes

(a) Réaliser votre codage des prédicats membre(X,L) et concat(L1,L2,L3) vus en cours et qui doivent réussir quand X est l'un des éléments de la liste L ou lorsque la liste L3 est la concaténation des listes L1 et L2. Réaliser les tests des différentes configurations d'appel. A nouveau, la trace de l'exécution du moteur Prolog est utile pour comprendre l'ordre dans lequel les réponses sont produites.

(b) Coder le prédicat dernier(X,L) qui réussit quand X est le dernier élément de la liste L.

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

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

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

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

(d) En exploitant un prédicat de concaténation, réaliser l'inversion d'une liste au moyen d'un prédicat inv\2 : inv(L1,L2) est vrai quand la liste L1 est la liste L2 inversée. On souhaite donc 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. Notons que le renversement de listes est possible avec le BIP reverse/2.

(e) On souhaite maintenant un prédicat composante(I,X,L) qui réussit quand le I-ème élément (en comptant à partir de 1) de la liste L a la valeur X. C'est le travail réalisé par le BIP nth1/3.

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

?-composante(I,a,[a,b,a]). doit réussir avec I=1 et I=3.

(f) Coder un prédicat subsAll(X,Y,L1,L2) qui réussit quand L2 est la liste

L1 dans laquelle les occurrences de X sont remplacées par des occurrences de Y . Ainsi la preuve de subsAll(a,x,[a,b,a,c],R) doit réussir avec R=[x,b,x,c]. Modifier la définition de ce prédicat pour qu'il soit reprouvable en ne faisant à chaque fois qu'une substitution.

(g) Nous pouvons maintenant considérer la manipulation d'ensembles (définis en extension) représentés par des listes (sans répétition) d'objets. Coder un prédicat list2ens(L1,L2) qui réussit si la liste L2 est la représentation d'un ensemble (pas de répétitions) contenant les éléments de la liste L1. Ainsi la preuve de list2ens([a,b,c,b,a],E) doit réussir avec E=[a,b,c]. On s'interdira ici l'utilisation du méta-prédicat « setof ».

(h) On peut ensuite décrire l'union, l'intersection, la différence, l'égalité de deux ensembles représentés par des listes. Quel aurait été l'avantage, dans ce dernier cas, d'utiliser sort/2 (tri selon « @< ») ?

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).