Prolog : Compte-rendu de TP

Baptiste Rébillard

15 décembre 2024

Table des matières

1 TP1 : les bases

1.1 Partie 1

1. -

2. Dans la partie **personne(Pere, homme, Age, Profession, Ville)**, les variables Age, Profession, et Ville sont mentionnées mais jamais utilisées dans la règle. Il faudrait les remplacer par des '_'. Cette solution est utilisé dans la définition de **fille/2**. Les '_' correspondent a un élément qui est ignoré.

```
pere(Pere, Enfant) :-
    personne(Pere, homme, _, _, _),
    parent(Pere, Enfant).
```

```
?- personne(X,_,_,_,nice). % les personnes vivants à Nice
```

-> Edouart, Agathe, Octave

```
habite_avec_parents(Personne) :-
    personne(Personne,_,_,_,Ville),
    parent(Pere,Personne),
    parent(Mere, Personne),
    personne(Mere,_,_,_,Ville),
    personne(Pere,_,_,_,Ville).

% les personnes qui habitent dans la même ville que leurs deux parents
?- habite_avec_parents(X).
```

-> Octave, Elton, Vanessa

3. On implémente ceci de cette manière :

```
mere(Mere, Enfant) :-
    personne(Mere, femme, _, _, _),
    parent(Mere, Enfant).

fils(Fils, Parent) :-
    personne(Fils, homme, _, _, _),
    parent(Parent, Fils).
```

4. Cette requête retourne E et P tel que E est un écolier et P le parent correspondant. En d'autres termes, on cherche les parents d'écolier.

```
?- parent(P,E), personne(E,_,_,ecolier,_).
```

5. Définissons **parent_ecolier** :

```
parent_ecolier(P) :-
    personne(P,_,_,_,_),
    once((parent(P,E), personne(E,_,_,ecolier,_))).

?- parent_ecolier(X)
```

1.2 Partie 2

1. Il faut faire un OU logique (';') sur 2 types de plats :

```
plat(P) :-
    (viande(P);poisson(P)).

repas_leger([Entree, Plat, Dessert, Boisson]) :-
    hors_d_oeuvre(Entree),
    plat(Plat),
    dessert(Dessert),
    boisson(Boisson),
    calories(Entree, C1),
    calories(Plat, C2),
    calories(Dessert, C3),
    calories(Boisson, C4),
    (C1+C2+C3+C4 < 600).
```

2. Il faut simplement redefinir la même règle mais en créant une variable "C" à l'aide du mot clé **is** ainsi que changer le 600 par une variable **Limite**.

```
repas_leger_v2([Entree, Plat, Dessert, Boisson], Cal, Limite) :-
    hors_d_oeuvre(Entree),
    plat(Plat),
    dessert(Dessert),
    boisson(Boisson),
    calories(Entree, C1),
    calories(Plat, C2),
    calories(Dessert, C3),
    calories(Boisson, C4),
    (Cal is C1+C2+C3+C4),
    (Cal < Limite).
```

Mais alors existe-t-il des repas de moins de 550 calories ?

```
?- repas_leger_v2(_,_,550)
```

-> renvoie **true** donc ça existe bien.

1.3 Partie 3 : Récursions arithmétiques

1.3.1 Factorielle

1. On précise d'abord que $0! = 1$ afin que prolog sache que factoriel de 0 est 1 avant d'executer l'algo fact(N,F) (qui ne va pas fonctionner pour cette valeur).

2. On peut simplement retirer le coupe-choix('**!**'). De plus, on ajoute la condition $N > 0$ pour éviter d'évaluer des nombres négatifs.

```
fact(0,1).
fact(N,F) :-
    N>0,
    N1 is N-1,
    fact(N1,F1),
    F is N*F1.
```

3. Pour ce qui est des inconvénients : la version simplement avec coupe-choix va planter lorsque N<0, mais va fonctionner. La version sans coupe-choix va planter lorsqu'on cherche une seconde solution.
Pour ce qui est des avantages : la version avec coupe-choix ne va jamais évaluer un $N < 0$, la version sans coupe-choix ne va pas chercher plusieurs solutions.

1.3.2 Somme d'une liste d'entiers

1. On peut définir en 2 clauses le prédicat **somme** de la manière suivante :

```
somme([], 0).
somme([X|Rest], S) :-
    somme(Rest,S1),
    S is X+S1.
```

2. Cette version du programme utilise cette fois-ci un offset/accumulateur (I) :

```
somme2([1,2,3,4], 0, S).    % S = 10
somme2([1,2,3,4], 0, 9).    % false
somme2([1,2,3,4], 123, S). % S = 133
somme2([1,2,3,4], 0, 10).   % true
```

3.

1.3.3 Suite de Fibonacci

1. On peut définir cette fonction de la manière suivante :

```
fibo(0, 0).
fibo(1, 1).
fibo(X, Y) :-
    X > 1,
    X1 is X - 1,
    X2 is X - 2,
    fibo(X1, Y1),
    fibo(X2, Y2),
    Y is Y1 + Y2.
```

Ce programme ne réussit pas pour fibo(30,X) car la complexité de l'algo est grande.

2. Il suffit de faire une double itération de la manière suivante :

```
fiboplus(0, 0, 1).

fiboplus(N, F, F1) :-
    N > 0,
    N1 is N - 1,
    fiboplus(N1, F1, F2),
    F is F1 + F2.
```

fiboplus(56000, X, _) renvoie Trapped tripwire max_integer_size : big integers and rationals are limited to 0 bytes car le nombre que renvoie fiboplus de 560000 est beaucoup trop grand est n'est pas stockage par le processus.

2 TP2 : Recursions sur les listes

2.1 Partie 1 : Palindrome

1. On code palindrome de cette manière :

```
palindrome(P) :-
    reverse(P, P).
```

Une autre version :

```
palindrome2([]).
palindrome2([_]).

palindrome2([X|Y]) :-
    append(B,[X], Y),
    palindrome2(B).
```

-> Celle ci consiste a couper le début et la fin de la liste et récursivement jusqu'à arriver à une liste de taille 0 ou 1.

2.2 Partie 2 : Dispatch et QuickSort

1. On défini dispatch de la manière suivante :

```
dispatch(_, [], [], []).
dispatch(Comp, [Y|Rest], [Y|Infeg], Sup) :-
    Y=<Comp,
    dispatch(Comp, Rest, Infeg, Sup).
dispatch(Comp, [Y|Rest], Infeg, [Y|Sup]) :-
    Y>Comp,
    dispatch(Comp, Rest, Infeg, Sup).
```

2. On peut définir quicksort de la manière suivante :

```
quicksort([X], [X]).
quicksort([], []).
quicksort([X|Rest], Out) :-
```

```
    dispatch(X, Rest, L1, L2),
    quicksort(L1, Out1),
    quicksort(L2, Out2),
    append(Out1,[X|Out2],Out).
```

3. De la manière suivante :

```
quicksort2([], Acu, Acu).
quicksort2([X|Rest], Acu, Out) :-
    dispatch(X, Rest, L1, L2),
    quicksort2(L2, Acu, OutSup),
    quicksort2(L1, [X|OutSup], Out).
```

Quel est l'avantage ? le append fait des ajout en fin de liste : plus grande complexisté comparé a quicksort2 qui fait des ajout en début de liste.

2.3 Décomposition d'une liste

1. Comme ceci :

```
a_droite(X, L, D) :-
    append(_,[X|D],L).

a_gauche(X, L, G) :-
    append(G,[X|_],L).
```

2. Comme ceci :

```
separer(X, L, G, D) :-
    append(G,[X|D],L).
```

2.4 Chemins dans un graphe fini orienté

1. Il suffit d'utiliser append pour vérifier qu'une occurence est dans la liste(les "_" corresponde à partie gauche et droite de la liste autour de l'élement recherché) :

```
arc(G, O, E) :-
    graphe(G, _, L),
    append(_, [[O, E]|_], L).
```

2. On peut essayer d'imaginer cette connerie :

```
existe_chemin(G, O, E) :-
    arc(G, O, E) ; ( arc(G, O, I), existe_chemin(G, I, E) ).
```

3. Pour stocker au fur et a mesure l'itinéraire :

```
itineraire(Q, O, E, [O, E]) :-
    arc(Q, O, E). % Cas de base : route directe entre O et E.

itineraire(Q, O, E, [O | Rest]) :-
```

```
    arc(Q, O, I),          % Trouver un point intermédiaire I.
    I \= E,                % Éviter les boucles directes.
    itineraire(Q, I, E, Rest).
```

4. -

5. Il existe une infinité de solutions et ça boucle à l'infini

6. On défini un path finder qui ne boucle pas de cette manière :

```
chemin_sans_circuit(G, O, O, [O], _). % départ = arrivée.
chemin_sans_circuit(G, O, E, [O | Rest], Memo) :-
    arc(G, O, I),
    \+ member(I, Memo),     % Vérifier que I n'a pas déjà été visité.
    chemin_sans_circuit(G, I, E, Rest, [I | Memo]).
```

et on l'appelle comme ça : chemin_sans_circuit(g2, 1, 6, Chemin, []).

3 TP 3 et 4 : résolution de casse tête

3.1 Modélisation du problème

1. On met chaque clause de cette manière :

```
disque(a,[-1, 0, 0, 0, 0, 0]).
disque(b,[-1, -1, 0, 0, 0, 0]).
disque(c,[-1, 0, -1, 0, 0, 0]).
disque(d,[-1, 0, 0, -1, 0, 0]).
disque(e,[-1, -1, -1, 0, 0, 0]).
disque(f,[-1, -1, 0, -1, 0, 0]).
disque(g,[-1, 0, -1, 0, -1, 0]).
disque(h,[-1, -1, -1, -1, 0, 0]).
disque(i,[-1, -1, -1, 0, -1, 0]).
disque(j,[-1, -1, 0,-1, -1, 0]).
disque(k,[-1, -1, -1, -1, -1, 0]).
disque(l,[-1, -1, -1, -1, -1, -1]).
```

2. On hardcode également :

```
liste_des_disques([a,b,c,d,e,f,g,h,i,j,k,l]).
```

3. On peut tourner par pattern matching :

```
rotation_droite([A,B,C,D,E,F],[F,A,B,C,D,E]).
```

On peut également orienter :

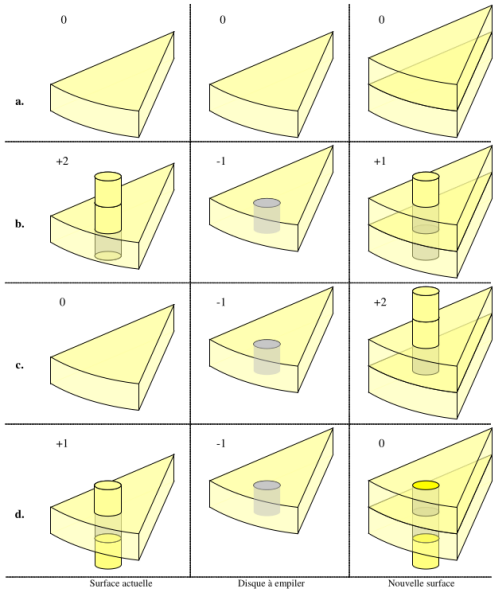
```
orienter_mem(M1,M1,0, _).
orienter_mem(M1,M2,N, Mem) :-
    rotation_droite(M1, INT),
    Mem \= INT,
    orienter_mem(INT, M2, NewN, Mem),
    N is NewN+1.

orienter(M1, M2, N) :- orienter_mem(M1, M2, N, M1).
```

3.2 Empilement des disques

Il faut décrire les états possibles

```
empilement_secteur(0,0,0).
empilement_secteur(2,-1,1).
empilement_secteur(0,-1,2).
empilement_secteur(1,-1,0).
```

Ci-dessous, les 4 situations possibles (a,b,c,d) décrivant l'empilement d'un des 6 secteurs angulaires d'un disque sur une surface.




```
empilement([S1],[M],[S2]) :-
    empilement_secteur(S1, M, S2).

empilement([S1|RestS1],[M|RestM],[S2|RestS2]) :-
    empilement_secteur(S1, M, S2),
    empilement(RestS1, RestM, RestS2).
```

3.3 Resolution du casse-tête

```
etat_initial([0,0,0,0,0,0], L) :- liste_des_disques(L).
etat_final([0,0,0,0,0,0],[]).

arc([S1,L1],[S2,L2],[D,N]) :-
    append(PG, [D|PD], L1),
    append(PG, PD, L2),
    disque(D, DList),
    orienter(DList, DResult, N),
    empilement(S1,DResult, S2).

action(Edepart, Edepart, []).

action(Edepart, Earrive, [A|LRest]) :-
    arc(Edepart, EInt, A),
    action(EInt, Earrive, LRest).

solution(S) :-
    etat_initial(A,B),
    etat_final(C,D),
    action([A,B], [C,D], S).
```

Faut arriver a 9030 solutions à la fin