**IA01**

A14

**TP1 : Montée en compétences Lisp**

**Alexis Schad - Matthieu Zins**

Sommaire

[Exercice 1 : Mise en condition 3](#_Toc400233870)

[reverseA (arg1 arg2 arg3) : 3](#_Toc400233871)

[reverseB (L) : 3](#_Toc400233872)

[reverseC (L ) : 3](#_Toc400233873)

[Méthode itérative : 3](#_Toc400233874)

[Méthode récursive : 4](#_Toc400233875)

[double (L) : 4](#_Toc400233876)

[Méthode itérative : 4](#_Toc400233877)

[Méthode récursive : 4](#_Toc400233878)

[doublebis (L) : 5](#_Toc400233879)

[monAppend (L M) : 5](#_Toc400233880)

[Méthode itérative : 5](#_Toc400233881)

[Méthode récursive : 6](#_Toc400233882)

[myEqual : 6](#_Toc400233883)

[Exercice 2 : Objets fonctionnels 7](#_Toc400233884)

[list-paire : 7](#_Toc400233885)

[Exercice 3 : a-list 8](#_Toc400233886)

[my-assoc (cle a-liste) : 8](#_Toc400233887)

[Exercice 4 : gestion d'une base de connaissance en Lisp 8](#_Toc400233888)

[A. Fonctions de service : 8](#_Toc400233889)

[B. Fonctions de gestion de la base : 9](#_Toc400233890)

[**FB1** : affiche toutes les personnes de la base. 9](#_Toc400233891)

[**FB2** : affiche les personnes qui s'appellent Perrot. 10](#_Toc400233892)

[**FB3** : retourne la liste des personnes dont le nom est précisé en argument. 10](#_Toc400233893)

[**FB4** : retourne la première personne qui a X ans (X = un argument) ou nil. 11](#_Toc400233894)

[**FB5** : retourne la première personne qui possède moins de 100 livres ou nil. 11](#_Toc400233895)

[**FB6** : calcule et retourne la moyenne des âges des personnes de la famille X (X = un argument). 11](#_Toc400233896)

[Conclusion 12](#_Toc400233897)

# Exercice 1 : Mise en condition

## reverseA (arg1 arg2 arg3) :

Elle retourne la liste constituée des trois arguments dans l’ordre inverse. Nous utilisons la fonction **list** pour créer la liste en lui passant les trois arguments en sens inverse..

(defun reverseA (a b c)

(list c b a))

exemple *:*

>(reverseA 1 2 3)

(3 2 1)

## reverseB (L) :

Elle 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)) )

)

)

exemple :

> (reverseB '(1 2 3))

(3 2 1)

## reverseC (L ) :

Elle 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\_r 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.

exemple :

> (reverseC '(1 2 3 (4 5) 6))

(6 (4 5) 3 2 1)

## double (L) :

Elle 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 le 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)))

)

)

)

exemple :

> (double '(1 (2 3) 4))

(1 1 (2 3) 4 4)

Comparaison des deux méthodes :

Pour cette fonction, la méthode itérative semble plus adaptée. Le **dolist** permet de simplement parcourir la liste, alors que dans la méthode récursive la fonction est rappelée pour chaque élément de la liste, donc un grand nombre de fois si la liste est longue.

## doublebis (L) :

Elle 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 à tous les niveaux en profondeur. Par exemple, si la liste contient plusieurs couches de sous-listes.

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)))

)

)

)

exemple :

> (doublebis '(1 (2 3) 4))

(1 1 (2 2 3 3) 4 4)

## monAppend (L M) :

Elle 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, on renvoie donc (**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 première 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)))

)

)

)

exemple :

> (monAppend '(1 2) '(3 4))

(1 2 3 4)

Comparaison des deux méthodes :

On peut remarquer que la méthode itérative est plus efficace. Deux **dolist** permettent de concaténer chaque élément alors que dans la méthode récursive la fonction va être rappelée pour chaque élément des deux listes. Ce grand nombre d'appels de la fonction va la rendre moins rapide.

## myEqual :

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

Nous avons remarqué que la fonction **eq** permettait de tester l'égalité entre deux atomes. Si les deux arguments sont le même atome elle renvoie **t**. Par contre, si ce sont des atomes différents, des listes ou des strings, elle renvoie **nil**. Pour les listes, **eq** renvoie **nil**, même si elles sont identiques et de même pour les strings.

(eq '(1 2 3) '(1 2 3)) ==> nil

(eq "abc" "abc") ==> nil

Deux listes ou strings pouvant être considérées comme égales si leurs éléments sont égaux un par un, nous allons donc les comparer éléments par élément.

Nous avons également remarqué que la fonction equal permet de comparer des nombres mais seulement des nombres de même type.

(equal 1 1) ==> t

(equal 1.2 1.2) ==> t

par contre

(equal 1 1.0) ==> nil

Donc si les deux arguments sont des entiers tous les deux ou s'ils sont des réels tous les deux, on pourra les comparer avec =.

On utilise un **cond** pour différencier les différents cas :

* Si les deux arguments sont des string et s'ils sont de même taille. La fonction **char** dans un **dotimes** nous permet de comparer un par un les caractères des deux arguments.
* Si ce sont soit 2 entiers, soit 2 réels, on les compare avec =.
* Si ce sont des atomes, on utilise simplement **eq**.
* Si ce sont des listes, il suffit de tester l'égalité des deux premiers éléments avec **eq** et de répéter le processus sur les **cdr**.

Pour cette fonction une méthode simplement itérative ne serait pas envisageable car il faut pouvoir comparer chaque sous-liste de liste.

(defun myEqual (a b)

(cond

((and (stringp a) (stringp b) (eq (length a) (length b)))

(dotimes (x (length a) t)

(when (not (eq (char a x) (char b x)))

(return nil))

)

)

((or (and (integerp a) (integerp b)) (and (integerp a) (integerp b)))

(= a b)

)

((and (atom a) (atom b))

(eq a b)

)

((and (listp a) (listp b))

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

)

)

)

exemple :

> (myEqual '(1 (2 3) 4) '(1 (2 3) 4))

t

> (myEqual '1 '1)

t

> (myEqual "abc" "abc")

t

>(myEqual 1 1)

t

>(myEqual 1.2 1.2)

t

>(myEqual 1 1.0)

NIL

# Exercice 2 : Objets fonctionnels

## list-paire :

Elle 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)

)

exemple :

> (list-paire '(1 2 3))

((1 1) (2 2) (3 3))

# Exercice 3 : a-list

## my-assoc (cle a-liste) :

Elle retourne **nil** si cle ne correspond à aucune clé de la liste d'association et la paire correspondante dans le cas contraire. Cette fonction existe déjà sous le nom de **assoc**.

a-liste est une liste d'associations de la forme : ((clé1 valeur1) (clé2 valeur2) ... (clé-n valeur-n)).

Nous utilisons **dolist** pour parcourir 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 parcourt 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)

)

)

)

*exemple :*

> (my-assoc 'Pierre ((Yolande 25) (Pierre 22) (Julie 45)))

(PIERRE 22)

# Exercice 4 : gestion d'une base de connaissance en Lisp

Le but de cet exercice est de pouvoir manipuler une base de connaissance grâce à des fonctions de service. La base de connaissance contient des personnes définies par les propriétés suivantes : nom, prénom, ville, âge et nombre de livres possédés.

exemple de base :

(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 service :

Pour ces fonctions, nous utilisons des combinaisons des fonctions **car** et **cdr** pour accéder aux différentes propriétés des personnes. Pour le nombre-livres caddddr n'existe pas, nous utilisons donc le **car** de **last**. Si il y avait eu plus de champs, nous aurions pu utiliser la fonction **nth**.

**nom(personne)** : retourne le nom de la personne passée en argument.

(defun nom (personne) (car personne))

*exemple :*

>(nom ‘(Dupond Pierre Lyon 45 150))

DUPOND

**prenom (personne)** : retourne le prénom de la personne passée en argument.

(defun prenom (personne) (cadr personne))

*exemple :*

>(prenom ‘(Dupond Pierre Lyon 45 150))

PIERRE

**ville (personne)** : retourne la ville de la personne passée en argument.

(defun ville (personne) (caddr personne))

*exemple :*

>(ville ‘(Dupond Pierre Lyon 45 150))

LYON

**age (personne)** : retourne l’âge de la personne passée en argument.

(defun age (personne) (cadddr personne))

*exemple :*

>(age ‘(Dupond Pierre Lyon 45 150))

45

**nombre-livres (personne)** : retourne le nombre de livres possédés par la personne passée en argument.

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

*exemple :*

> (nombre-livres ‘(Dupond Pierre Lyon 45 150))

150

## B. Fonctions de gestion de la base :

Pour réaliser ces fonctions, nous utilisons les fonctions de service précédentes.

### **FB1** : affiche toutes les personnes de la base.

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**.

(defun FB1 (base)

(dolist (x base)

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

)

)

*exemple :*

>(FB1 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

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

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

(defun FB2 (base)

(dolist (x base)

(when (equal 'Perrot (nom x))

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

)

)

)

*exemple :*

>(FB2 BaseTest):

PERROT JACQUES GENEVE 28 500

PERROT JEAN NICE 55 60

PERROT ANNA GRENOBLE 19 180

### **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 n)

(let ((l ()))

(dolist (x base l)

(if (equal (nom x) n)

(setq l (append l (list x))))

)

)

)

*exemple :*

>(FB3 BaseTest 'Dupond)

((PERROT JACQUES GENEVE 28 500) (PERROT JEAN NICE 55 60) (PERROT ANNA GRENOBLE 19 180))

>(FB3 BaseTest 'Martin)

NIL

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

On parcourt là encore la base, et on s’arrête dès qu'une personne a un âge égal à l’âge passé en argument. On quitte la fonction directement grâce à return et renvoie cette personne. Sinon aucune personne n'a cet âge la fonction renverra **nil**.

(defun FB4 (base x)

(dolist (p base nil)

(if (equal x (age p))

(return p)

)

)

)

*exemple :*

>(FB4 BaseTest 32)

(DUPOND MARIE NICE 32 200)

>(FB4 BaseTest 33)

NIL

### **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)

)

)

)

exemple :

>(FB5 BaseTest)

(DUPOND JACQUES LYON 69 20)

### **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.

A la fin de 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 (= somme-pers 0) nil (/ somme-ages somme-pers)))

(if (equal x (nom p))

(progn

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

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

))

)

)

*exemple :*

>(FB6 BaseTest 'Dupond)

146/3

# Conclusion

Ce TP nous a permis une mise en pratique du langage Lisp et notamment la manipulation de listes et la recherche d'informations dans une base de connaissance. Il nous a également permis de nous familiariser avec le fonctions récursives.