Université Paris Descartes
UFR de Mathématiques et Informatique
Master 1 Informatique

Projet de Programmation Logique

Sujet

Implémentation du jeu Des Chiffres et Des Lettres en prolog

Réalisé par:

BELAHCEL Wacim	21911555
BOURAÏ Assia	21907038

Chargé de TP : Elise Bonzon

2019-2020

Table des matières

Table des figures

Partie 1 : Chiffres

Le but de la partie 1 est de trouver une combinaison d'opérations sur des chiffres (sous forme d'une liste LC, exemple : LC = [[1 + 2] * 3]) qui donnera comme résultat un nombre donné D (exemple, D = 9), tous les chiffres de la combinaison d'opérations doivent être membre d'une liste L imposée par l'utilisateur (dans l'exemple précédent, L pourrait être égal à [1,2,3,3,5]), chaque élément de la liste L doit être utilisé qu'une seule fois.

```
?- chiffres([1,2,3,3,5], 9, D).
D = [9, [[1, +, 2], *, 3]] 
```

Figure 1- Résultat final de l'exécution du programme chiffres.pl

Nous allons décrire chaque prédicat utilisé afin d'arriver au résultat final (Figure 1) dans la suite de cette section.

Les Prédicats :

Prédicat chiffres :

```
64 chiffres(L, S,[S, LC]) :-
65       sousliste(L1, L),
66       calcule(L1, [S, LC]).
```

Figure 2 – Prédicat chiffres

Le prédicat chiffre est assez simple, il est composé de 3 arguments :

- L : La liste contenant les chiffres qui seront utilisés lors de la combinaison d'opérations.
- S : Le résultat final que nous cherchons à atteindre via la combinaison d'opérations.
- D = [S, LC] : une liste composée de 2 éléments, S qui doit être égale au résultat final à atteindre, et LC représentant la combinaison d'opérations afin d'atteindre S.

Nous souhaitons atteindre toutes les combinaisons possibles afin d'arriver au résultat S, c'est-à-dire, toutes les combinaisons d'opérations à partir des éléments de L ainsi que des éléments des sous listes de L qui permettraient d'atteindre S.

Pour cela nous faisons appelle au prédicat « sousliste » (ligne 65) que nous décrirons plus en détail et qui nous renverra toutes les sous listes possibles à partir de L dans L1.

La sous liste est ensuite envoyer au prédicat calcule(L1, [S, LC]) qui est le cœur de notre solution et que nous décrirons ci-dessous, ce prédicat nous renverra toutes les solutions [S,LC] possible à partir de toutes les sous listes possibles de L.

Prédicat sousliste :

```
1 sousliste([], _).
2 sousliste([X|S], [X|L]) :-
3     sousliste(S, L).
4 sousliste([X|S], [_|L]) :-
5     sousliste([X|S], L).
6
```

Figure 3 – Prédicat sousliste

Le prédicat sous liste est composé de deux arguments : L1 : la sous liste et L2 : la liste. Ce prédicat sert à vérifier qu'une liste L1 est une sous liste de L2, ainsi, une réponse triviale serait notre cas d'arrêt : L1 est une liste vide, la liste vide étant une sous liste pour toutes les listes possible (représenté dans la ligne 1).

Ensuite, afin de vérifier qu'une liste L1 est une sous liste d'une liste L2, il faut vérifier que tout élément contenu dans L1 est aussi contenu dans L2, représenté ici par les lignes 2 et 4.

Prédicat calcule :

```
51
52  calcule([X], [X, X]) :- !.
53  calcule(L, [C, LC]):-
54      sousliste(SousListeA,L),
55      sousliste(SousListeB,L),
56      concatenation(SousListeA, SousListeB, SousListeTemp),
57         permutation(SousListeTemp, L),
58             calcule(SousListeA, [A, LA]),
59      calcule(SousListeB, [B, LB]),
60      associe([A, LA], [B, LB], [C, LC]).
```

Le prédicat calcule prend deux arguments cité précédemment :

L : le résultat à obtenir.

D = [C, LC] : une liste composée de 2 éléments comme cité précédemment.

Figure 4 – Prédicat calcule

Le Prédicat calcule est rappelé récursivement, le cas d'arrêt représenté dans la ligne 52 signifie que lorsque la liste des éléments L ne contient qu'un seul élément, il ne peut y avoir qu'un seul résultat possible est c'est l'élément lui-même.

Détaillons maintenant les lignes 53 à 60, ici nous pouvons voir que nous commençons tout d'abord par effectuer une liste d'opérations qui permettrait de diviser notre liste d'éléments initiale en deux sous listes, SousListeA et SousListeB (ligne 54 à 57) :

Ligne 54 et 55 : nous définissons nos deux sous listes comme étant bel et bien des sous listes de L en appelant notre prédicat sousliste deux fois.

Ligne 56 et 57 : il faut maintenant vérifier que l'union de nos liste SousListeA et SousListeB donnerait L, pour cela nous utilisons concaténation (prédicat qui permet de concaténer comme append à l'exception qu'il fail si l'une des sous listes est vide) mais aussi permutation (prédicat qui permet de vérifier qu'une liste est une permutation d'une autre liste) car le résultat de la concaténation de nos 2 sous liste pourrait donner une permutation de L et non pas L lui-même.

Ensuite la ligne 58 et 59 permet de rappeler calcule récursivement sur les sous listes de L, et de renvoyer toutes les combinaisons de résultats possible à partir des ces résultats dans le 2éme argument du prédicat.

Nous finissons par appeler le prédicat associe (ligne 60) ou A, B et C sont des chiffres ; LA, LB et LC sont les formules ayant permis d'obtenir ces chiffres sous forme de listes.

Prédicat permutation :

```
 9  supprimer_element(X, [X|L1], L1).
10  supprimer_element(X,[Y|L1], [Y|L2]):-
11      supprimer_element(X,L1, L2).
12
13  permutation([], []).
14  permutation(L, [X|L1]) :-
15      supprimer_element(X,L, L2),
16      permutation(L2, L1).
17
```

Le prédicat permutation permet de vérifier qu'une liste est une permutation d'une autre liste. Elle est composée de deux arguments :

- Argument 1 : la liste.
- Argument 2 : la permutation de la liste à vérifier.

Figure 5 – Prédicat permutation

Pour cela nous utilisons comme cas d'arrêt le cas de la liste vide qui est seulement une permutation d'elle-même, ensuite pour le reste des cas, il suffit de supprimer récursivement un élément X commun aux deux liste j'jusqu'à atteindre le cas d'arrêt.

Prédicat chaine :

```
26  chaine( X, Y, Op, Full):-
27      concatenation([X], Op, Half),
28      concatenation(Half, [Y], Full).
```

Le prédicat chaine est vrai si Full est la liste provenant de la concaténation de X, Y et Op.

Figure 6 – Prédicat chaine

Il est utilisé dans le prédicat associe afin de concaténer les opérations.

Prédicat associe :

```
31  associe([A, LA], [B, LB], [C, LC]):-
32      C is A + B,
33      chaine(LA, LB, [+], LC).
34
35  associe([A, LA], [B, LB], [C, LC]):-
36      C is A * B,
37      chaine(LA, LB, [*], LC).
38
39  associe([A, LA], [B, LB], [C, LC]):-
40      C is A-B,
41      chaine(LA, LB, [-], LC).
42
43  associe([A, LA], [B, LB], [C, LC]):-
44      B =\= 0,
45      0 =:= A mod B,
46      C is A / B,
47      chaine(LA, LB, [/], LC).
```

Où A, B et C sont des chiffres ; LA, LB et LC sont les formules ayant permis d'obtenir ces chiffres sous forme de listes. Quatre cas sont à considérer

- Si C=A+B, alors LC = [LA, +, LB]
- Si C=A−B, alors LC = [LA, −, LB]
- Si C=A⋆B, alors LC = [LA, ⋆, LB]
- Si C=A/B, alors LC = [LA, /, LB]

Figure 7 – Prédicat associe

Partie 2 : Lettres

La partie 2 a pour but d'obtenir le mot le plus long possible (récupéré à partir d'un dictionnaire) pouvant être formé à partir de 9 lettres données sous forme d'une liste L, nous récupérons aussi la taille de ce mot (nombre de lettres).

La difficulté dans cette partie est principalement d'obtenir un résultat rapidement, nous avons donc essayé plusieurs méthodes avant d'arriver au résultat final, parmi les différentes solutions testées (ordonnées de la moins rapide à la plus rapide),

- Tester chaque permutation de chaque sous liste sur tous les mots directement en listant le fichier.
- Charger les mots dans une liste au préalable et tester chaque résultat.
- Tester chaque mot sur toutes les permutations possibles directement en lecture (en ajoutant des conditions telle que ne pas tester un mot dépassant la taille de L, ne pas tester un mot plus petit que le plus grand mot déjà trouvé …).
- Charger les mots dans un « fait » ou « fact » et tester chaque permutation de chaque sous liste de L puis récupérer les le plus long.
- **Charger les mots dans un « fact » puis tester chaque permutation de la liste initiale sur notre fact en commençant par les plus longues permutations possibles puis décrémenter peu à peu et s'arrêter dès qu'un résultat est trouvé.**

En gras la description de la solution finale, on peut voir de l'énumération ci-dessus la logique du résonnement qui a été entrepris et les différents obstacles que nous avons dû surmonter durant cette partie afin de parvenir au résultat final. En effet, le dictionnaire étant composé de 360 000 mots, il est compliqué de tester toutes les permutations possible de toutes les sous listes (il en existe environ 986 410 pour une liste de 9 éléments) sur tous les mots.

La solution la plus logique serait donc d'ordonner les permutations à tester afin de tester d'abord les plus grande, et de décrémenter ensuite, ce qui réduit drastiquement le temps avant une première réponse.

Le résultat final :

```
?- lettres([t,m,e,s,e,e,s,t,s],Mot, Taille, "C:/Users/user/Documents/Prolog/dictionnaire.txt").
Mot = testes,
Taille = 6.

?-
```

Figure 8 – Résultat final de l'exécution du programme lettres .pl

Il est a noté que cette solution n'est pas optimale et qu'il y'a encore beaucoup de choses que l'on pourrait tester et améliorer (une liste supérieur à 11 devient beaucoup trop lente à tester), on pourrait par exemple ordonner nos mots dans le dictionnaire du plus long au plus petit, et trouver le mot le plus long dont tout les éléments sont contenu dans L, cependant la solution trouvée est suffisante pour une liste à 9 éléments.

Dans la suite de cette section nous tacherons de décrire les prédicats utilisés afin de résoudre le problème.

Les Prédicats :

Prédicat lettres :

```
34  lettres(L2, M, Taille, File):-
35    dicbuilder(File),
36    sublist(L2,_, L, _),
37    permutation(L,L1),
38    atom_chars(M, L1),
39    dico(M, Taille), !.
40
```

Le prédicats lettre est composé de 4 arguments :

L2 : la liste des 9 lettres.

M : le Mot le plus long trouvé.

Taille : la taille du mot.

File : le chemin complet du fichier dictionnaire.

Figure 9 – Prédicat lettres

Ce prédicat représente le cœur de la solution, il sert notamment à renvoyer le plus long mot et ce en utilisant plusieurs autres prédicats que nous définirons en détail plus bas.

Tout d'abord, à la ligne 35, nous faisons appel au prédicat dicbuilder(File) qui permet de charger l'ensemble des mots de notre fichier dans un Fact nommé « dico ».

A la ligne 36, nous utilisons un prédicat sublist qui permettra de générer toutes les sous listes possibles à partir de L2 en commençant par la plus grande sous liste possible (donc L2).

Nous utilisons ensuite le prédicat prédéfini permutation pour vérifier l'ensemble des permutations d'une sous liste donné.

A la ligne 38, on transforme la permutation de notre sous liste (L1) en un atom grâce au prédicat prédéfinit « atom_chars\2 ».

On finit par tester si notre atom M fait partie du fact « dico ».

Prédicat dicbuilder :

```
1  readWord(InStream,W, WCount):-
2     get_code(InStream,Char),
3     checkCharAndReadRest(Char,Chars,InStream, WCount, 0),
4     atom_codes(W,Chars).
5
6
7  checkCharAndReadRest(10,[],_, WCount, WCount):-!.
8
9  checkCharAndReadRest(32,[],_, WCount, WCount):-!.
10
11 checkCharAndReadRest(-1,[],_, WCount, WCount):-!.
12
13 checkCharAndReadRest(end_of_file,[],_, WCount , WCount):-!.
14
15 checkCharAndReadRest(Char,[Char|Chars],InStream,WCount,WCountemp):-
16    Temp is WCountemp + 1,
17    get_code(InStream,NextChar),
18    checkCharAndReadRest(NextChar,Chars,InStream, WCount, Temp).
19
20 dicbuilder(File):-
21    open(File,read,Str),
22    builddic(Str),
23    close(Str).
24
25 builddic(Stream):-
26    at_end_of_stream(Stream),!.
27
28 builddic(Stream):-
29     readWord(Stream, W, Count),
30     assert(dico(W,Count)),
31     builddic(Stream).
32
```

Le prédicat dicbuilder prend un seul argument, le chemin du fichier.

On commence d'abord par ouvrir le ficher en mode lecture (ligne 21), on passe ensuite le flux de lecture au prédicat builddic, qui s'occupera de charger nos mots, on finit par fermer le flux (ligne 23).

Le prédicat builddic lit chaque mot ligne par ligne en faisant appel au prédicat readword (ligne 29), et retourne le mot dans W, ainsi que la taille du mot dans Count.

Le fonctionnement de readword est assez basique et est complétement basé sur « working with Files » du tutoriel « learn Prolog now » fourni en référence.

Figure 10 – Prédicat dicbuilder

On utilise ensuite le prédicat prédéfini assert\1 qui permet de charger un « atom » dans un « fact » afin de charger notre mot ainsi que sa taille dans le fact « dico », on rappelle récursivement builddic et on s'arrête lorsque l'on arrive au cas d'arrêt «at_end_of_stream » (ligne 25) une foi arrivé à la fin du fichier.

Prédicat sublist :

```
43  sublist(L, [], L, N):- length(L, N).
44  sublist(L, S, R, Leng) :-
45   length(L, N),
46   between(1, N, M),
47   length(S, M),
48   append([A,S,B], L),
49    append(A,B,R), Leng is N - M.
```

Figure 11 –Prédicat sublist

Le prédicat sublist est composé de 4 arguments et fonctionne différemment du sous liste de la partie 1 :

- L : la taille de la liste initiale.
- S : sous liste de L.
- R : sous liste de L.
- Leng : taille de R.

La différence entre S et R est que S fera une recherche croissante allant de la plus petite sous liste à la plus grande, alors que R fait le contraire, il est évident que dans notre cas nous avons besoin d'utiliser R afin de récupérer les les sous liste les plus grands possibles et de décrémenter ensuite.

Ce prédicat est très inspiré d'une ressource que nous avons référencé sur stackoverflow, l'un des éléments les plus important étant le prédicat append(Ls, L)\2 (différent de append\3) qui permet de vérifier que L est la concaténation des liste contenu dans la liste Ls.

On commence par length(L,N), on récupère la taille de liste dans N.

between(1, N, M), M contiendra un chiffre entre 1 et N, en gros M va commencer à 1 et s'incrémenter à chaque appel.

length(S, M), on définit un liste S on ne connait pas encore le contenu de S à ce niveau, mais on sait que sa taille est M.

append([A, S, B], L), au premier appel demande S a une taille de 1 du coup S peut être n'importe quelle sous liste de L contenant 1 élément, et R1 et R2 contienne le reste.

append(R1,R2, R). vu que l'on veut renvoyer les plus grandes sous listes d'abord, on renvoie le reste R = L - S ou R = R1+ R2.

Références

https://stackoverflow.com/questions/34846414/prolog-find-sublists-in-order/34856424#34856424

http://lpn.swi-prolog.org/lpnpage.php?pageid=online

http://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlch12

https://www.swi-prolog.org/pldoc/

https://www.swi-prolog.org/pldoc/man?predicate=append/2

https://www.swi-prolog.org/pldoc/man?predicate=assert/1

Annexe

1. Prédicat sousliste

```
?- sousliste([9,1],[9,4,1,6,9]).
true .

?- sousliste([5,8],[5,2,1]).
false.

?- sousliste(L,[9,2,1]).
L = [] ;
L = [9] ;
L = [9, 2] ;
L = [9, 2, 1] ;
L = [9, 1] ;
L = [2] ;
L = [2, 1] ;
L = [1] ;
false.
```

2. Prédicat permutation

```
?- permutation([31,12,19,10],[12,10,31,19]).
true .

?- permutation([4,2],[2,6,1]).
false.
```

3. Prédicat concatenation

```
?- concatenation([2,5],[6,8],L).
L = [2, 5, 6, 8].

?- concatenation([],[2,5,6,8],L).
false.

?- concatenation([],[2,5,6,8],[2,5,6,8]).
false.
```

4. Prédicat chaine

```
?- chaine(3,4,[*],L).
L = [3, *, 4].

?- chaine([5,+,9],[8,-,2],[*],L).
L = [[5, +, 9], *, [8, -, 2]].
```

5. Prédicat associe

```
?- associe([12,[5,+,7]],[6,[2,+,4]],[C,LC]).
C = 18,
LC = [[5, +, 7], +, [2, +, 4]] ;
C = 72,
LC = [[5, +, 7], *, [2, +, 4]] ;
C = 6,
LC = [[5, +, 7], -, [2, +, 4]] ;
C = 2,
LC = [[5, +, 7], /, [2, +, 4]].
```

6. Prédicat calcule

```
?- calcule([2,3],D).
D = [5, [2, +, 3]] ;
D = [6, [2, *, 3]] ;
D = [-1, [2, -, 3]] ;
D = [5, [3, +, 2]] ;
D = [6, [3, *, 2]] ;
D = [1, [3, -, 2]] ;
false.
```

7. Prédicat chiffre

```
?- chiffres([2,3,25], 125, D).
D = [125, [[2, +, 3], *, 25]] ;
D = [125, [[3, +, 2], *, 25]] ;
D = [125, [25, *, [2, +, 3]]] ;
D = [125, [25, *, [3, +, 2]]] ;
false.
```

8. Prédicat dicbuilder

```
?- dicbuilder("/home/assia/Documents/PROGRAMMATION LOGIQUE/projet/dict
ionnaire.txt").
true.
```

9. Prédicat sublist

```
?- sublist([a,z,e,r,t,y,u,j,d,h,f,g,c,v,b,h],R,S,9).
R = [a, z, e, r, t, y, u],
S = [j, d, h, f, g, c, v, b, h] ;
R = [z, e, r, t, y, u, j],
S = [a, d, h, f, g, c, v, b, h] ;
R = [e, r, t, y, u, j, d],
S = [a, z, h, f, g, c, v, b, h] ;
R = [r, t, y, u, j, d, h],
S = [a, z, e, f, g, c, v, b, h] ;
R = [t, y, u, j, d, h, f],
S = [a, z, e, r, g, c, v, b, h] ;
R = [y, u, j, d, h, f, g],
S = [a, z, e, r, t, c, v, b, h] ;
R = [u, j, d, h, f, g, c],
S = [a, z, e, r, t, y, v, b, h] ;
R = [j, d, h, f, g, c, v],
S = [a, z, e, r, t, y, u, b, h] ;
R = [d, h, f, g, c, v, b],
S = [a, z, e, r, t, y, u, j, h] ;
R = [h, f, g, c, v, b, h],
S = [a, z, e, r, t, y, u, j, d] ;
false.
```

10. Prédicat lettres

```
?- lettres([a,e,r,g,x,r,e,n],Mot,Taille,"/home/assia/Documents/PROGRAM
MATION LOGIQUE/projet/dictionnaire.txt").
Mot = enrager,
Taille = 7.
```