[IA01]

Compte-rendu TP n°1

Pour le 6 octobre à 18h

Etienne Laviolette (GI01)
Jianghan LI (GI01)

Introduction :

Ce premier TP du semestre en IA01 nous a permis de nous familiariser avec le langage LISP et l'environnement de développement ACL. En effet ce langage syntaxiquement à part des différents langages de programmation que nous avons pu rencontrer précédemment nous a dérouté, tout du moins a début. Les manipulations des listes (que l'on avait jamais rencontré auparavant) ainsi que la manipulation d'une base de données furent des exercices très instructifs.

Exerice n°1 :

Il nous a été demandé ici de construire 6 fonctions différentes (à l'aide du DEFUN) afin de réaliser les actions suivantes :

- reverseA (arg1 arg2 arg3) : retourne la liste constituée des trois arguments dans l'ordre inverse.

- reverseB (L) : retourne la liste inversée de 1, 2 ou 3 éléments.

- reverseC (L) : retourne la liste constituée des éléments de L inversés.

- double (L) : retourne la liste constituée des éléments de L en doublant les atomes

- doublebis (L) : retourne la liste constituée des éléments de L en doublant les atomes à tous les niveaux de profondeur.

- monAppend (L M) : retourne la concaténation des deux listes L et M

- myEqual : retourne t ou nil selon l'égalité de ses deux arguments.

Afin de répondre à ces exigences nous avons imaginé les fonctions suivantes :

- (defun reverseA (x y z) (list z y x)) :

A l'aide d'une définition de fonction grâce à DEFUN nous avons remanié l'ordre des arguments passés afin de créer une liste reprenant ces arguments dans l'ordre inverse.

```
- (defun reverseB (L)
           (if (caddr L)   (list  (caddr L) (cadr L) (car L))
          (if (cadr L)  (list (cadr L) (car L))
          (car L)))) :
```

Grâce à deux structures IF nous avons tester si la liste passée en argument contenait 1, 2 ou 3 éléments. A l'issue de ces tests nous renvoyons la liste inversée correspondante.

```
- (defun R (L)
    (if (equal nil  L) () (append (R (cdr L))
    (list (car L)))))
```

Ici nous avons testé tout d'abord dans un IF si la liste passée en argument était vide ou non (condition de fin de l'appel récursif utilisé ci-après). Dans le cas où elle ne l'était pas l'usage de la fonction append associé à une structure récursive nous a été très utile. En effet nous créons une liste grâce à append à partir du premier éléments (car L) de la liste créée grâce à l'appel récursif : tous les appels imbriqués se font, puis on crée une liste contenant le premier élément de la liste crée à chaque appel récursif.

```
- (defun D (L)
       (if L
           (if (listp (car L))
                (append (list (car L)) (D (cdr L)))
                (append (list (car L)) (list (car L)) (D (cdr
L)))
       )))
```

Dans cette fonction D (pour Double), nous avons pris aussi le parti d'utiliser la fonction append à l'aide d'appels récursifs successifs. Le point important pour ne double que les atomes et non pas les listes se situe au niveau du second IF imbriqué : celui qui test si le premier élément de la liste est une liste ou non grâce à la fonction listp. Si l'élément en question est une liste nous n'appelons pas récursivement D sur le premier élément.

L'alternative à la fonction ci-dessus lors du codage a été celle-ci :

```
(defun D (L)
       (if L
           (if (and (listp (car L)) (not (equal (car L) nil)))
         (append (list (car L)) (D (cdr L)))
         (append (list (car L)) (list (car L)) (D (cdr L)))
       )))
```

```
-  (defun DD (L)
       (if L
           (if (listp (car L))
         (append (list (DD (car L))) (DD (cdr L)))
         (append (list (car L)) (list (car L)) (DD (cdr L)))
       )))
```

Pour le doubleBis nous avons repris la même ossature que le double normal ci-dessus. Seulement nous avons rajouté l'appel récursif de DD sur la premier élément de la liste imbriquée dans la liste passée en argument. En effet contraire à la fonction ci-dessus : lorsque que l'élément de la liste est lui même une liste nous rappelons DD. C'est ce qui nous permet le doublage au second niveau hiérarchique.

Voici notre alternative à cette fonction double BIS :

```
(defun DD (L)
       (if L
           (if (and (listp (car L)) (not (equal (car L) nil)))
         (append (list (DD (car L))) (DD (cdr L)))
         (append (list (car L)) (list (car L)) (DD (cdr L)))
       ))
```

```
- (defun A (x y)
       (if (equal x nil)
           y
           (cons (car x) (A (cdr x) y))
       ))
```

Pour ré-écrire une fonction équivalent à append nous avons d'abord testé si la list x était nul : cette condition sert à l'arrêt de l'appel récursif de A sur le cdr de x. En effet, tant que la liste x n'est pas vide on construit une liste avec le car de x en appelant A sur le cdr de x ainsi que y. De ce fait on construit une list grâce à cons contenant les éléments de x dans l'ordre (grâce aux appels de A) auxquels on concatène ceux de y au bout de la chaine d'appels.

```
- (defun E (x y)
       (cond
         ((eq x y)  T)
         ((and (listp x) (listp y) (eq (car x) (car y))) (E
(cdr x) (cdr y)))
         (T Nil)
       ))
```

Dans cette ré-écriture de la fonction equal (et non pas eq), nous passons deux arguments. Si ces sont égaux au (sens de eq) nous renvoyons T (true). C'est à dire que la fonction renvoie T si les objets sont égaux et ne sont pas des listes. En effet le equal est plus général que le eq. Le eq s'occupe de l'égalité entre pointeurs, entre objets LISP. Or les listes ne sont pas des objets LISP. C'est pourquoi dans cette ré-écriture de la fonction equal (qui elle, pour sa part s'occupe d'évaluer l'égalité entre les listes), nous appelons récursivement notre E sur les cdr de chaque liste passées en argument.
Ainsi nous voyons la différence entre equal et eq et nous nous occupons de ré-écrire la fonction equal qui teste l'égalité entre deux listes.

Exercice n°2 :

Noue devons ici créer une fonction de type lambda avec l'utilisation d'un mapcar qui va retourner la liste des éléments par paire de la liste fournie en argument.

Nous utilisons pour ce la la syntaxe suivante :

(defun liste-paire (L) (mapcar #'(lambda (x) (list x x)) L))

Le lambda permet la définition d'une fonction locale (qui ne pourra pas être réutilisée ensuite) que mapcar va appliquer à car puis cdr de la liste : c'est à dire à tous les éléments de la liste L. On crée ici une liste doublant les éléments de la liste passée en argument.

Exercice 3 : a-list

Ce troisième exercice nous fait utiliser une ébauche de base de données en LISP.

On dispo d'une liste d'éléments associés deux à deux de la forme : ((clé1 valeur1) (clé2 valeur2) ... (clénk valeurk)). Notre but est d'écrire une fonction à avec comme argument une clé et la base et qui nous retourne la paire correspondante. Si la clé ne correspond à aucune autre dans la liste, la fonction retourne nil.

Voici la définition de la fonction que nous proposons :

```
(defun my-assoc (cle a-liste)

        (if (equal nil a-liste ) 'nil

                (if (equal cle (first (first a-liste))) (first a-liste) (my-assoc cle
                (rest   a-liste)))

        )

)
```

Nous parcourons la liste a-liste jusqu'à ce que cle soit égal à la clé d'une des paires disponible dans la base de connaissances. La fonction regarde si cela est vrai pour la première paire, si non elle effectue un appel récursif sur la liste de connaissances privée du premier doublet. Et regarde donc si la condition est vraie pour la première paire du reste, c'est-à-dire la deuxième paire. Au cas où la recherche ne soit pas fructueuse, nous appelons my-assoc jusqu'à ce que la liste soit vide. A ce moment cela signifie qu'on a pas trouvé de doublet contenant la cle. On renvoit donc nil.

Exercice 4 :

Ici nous allons gérer en profondeur une base de connaissances.

On considère une base de connaissances sur des personnes définies par les propriétés suivantes : nom, prénom, ville, âge et nombre de livres possédés.

Soit la base BaseTest suivante : (

```
setq BaseTest '((Dupond Pierre Lyon 45 150)
        (Dupond Marie Nice 32 200)
        (Dupond Jacques Lyon 69 20)
        (Perrot Jacques Geneve 28 500)
        (Perrot Jean Nice 55 60)
        (Perrot Anna Grenoble 19 180)
))
```

A. Fonctions de services :

Nous avons utilisé le nth qui permet d'accéder à l'élément placé en position du premier argument de la liste passé en second argument.

```
(defun nom (personne) (nth 0 personne))
(defun prenom (personne)(nth 1 personne))
(defun ville (personne)(nth 2 personne))
(defun age (personne) (nth 3 personne))
(defun nombre-livres (personne) (nth 4 personne))
```

Ces 5 fonctions de service nous permettent de gérer une entrée de la base de données. Elles vont nous êtres très utiles ensuite pour gérer la base en entier (et non pas seulement les entrées).

B. Fonctions de manipulation de BDD :

1°) Fonction FB1

Méthode Itérative :	*Méthode Récursive*
(defun FB1 (BDD) (dolist (i BDD) (print i)))	(defun FB1-recursif (BDD) (when (not (null BDD)) (FB1-recursif (rest BDD))) (print (first BDD)))
Nous parcourons la base élément par élément grâce au dolist qui les stocke au fur et à mesure dans le i. Le print affiche le contenu de ces i, c'est à dire des éléments de la la base.	Avec un when , nous pouvons construire une fonction récursive qui nous amène au même résultat . En effet nous appelons FB1-recursif tant que la BDD n'est pas nulle. On affiche ensuite la pile

2°) Fonction FB2

Méthode Itérative :	*Méthode Récursive*
(defun FB2 (BDD) (dolist (i BDD) (when (equal 'perrot (nom i)) (print i))))	(defun FB2-recursif (BDD) (cond ((equal 'perrot (nom (first BDD))) (cons (first BDD) (FB2-recursif (rest BDD)))) ((null BDD) nil) (t (FB2-recursif (rest BDD)))))

Le dolist sur la BDD nous permet de stocker au fur et à mesure tous les éléments de la BDD dans le i. Puis on affiche les éléments de l'entrée ayant le nom de famille « Perrot » grâce à l'appel de la fonction nom définie ci-dessus	Ici le « cond » nous permet de tester si le nom correspond à Perrot. On construit une liste grâce au cons tout en appelant FB2-recursif sur le rest e de la BDD pour ajouter à la liste les éléments suivants si le nom coincide toujours. Si ça n'est pas le cas, rien ne se passe, et on continue à appeler FB2-recursif avec le reste de la base jusqu'à ce que la liste soit nulle. On sort à ce moment de la boucle.

NOTE : La structure récursive utilisée dans FB2-récursive sera ré-utilisée selon le même schéma dans les FB3, FB4 et FB5 récursif avec quelques adaptations selon les spécifications.

3°) Fonction FB3 :

Méthode Itérative :	*Méthode Récursive*
(defun FB3 (name BDD) (dolist (i BDD) (when (equal name(nom i)) (print i))))	(defun FB3-recursif (name BDD) (cond ((equal name (nom (first BDD))) (cons (first BDD) (FB3-recursif name(rest BDD)))) ((null BDD) nil) (t (FB3-recursif name (rest BDD)))))
La fonction FB3 ressemble à la FB2 puisque l'idée générale reste la même. Seul change le fait que c'est l'utilisateur qui passe en second argument le nom qui l'intéresse	On a adapté ici FB2-recursif en remplaçant le nom qui était spécifié en FB2 par un argument, la structure reste la même pour le reste.

4°) Fonction FB4 :

Méthode Itérative :	*Méthode Récursive*
(defun FB4 (age-demand BDD) (dolist (i BDD) (when (equal age-demand (nom i)) (return i))))	(defun FB4-recursif (age-demand BDD) (cond ((equal name (age-demand(first BDD))) (cons (first BDD) (FB4-recursif age-demand(rest BDD)))) ((null BDD) nil) (t (FB4-recursif age-demand (rest BDD)))))
La fonction FB4 ressemble à la FB2 ainsi qu'à la FB3 puisque l'idée générale reste la même. Seul change le fait que c'est l'utilisateur qui passe en second argument l'âge qui l'intéresse. A noter qu'on utilise return et non print. On cherche seulement à afficher la première occurrence : return met fin à la boucle dès son appel.	On a adapté ici FB3-recursif en remplaçant le nom qui était spécifié en argument de FB3-recursif par l'âge, la structure reste la même pour le reste.

5°) Fonction FB5:

Méthode Itérative :	*Méthode Récursive*

(defun FB5 (BDD) (dolist (i BDD) (when (> 100 (nombre-livre i)) (return i)))	(defun FB5-recursif (BDD) (cond ((> 100(nombre-livres (first BDD))) (first BDD)) ((null BDD) nil) (t (FB5-recursif (rest BDD)))))
La fonction FB5 ressemble à la FB4. Ici on cherche juste la première personne de la base qui possède au moins 100 livres. L'utilisation du return est fait pour les mêmes raisons qu'en FB4.	On a adapté ici FB3-recursif en remplaçant le nom qui était spécifié en argument de FB3-recursif par l'âge, la structure reste la même pour le reste. On utilise la fonction nombre-livres définie en A.

6°) Fonction FB6:

Méthode Itérative :	*Méthode Récursive*
(defun FB6 (name BDD) (let ((somme 0)) (dolist (i BDD) (when (equal name (nom i)) (setq somme (+ somme (age i))))) (print (/ somme (length (FB3 name BDD))))))	

(return i)))	
Ici on introduit une variable locale somme grâce au let dans le defun de la fonction. On ajoute dans cette somme tous les âges des personnes dont le nom est passé en argument. On calcul ensuite la moyenne en divisant la valeur de somme par le nombre de personne portant le nom passé en argument grâce à la fonction FB3 défini plus haut. On affiche ce résultat dans un print	