TP1 : Montée en compétences Lisp

Alexis Schad – Matthieu Zins

## Exercice 1 : Mise en condition

reverseA (arg1 arg2 arg3) : retourne la liste constituée des trois arguments dans l’ordre inverse. Nous utilisons la fonction (list) pour créer la liste qui sera renvoyée.

(defun reverseA (a b c)

(list c b a))

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

En fonction de la taille de la liste passée en paramètre, on retourne la liste inversée. Si la liste ne contient qu'un seul atome on peut la renvoyer et si elle est de taille 2 ou 3 on crée la liste inverse en utilisant les fonctions **list**, **car** et **cdr**.

(defun reverseB (n)

(cond

((= (length n) 1) n )

((= (length n) 2) (list (cadr n) (car n)) )

((= (length n) 3) (list (caddr n) (cadr n) (car n)) )

)

)

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

Méthode itérative : On utilise l'opérateur let pour créer une liste vide. Puis, grâce à **dolist** et **push**, on insère chaque élément de la liste L au début de la liste n. A la fin cette liste n est retournée par la fonction.

(defun reverseC\_i (L)

(let ((n ()))

(dolist (x L n) (push x n))

)

)

Méthode récursive : La fonction **append** permet d'assembler le résultat de reverseC à laquelle on passe la liste L sans le premier élément et la liste composée uniquement du premier élément de la liste L. La récursivité va s'arrêter lorsqu'on passe une liste vide en paramètre de la fonction.

(defun reverseC\_r (L)

(when L

(append (reverseC\_r (cdr L)) (list (car L)))

)

)

Ces deux méthodes permettent d'inverser une liste sur le premier niveau, elles n'agissent pas en profondeur. Si la liste contient une sous-liste, celle-ci ne sera pas inversée.

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

Méthode itérative : Grâce à let, on crée une liste vide, puis, pour chaque élément de la liste L on l'ajoute une ou deux fois à la liste créée. Pour cela une utilise append qui va assembler la liste temp et une liste contenant une ou deux fois l'élément x. On affecte ensuite cette liste à temp. A la fin la liste temp est renvoyée.

(defun double\_i (L)

(let ((temp ()))

(dolist (x L temp)

(if (atom x)

(setq temp (append temp (list x x)))

(setq temp (append temp (list x)))))

Méthode récursive : Si la liste passée en argument n'est pas vide, la fonctions est rappelée avec en argument la liste L sans le premier élément. Lorsque l e premier élément est un atome on l'ajoute deux fois, sinon une seule fois.

(defun double\_r (L)

(when L

(if (atom (car L))

(append (list (car L) (car L)) (double\_r (cdr L)))

(append (list (car L)) (double\_r (cdr L)))

)

)

)

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

Pour cette fonction, seule une méthode récursive est envisageable parce qu'il faut traiter la liste en profondeur.

Si la liste L n'est pas vide et si le premier élément est un atome on le double et l'assemble au résultat de doublebis du reste de la liste.

Si le premier élément est une sous-liste on rappelle doublebis sur cette sous-liste et on assemble le résultat avec doublebis du reste de la liste.

(defun doublebis (L)

(when L

(if (atom (car L))

(append (list (car L) (car L)) (doublebis (cdr L)))

(append (list (doublebis (car L))) (doublebis (cdr L)))

)

)

)

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

Méthode itérative : On utilise deux **dolist** pour parcourir les deux listes passées en paramètre et on ajoute chaque élément au début de la liste temp. A la fin la liste concaténée est inversée donc on renvoie (reverse temp).

(defun monAppend\_i (L M)

(let ((temp ()))

(dolist (x L)

(setq temp (cons x temp))

)

(dolist (x M)

(setq temp (cons x temp))

)

(reverse temp)

)

)

Méthode récursive : Si la 1ère liste n'est pas vide on en construit une avec la fonction **cons** qui va assembler le premier élément de liste L et le résultat de monAppend\_r avec en paramètre le reste de L et M.

Lorsque L est vide on refait de même avec M.

(defun monAppend\_r (L M)

(if L

(cons (car L) (monAppend\_r (cdr L) M))

(when M

(cons (car M) (monAppend\_r L (cdr M)))

)

)

)

myEqual **:** retourne **t** ou **nil** selon l’égalité de ses deux arguments.

Dans cette fonction, nous testons, en premier, si les types des deux arguments est le même. Si non, on renvoie **nil**. Si oui, on utilise un **cond** pour tester si ce sont des atomes ou des listes. Dans le cas d'atomes on les compare avec **eq**. Dans le cas de liste, nous comparons récursivement les éléments des listes un par un.

(defun myEqual (a b)

(when (eq (type-of a) (type-of b))

(cond

((atom a)

(eq a b)

)

((listp a)

(and (myEqual (car a) (car b)) (myEqual (cdr a) (cdr b)))

)

)

)

)

## Exercice 2 : Objets fonctionnels

La fonction list-paire retourne la liste des éléments par paire de la liste fournie en paramètre. Pour cela nous utilisons **mapcar** qui va appliquer à chaque élément une fonction lambda qui crée une paire grâce à **(list x x)**.

(defun list-paire (L)

(mapcar #'(lambda (x) (list x x)) L))

## Exercice 3 : a-list

(my-assoc cle a-liste) retourne nil si cle ne correspond à aucune clé de la liste d'association, la paire correspondante dans le cas contraire; cette fonction existe sous le nom de assoc.

a-liste est une liste d'associations de la forme : ((clé1 valeur1) (clé2 valeur2) ... (clén valeurn)).

Nous utilisons un **dolist** sur le paramètre a-liste. Dès qu'une clé (qui correspond à **car** de x) est égale à celle recherchée on renvoie la paire grâce à **return**. Si a-liste ne contient pas la clé recherchée, on parcours entièrement la liste et **dolist** va automatiquement renvoyer **nil**.

(defun my-assoc (cle a-liste)

(dolist (x a-liste)

(if (equal (car x) cle)

(return x)

)

)

)

## Exercice 4 : gestion d’une base de conaissances en Lisp

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

### Les fonctions de service

nom (personne) : retourne le nom de la personne passée en argument  
prenom (personne) : retourne le prénom de la personne passée en argument  
ville (personne) : retourne la ville de la personne passée en argument  
age (personne) : retourne l’âge de la personne passée en argument  
nombre-livres (personne) : retourne le nombre de livres possédés par la personne passée en argument

Etant donné que **personne** et une liste, nous utilisons des combinaisons de **car** et de **cdr**. Pour **nombre-livres**, **caddddr** n’existe pas, nous utilisons donc **last**. Si il y avait eu plus de champs, nous aurions pu utiliser la fonction **nth**.

(defun nom (personne) (car personne))

(defun prenom (personne) (cadr personne))

(defun ville (personne) (caddr personne))

(defun age (personne) (cadddr personne))

(defun nombre-livres (personne) (car (last personne)))

### Les autres fonctions

**FB1 : affiche toutes les personnes**

On utilise ici un **dolist** pour parcourir la base de personnes. Pour chaque **personne x**, nous affichons toutes les informations que nous avons grâce à la fonction **format** et aux fonctions de service ci-dessus.

(defun fb1 (base)

(dolist (x base)

(format t "~S ~S ~S ~S ~S ~%" (nom x) (prenom x) (ville x) (age x) (nombre-livres x))

)

)

**FB2 : affiche les personnes qui s'appellent Perrot**

On fait exactement comme la fonction ci-dessus, sauf qu’on ajoute une condition sur le nom, pour n’afficher que les personnes s’appelant Perrot.

(defun fb2 (base)

(dolist (x base)

(if (equal 'Perrot (nom x))

(format t "~S ~S ~S ~S ~S ~%" (nom x) (prenom x) (ville x) (age x) (nombre-livres x)))))

**FB3 : retourne la liste des personnes dont le nom est précisé en argument**

Pour ne filtrer que les personnes dont le nom est précisé en argument, c’est exactement comme FB2, sauf qu’à la place de Perrot, on met l’argument de la fonction.

Ensuite, il faut retourner une liste. On l’initialise vide au début de la fonction, en variable locale avec **let**. Ensuite, dès qu’on a une personne à ajouter, on utilise **append** pour ajouter à la fin de la liste l’élément trouvé et on affecte la liste retournée par append à la liste finale.

On a précisé à la boucle **dolist** qu’on voulait retourner la liste à la fin, en mettant de nom de la liste en troisième paramètre.

(defun fb3 (base nom)

(let ((l '()))

(dolist (x base l)

(if (equal (nom x) nom)

(setf l (append (list x) l))

)

)

)

)

**FB4 : retourne la première personne qui a X ans (X = un argument) ou nil**

On parcourt là encore la base, et on s’arrête quand quelqu’un à un âge égal à l’âge passé en argument. On quitte la fonction directement grâce à return.

(defun fb4 (base x)

(dolist (p base nil)

(if (equal x (age p))

(return p)

)

)

)

**FB5 : retourne la première personne qui possède moins de 100 livres ou nil**

Exactement comme FB4, sauf que la condition d’arrêt est de posséder moins de 100 livres.

(defun fb5 (base)

(dolist (x base nil)

(if (< (nombre-livres x) 100)

(return x)

)

)

)

**FB6 : calcule et retourne la moyenne des âges des personnes de la famille X (X = un argument)**

On initialise en variable locale avec **let** les variables **somme-ages** et **somme-pers**, pour pouvoir faire la moyenne des âges.

Dans la boucle, on vérifie d’abord que la personne a pour nom l’argument passé à la fonction. Ensuite, on incrémente **somme-pers** et on ajoute à **somme-ages** l’âge de la personne.

Après la boucle, on teste si **nombre-pers** est égal à 0 ou pas. Si c’est le cas, il n’y a aucun personne de la famille donc on retourne **nil**. Sinon, on retourne **somme-ages/somme-pers**, qui est la moyenne des âges.

(defun fb6 (base x)

(let ((somme-ages 0) (somme-pers 0))

(dolist (p base)

(if (equal x (nom p))

(progn

(setf somme-ages (+ somme-ages (age p)))

(setf somme-pers (+ somme-pers 1))

)

)

)

(if (= somme-pers 0)

nil

(/ somme-ages somme-pers)

)

))