

Prolog TP n° 1

Université Paris I. Maîtrise de Logique. Année 2004-2005.
Nicolas Stroppa - stroppa@enst.fr

1 Introduction

Le langage Prolog (PROgrammation LOGique) est le représentant du paradigme de programmation appelé (!) *programmation logique*. Par opposition à des langages qualifiés d'*impératifs* ou *procéduraux* (C, Pascal, ...), Prolog est qualifié de *déclaratif*.

Dans Prolog, il s'agira donc de « déclarer » des faits et les conditions pour lesquelles ces faits sont vérifiés. D'un point de vue logique, Prolog permet de définir des prédicats, des termes et des variables. La structure de liste est une forme particulière de terme très souvent utilisée pour les objets structurés.

L'objectif du TP est de se familiariser avec la nature déclarative de Prolog, et de manipuler la structure de liste.

2 Nature déclarative de Prolog

- Définissez un certain nombre de *faits* (assertions). Par ex:
'Il fait beau', 'Je suis en vacances', 'Je suis content'..
(Ces faits sont des prédicats d'arité 0.)
- Établissez des relations logiques (règles) entre ces faits pour pouvoir faire de la déduction. Par ex.:
'Il va falloir travailler' :- 'C est la rentrée'.
- Essayez quelques *buts* (requêtes) et vérifiez que Prolog fait les bonnes déductions.
- **Remarque** : Essayer un but consiste à demander à Prolog si un fait est vrai - et si oui à quelles conditions. En d'autres termes, Prolog cherche à rendre le fait vrai à l'aide des assertions et des règles de déduction fournies. Si il n'y arrive pas, il considère que le fait recherché est faux.

3 Manipulations de listes

Les structures de listes sont supportées nativement dans le langage Prolog. Elles représentent un élément fondamental du langage. Elles sont notées entre crochets (ex.: $[a, b, c]$). La notation $[a, b|L]$ désigne la liste L augmentée en tête des éléments a et b . Par exemple $[a|[a, b]]$ est identique à $[a, a, b]$.

3.1 Prédicats « simples »

Dans cette partie, il s'agit de définir un certain nombre de prédicats simples sur les listes. Certains sont déjà présents dans les bibliothèques standards du langage, d'autres non. (On pourra se servir des prédicats présents, notés entre crochets, pour vérifier la validité des prédicats définis.)

Définir les prédicats suivants :

1. `prem(X, Y)`: Y est la tête de la liste X.
2. `rest(X, Y)`: Y est la queue de la liste X.
3. `der(X, Y)`: Y est le dernier élément de la liste X [last].
4. `nieme(N, X, Y)`: Y est le N-ième élément de la liste X.
5. `elem(Y, X)`: Y est un élément de la liste X.
6. `concat(X, Y, Z)`: Z est la concaténation de X et Y [append].
7. `long(X, N)`: N est longueur de la liste X [length].
8. Écrire `paire` et `impaire` qui permettent de tester la parité de la longueur d'une liste.

3.2 Prolog et la réversibilité

1. Écrire le prédicat `extraire(X, Y, E)`: Y est la liste X dont on a extrait l'élément E.
2. Essayer les buts : `extraire([3], [], 3)`.
`extraire(L, [1,2,3,2,1], 2)`.
`extraire([1,2,3,4,5], [1,2,4,5], X)`.
`extraire([1,2,3,4,5], L, X)`.
`extraire([1,2,4,5], L, 3)`.
3. Commentez.
4. Utiliser `concat` pour écrire `sous_liste` :
`sous_liste(ListeIncluse, ListeContenant) :- ...`
5. Écrire `palindrome` qui permet de tester si une liste est un palindrome. On pourra penser à utiliser un accumulateur, c'est-à-dire une liste dans laquelle on stocke les éléments lus au moment de l'appel récursif.

3.3 Les listes : des structures trop simples ?

En réalité, les listes en Prolog permettent de représenter beaucoup plus d'objets que ce qu'on entend par « liste » dans le langage courant. En effet, les listes en Prolog ne sont pas « typées », i.e. elles peuvent contenir des éléments hétérogènes. En particulier, il est possible de construire des listes qui contiennent d'autres listes. Par exemple, la liste `[a, [a, b, c], [[d, e], f], g, [h, i]]` est totalement valide.

Une conséquence intéressante est qu'on peut représenter simplement des arbres (ex : [tete, fils1, fils2, ... , filsn]). Dans la suite, nous travaillerons uniquement sur des arbres binaires dont les nœuds ont soit deux fils (appelés respectivement fils gauche et fils droit), soit aucun fils (dans ce cas, le nœud est appelé feuille).

1. Écrire un prédicat qui vérifie qu'une liste est un arbre binaire.
2. Écrire un prédicat qui calcule la profondeur d'un arbre binaire.
3. Écrire un prédicat qui calcule le nombre de nœud d'un arbre binaire.

4 Éléments de corrigé

```
%%% - partie 2 - %%%

'Il fait beau'.
'Je suis en vacances'.
'C est la rentree'.

'Il va falloir travailler' :- 'C est la rentree'.
'Je suis content' :- 'Il fait beau'.
'Je suis content' :- 'Je suis en vacances'.

%%% - partie 3.1 - %%%
prem([H|_], H).

rest([_|T], T).

der([E], E).
der([_|T], Z) :-
    der(T, Z).

nieme(1, [H|_], H).
nieme(N, [_|T], Z) :-
    N > 0,
    N1 is N - 1,
    nieme(N1, T, Z).

elem(E, [E|_]).
elem(E, [_|T]) :-
    elem(E, T).

concat1([], X, X).
concat1([H|T], Y, [H|Z]) :-
    concat1(T, Y, Z).

long([], 0).
long([_|T], N) :-
    long(T, N1),
    N is N1 + 1.
```

```

paire([]).
paire([_|T]) :-
   impaire(T).
impaire([]).
impaire([_|T]) :-
   paire(T).

%% - partie 3.2 - %%
extraire(X, [X|L], L).
extraire(X, [Y|L1],[Y|L2]) :-
   extraire(X,L1,L2).

sous_liste(L1, L) :-
   concat1(_, L1, L2),
   concat1(L2, _, L).

miroir(X, Y) :-
   transfere(X, [], Y).

transfere([], L, L).
transfere([H|T], L1, L2) :-
   transfere(T, [H|L1], L2).

palindrome(L) :-
   miroir(L, L).

%% - partie 3.3 - %%
est_arbre([X]) :-
   atom(X).

est_arbre([X, Y, Z]) :-
   atom(X),
   est_arbre(Y),
   est_arbre(Z).

nombre_noeuds([], 1).

nombre_noeuds([_, Y, Z], N) :-
   nombre_noeuds(Y, N1),

```

```
nombre_noeuds(Z, N2),  
N is N1 + N2 + 1.
```

```
prof([_], 0).  
prof([_, Y, Z], N) :-  
    prof(Y, N1),  
    prof(Z, N2),  
    N is max(N1, N2) + 1.
```