IA01 : RAPPORT TP01

Exercice 1 : mise en condition

1. Fonction ReverseA :

Cette fonction permet de retourner la liste constituée des trois arguments dans l'ordre inverse.

Définition de la fonction : il suffit de retourner la liste avec les arguments dans l'ordre inverse.

```
(defun reverseA (arg1 arg2 arg3)
  (list arg3 arg2 arg1))
```

Exemple de résultat : on a bien une liste inversée.

```
CG-USER(6): (reverseA 1 2 3)
(3 2 1)
```

2. Fonction ReverseB :

La fonction qui retourne la liste inversée de 1,2 ou 3 éléments.

Définition de la fonction : on fait un test sur le nombre d'élément puis on renvoi la liste à l'envers. Si la liste est de longueur 1 on renvoi la liste.

```
(defun reverseB(L)
  (cond ((= (length L) 3) (reverseA (car L)(cadr L)(caddr L)))
        ((= (length L) 2) (list (cadr L) (car L)))
       (= (length L) 1) L))
  )
 )
```

Exemple de résultat :

```
CG-USER(12): (reverseB '(1 2))
(2 1)
```

3. Fonction monreverse :

ReverseC ou monreverse est une fonction permettant de retourner la liste inversée.

Comme une liste peut avoir un nombre n d'élément, j'ai opté pour la récursivité. A chaque fois qu'on fait un appel récursif, on ajoute le reste de la liste au premier élément. Le test sert à retourner la liste, si elle est de longueur 1.

```
(defun monreverse (L)
  (if (=(length L) 1)
        L
    (append (monReverse (cdr L)) (list (car L)))
    )
)
```

Exemple de résultat :

```
CG-USER(20): (monreverse '( a b (c d) e f))
(F E (C D) B A)
```

4. Fonction double

La fonction retourne une liste dont les atomes de la liste passée en argument ont été doublés. Une liste « buffer » est créée afin de copier les éléments de la liste et d'ajouter l'atome qui a été doublé à cette liste. On retourne l'inverse de la nouvelle liste pour qu'elle soit dans le bon sens.

```
(defun double(L)
  (setq newlist (list))
  (dolist (x L)
    (push x newlist)
    (if (atom x) (push x newlist)
      )
    )
    (reverse newlist)
)
```

Voici un test :

On voit que le résultat obtenu correspond à ce qu'on voulait.

```
(double '((1 2) 3 (4 5) 6))
```

```
((1 2) 3 3 (4 5) 6 6)
```

5. Fonction nombre3 :

-nombre3 : renvoie bravo si les trois premiers éléments de la liste sont des entiers, et perdu sinon.

Le test permet de voir si les trois premiers éléments sont des nombres c'est pourquoi on test le car, le cadr et le caddr de L. Si c'est le cas, on retourne bravo, sinon on retourne perdu.

```
(defun nombres3(L)
  (if (and (numberp(car L)) (numberp (cadr L)) (numberp (caddr L)))
      (print "BRAVO")
    (print "PERDU")
    )
  )
```

-Résultats : si les trois premiers éléments sont des nombres :

```
CG-USER(22): (nombres3 '( 1 2 3 R S 4))


"BRAVO"
"BRAVO"
```

Si les trois premiers éléments ne sont pas que des nombres :

```
CG-USER(23): (nombres3 '( 1 2 E R S 4))

"PERDU"
"PERDU"
```

6. Fonction grouper :

-fonction grouper : permet d'avoir une liste composée des éléments successifs de deux listes.

Fonction récursive, qui si les deux listes ne sont pas vides, concatène les éléments des listes successivement.

```
(defun grouper (L M)
  (if (and (not (null L)) (not (null M)))
      (CONS (list (car L) (car M)) (grouper (cdr L)(cdr M)))
    )
)
```

Résultat : on remarque que le résultat est satisfaisant.

```
CG-USER(60): (grouper '(1 2 3) '(4 5 6))
((1 4) (2 5) (3 6))
```

7. Fonction palindrome :

-fonction palindrome : renvoie vrai si la liste est un palindrome : on compare la liste L et la liste inversée de L en utilisant la fonction monreverse (définit plus tôt). Ainsi, si les deux listes sont identiques on renvoie True sinon NIL.

```
(defun palindrome(L)
  (if (EQUAL (monreverse L) L) (print"T") nil))
```

Quelques résultats : xamax est bien un palindrome, on retrouve bien T.

```
CG-USER(26): (palindrome '(x a m a x))

"T"
"T"
```

Xamas n'est pas un palindrome : on retrouve bien NIL.

```
CG-USER(27): (palindrome '(x a m a s))
NIL
```

Exercice 2 : Objets fonctionnels

Pour cet exercice nous allons avoir besoin de la fonction grouper définie dans l'exercice 1. La fonction permet de renvoyer les éléments de liste originale suivie de leur triple. La mapcar nous permet de tripler chacun des éléments de la liste L et la fonction grouper nous permet de regrouper les triples avec les éléments de la liste L.

```
(defun list-triple-couple (L)
    (grouper L (mapcar #'(lambda(L) (* L 3)) L))
  )
```

Résultat : on a bien un regroupement de l'élément et son triple.

```
CG-USER(62): (list-triple-couple '(0 2 3 11))
((0 0) (2 6) (3 9) (11 33))
```

Exercice 3 : A-list

1. Fonction myassoc :

Fonction itérative utilisant 'dolist'. On parcourt la 'a-liste' en testant chaque « car » des éléments de la a-list avec la clé entrée en paramètre. Si le test est vrai alors on retourne l'association.

```
(defun myassoc(cle a-list)
  (dolist (e a-list)
    (if (equal cle (car e))
        (return e)
      ))
  )
```

Résultats : La fonction renvoie bien (pierre 22) si Pierre est présent dans la 'a-liste' et nil sinon.

```
CG-USER(76): (myassoc 'Pierre '((Yolande 25) (Pierre 22) (Julie 45)))
(PIERRE 22)
CG-USER(77): (myassoc 'Yves '((Yolande 25) (Pierre 22) (Julie 45)))
NIL
```

2. Fonction cles :

-La fonction permet de retourner les clés d'une 'a-liste'. On utilise un mapcar pour parcourir notre a-list (plus performant que dolist), et on renvoie les « car » des a-list qui correspondent aux clés.

```
(defun cles(a-list)
  (mapcar #' car a-list
    )
  )
```

Résultat : On retrouve bien une liste des clés de la a-liste.

```
CG-USER(90): (cles '((Yolande 25) (Pierre 22) (Julie 45)))

(YOLANDE PIERRE JULIE)
```

3. Fonction creation :

La fonction permet de créer une a-liste à partir d'une liste de clé et d'une liste de valeur pour chaque clé.

Pour cela il suffit d'utiliser la fonction grouper définie plus haut. Idéalement on s'attend à ce que les deux listes soient de la même longueur, mais automatiquement le mapcar va s'arrêter au moment où la liste la plus petite se termine. Donc, nous n'avons pas besoin de faire de test.

```
(defun creation (listeCles listeValeurs)
  (grouper listeCles listeValeurs)
            )
```

Résultat :

```
CG-USER(100): (creation '(Yolande Pierre Julie) '(25 22 45))
((YOLANDE 25) (PIERRE 22) (JULIE 45))
CG-USER(101): (creation '(Yolande Pierre Julie) '(25 22 45 33))
((YOLANDE 25) (PIERRE 22) (JULIE 45))
```

On voit bien que pour le deuxième test, le résultat est le même pour le premier test où les deux listes ont la même longueur.

Exercice 4 : gestion d'une base de connaissance en LISP.

Question 1 :

-Définition de la fonction nom qui retourne le nom d'une tombe (premier élément de la liste) :

```
(defun nom(tombe)
  (car tombe))
```

Voici le résultat :

```
(nom '("Bécaud" 2001 (45 17) 2000 30))
```

```
CG-USER(162):
"Bécaud"
```

Ensuite, définition des fonctions :

-an-inhum qui retourne l'année d'inhumation d'une personne (second élément de la liste).

-num qui retourne le numéro de la tombe (troisième élément de la liste).

-rangee qui retourne le numéro de rangée où se situe la tombe (quatrième élément de la liste).

-debut-loc qui retourne l'année de début de location (cinquième élément de la liste).

-et duree-loc qui retourne le nombre d'année pendant lesquelles la tombe sera louée (sixième élément de la liste).

```
;;;fonction qui retourne l'année d'inhumation
(defun an-inhum(tombe)
  (cadr tombe)
  )
;;;test
(an-inhum '("Bécaud" 2001 (45 17) 2000 30))

;;;fonction retournant l'emplacement de la tombe num = 17
(defun num(tombe)
  (cadr (caddr tombe))
  )
;;;test
(num '("Bécaud" 2001 (45 17) 2000 30))


;;;fonction retournant la rangée de la tombe rangée = 45
(defun rangee(tombe)
  (car (caddr tombe))
  )
;;;test
(rangee '("Bécaud" 2001 (45 17) 2000 30))


;;;fonction retournant l'année de début de location
(defun debut-loc(tombe)
  (cadddr tombe)   ;;;renvoie le 4ième élément de la liste passée en argument
  )
;;;test
(debut-loc '("Bécaud" 2001 (45 17) 2000 30))


;;;fonction retournant la durée de location de la tombe
(defun duree-loc(tombe)
  (cadr(cdddr tombe))
  )
;;;test
(duree-loc '("Bécaud" 2001 (45 17) 2000 30))
```

Voici les résultats pour ces fonctions :

```
CG-USER(163):
AN-INHUM
CG-USER(164):
2001
CG-USER(165):
NUM
CG-USER(166):
17
CG-USER(167):
RANGEE
CG-USER(168):
45
CG-USER(169):
DEBUT-LOC
CG-USER(170):
2000
CG-USER(171):
DUREE-LOC
CG-USER(172):
30
```

On remarque que les tests sont valides.

Question 2 :

1. Définition de la base des cimetières :

```
(setq Base '((pere-lachaise("Bécaud" 2001 (45 17) 2000 30)
                           ("Desproges" 1988 (11 6) 1988 30)
                           ("Grappelli" 1997 (85 23) 1997 5)
                           ("Morrison" 1971 (6 12) 1971 30)
                           ("Mouloudji" 1994 (42 9) 1990 15)
                           ("Nohain" 1981 (89 14) 1979 15)
                           ("Oussekine" 1986 (85 37) 1986 5)
                           ("Petrucciani" 1999 (11 26) 1999 15)
                           ("Popesco" 1993 (85 16) 1985 30)
                           ("Signoret" 1985 (44 7) 1980 30)
                           ("Zavatta" 1993 (11 16) 1993 15)
                           ("Desproges" 2018 (11 6) 1988 30)
                           )
             (paris ("Bulag" 2009 (50 17) 2000 30) ("Dupuy" 1988 (11 6) 1988 30))
             (londres("wilson" 2003 (50 30) 2001 40))
             )
      )
```

On peut voir que trois cimetières existent : celui du père-lachaise, celui de Paris et celui de Londres. Pour chacun de ces cimetières il existe une ou plusieurs tombes.

2. Définition de la fonction qui-est-la :

Définition de la fonction QUI-EST-LA : on passe en argument une liste correspondant à la localisation de la tombe, le cimetière et la base de données.

```
(defun qui-est-la (local nomcim cim)
  (setq listtombe (trouver-cim nomcim cim))
  (dolist (tombe listtombe)
    (if (and (equal (car local)(rangee tombe))
             (equal (cadr local)(num tombe)))
        (return (nom tombe))
      "emplacement non attribué"
        )
    )
)
```

Nous avons besoin de la fonction « trouver-cim » comme vous pouvez le voir. Cette fonction est définie en amont, et sert à renvoyer la liste des tombes se trouvant dans le cimetière dont le nom est passé en argument dans la fonction qui-est-la.

Voici la fonction trouver-cim :

```
(defun trouver-cim (nomcim base)
  (dolist (cim base)
    (if (equal (car cim) nomcim) (return (cdr cim)) NIL
      )
    )
  )
```

On passe en argument le nom du cimetière voulu et la base de données.

La fonction va parcourir toute la base, et retourner la liste des tombes pour le nom du cimetière donné.

Voici son résultat :

```
(trouver-cim 'pere-lachaise Base)
```

```
CG-USER(174):
(("Bécaud" 2001 (45 17) 2000 30) ("Desproges" 1988 (11 6) 1988 30)
 ("Grappelli" 1997 (85 23) 1997 5)
 ("Morrison" 1971 (6 12) 1971 30 "Desproges" 2018 (11 6) 1988 30
  ...)
 ("Mouloudji" 1994 (42 9) 1990 15) ("Nohain" 1981 (89 14) 1979 15)
 ("Oussekine" 1986 (85 37) 1986 5)
 ("Petrucciani" 1999 (11 26) 1999 15)
 ("Popesco" 1993 (85 16) 1985 30) ("Signoret" 1985 (44 7) 1980 30)
 ...)
```

On remarque que seulement la liste des tombes du cimetière du pere-lachaise sont renvoyées.

Et voici le résultat pour la fonction qui-est-la :

```
(qui-est-la '(11 16) 'pere-lachaise base)
```

```
QUI-EST-LA
CG-USER(176):
"Zavatta"
```

Le test est valide puisque c'est bien Zavatta qui est à l'emplacement (11 16).

Question 3 :

Définition d'un prédicat : prevoyant?, qui a pour argument une tombe. Et qui retourne T si une personne a anticipé l'année de son inhumation.

```
(defun prevoyant? (tombe)
  (if (< (debut-loc tombe) (an-inhum tombe) )
      T
    NIL
    )
  )
(prevoyant? '("Zavatta" 1993 (11 16) 1993 15)) ;;;NIL
(prevoyant? '("Zavatta" 1993 (11 16) 1991 15)) ;;;T
(prevoyant? '("Grappelli" 1997 (3 10) 1997 5)) ;;;NIL
```

```
CG-USER(179):
PREVOYANT?
CG-USER(180):
NIL
CG-USER(181):
T
CG-USER(182):
NIL
```

On voit que le prédicat est valide : pour le premier test Zavatta n'a pas été prévoyant, alors que pour le second test il est prévoyant.

Question 4 :

Définition de la fonction nb-prevoyants dont on passe le nom du cimetière voulu et la base de données. Elle retourne le nombre de prévoyant qu'il y a dans le cimetière.

```
(defun nb-prevoyants(nomcim cim)
  (setq tombes (trouver-cim nomcim cim))
   (setq nombre 0) ;on définit un compteur
  (dolist (tombe tombes)
    (if (eq (prevoyant? tombe) T)
            (setq nombre (+ 1 nombre)) ;on incrémente le compteur
      )
    )
  nombre
)

(nb-prevoyants 'pere-lachaise Base)
```

D'abord, j'utilise la fonction trouver-cim pour avoir la liste des tombes du cimetière. Ensuite, j'initialise un compteur, puis dans la boucle, je test si la personne est prévoyante en appelant le prédicat. Le compteur est incrémenté si la personne est prévoyante. Le nombre de prévoyant est ensuite retourné. Le résultat pour le test ci-dessus est 6 ce qui correspond bien pour le père-lachaise.

Question 5 :

Définition de la fonction annuaire dont le nom du cimetière, la rangée et la base sont passés en arguments. La fonction retourne la liste des personnes étant enterrée à la rangée nrang.

```
(defun annuaire(nomcim nrang cim)
  (let ((liste (list)) (tombes (trouver-cim nomcim cim)))
    (dolist (tombe tombes)
      (if (equal nrang (rangee tombe))
          (push (nom tombe) liste)
        )
      )
    (reverse liste)
    )
  )
```

D'abord on définit des variables locales, la liste pour stocker les personnes et la liste tombes qui prendra la liste des tombes du cimetière voulu grâce à la fonction trouver-cim.

La boucle permet de parcourir toutes les tombes et pour chacune de tester si la personne est enterrée dans la bonne rangée grâce à la fonction rangée. Ensuite si tel est le cas, on ajoute le nom correspondant à la tombe à la liste c'est-à-dire à l'annuaire. Pour retourner la liste dans le sens original, on retourne l'inverse de la liste.

Voici le résultat du test ci-dessus :

```
CG-USER(214):
("Grappelli" "Oussekine" "Popesco")
```

On remarque que le résultat est concluant, ces personnes sont enterrées dans la même rangée et ce sont les seuls dans cette dernière.

Question 6 :

La fonction doyen-benjamin a pour argument le nom de cimetière voulu et la base de données. Elle doit retourner la liste du doyen et du benjamin du cimetière. Nous aurons besoin de la fonction trouver-cim pour avoir la liste des tombes du cimetière voulu. Nous déclarons trois variables pour stocker la valeur de l'année pour le doyen et celle du benjamin et la variable date qui sera utile pour comparer sa valeur avec doyen et benjamin, au lieu de rappeler la fonction an-inhum à chaque fois (c'est-à-dire quatre fois). Deux listes sont initialisées pour stocker la tombe quand on trouve un nouveau doyen ou benjamin, ceci évite de parcourir une nouvelle fois la liste tombes pour retrouver qui a été inhumé à telle date.

Le premier test dans le bloc dolist est fait pour traiter la première affectation, en effet par ce biais on peut affecter les deux variables de la date. Ensuite, à chaque itération de boucle, doyen et benjamin sont mis à jour s'ils correspondent aux tests. De même, on stocke les nouveaux doyen/benjamin potentiel dans leurs listes. Puis en terminaison, on retourne une seule liste en rejoignant les deux.

```
(defun doyen-benjamin(nomcim cim)
  (let ((tombes (trouver-cim nomcim cim))
        (listd (list))(listb (list))
        (date 0)
        (doyen 2020)(benj 0)
        ) ;;;ok
    (dolist (tombe tombes)
      (setq date (an-inhum tombe))
      (cond
       ((and (> date benj)(< date doyen))
        (progn (setq benj date)
          (setq doyen date)
          ))
       ((> date benj)(progn (setq benj date)
                            (setq listb tombe)))
        ((< date doyen)(progn (setq doyen date)(setq listd tombe))
       )
       )
      )
    (append listd listb)

    )
  )
```

Voici, le résultat du test :

```
(doyen-benjamin 'pere-lachaise Base)
```

```
("Morrison" 1971 (6 12) 1971 30 "Desproges" 2018 (11 6) 1988 30)
```

On remarque que ce sont bien d'une part le doyen du cimetière pere-lachaise, et de l'autre le benjamin de ce même cimetière.