RAPPORT TP1

# Exercice 1

Fonctions d’inversion (reverse) :

(defun reverseA(arg1 arg2 arg3)

    (list arg3 arg2 arg1)

)

La fonction n’admettant que 3 arguments, nous pouvons directement retourner une liste inversée de ces éléments. Cela est tout à fait gérable pour trois éléments.

(defun reverseB(L)

    (let ((listReversed '()))

    (if (listp L)

        (dolist (elem L)

            (push elem listReversed)

        )

        (print "Erreur : L n'est pas une liste")

    )

    listReversed

))

Pour l’inversion d’une liste à taille variable, nous avons développé deux versions (ici itérative) dont celle présentée repose sur une liste locale que l’on alimentera par ajout des éléments de la liste passée en paramètre au début.

(defun reverseC(L)

    (if (listp L)

        (if (> (length L) 0)

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

        )

        (print "Erreur : L n'est pas une liste")

    )

)

(defun monReverse(L)

    (if (listp L)

        (if (> (length L) 0)

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

        )

        (print "Erreur : L n'est pas une liste")

    )

)

Cette version, plus concise repose sur la récurisivité en allant récupérer le premier élement (« car ») des listes ôtées récursivement de leur premier élément (cdr). Ce « car » que l’on transforme en liste (pour assurer une bonne concaténation), après l’appel récursif (pour remonter la liste en sens inverse).

Fonctions de traitement sur les listes :

(defun double(L)

    (if (listp L)

        (if (> (length L) 0)

            (append (list (car L)) (if (atom (car L)) (list (car L))) (double (cdr L)))

        )

        (print "Erreur : L n'est pas une liste")

    )

)

Pour cette fonction de doublage, nous nous reprenons le principe de la version récursive de reverseC en ajoutant le « car » au début puis en ajoutant une seconde fois ce « car » si jamais il s’agit d’un atome.

(defun nombres3 (L)

    (if (listp L)

        (if (and (numberp (car L)) (numberp (cadr L)) (numberp (caddr L)))

            "BRAVO"

            "PERDU"

        )

        (print "Erreur : L n'est pas une liste")

    )

)

De la même manière que pour la première fonction d’inversement, ne s’agissant que de trois éléments à vérifier, on peut se contenter d’évaluer le « car », « cadr » et « caddr ».

(defun grouper(L1 L2)

    (let ((groupedList '()))

    (if (= (length L1) (length L2))

        (loop for idx from 1 to (length L1)

            do (push (list (nth (- (length L1) idx) L1) (nth (- (length L2) idx) L2)) groupedList)

        )

        (print "Erreur: Les deux listes sont de taille différente")

    )

    groupedList

))

Afin de grouper les éléments des deux listes, nous allons ici parcourir l’une d’entre elle (on choisit arbitrairement L1) à l’aide de son indice. On va ensuite prendre les éléments d’indice (taille – index courant) de L1 et L2, les regrouper dans une liste et les insérer en tête de la liste finale (d’où l’utilité de prendre l’élément d’indice taille – index parcouru).

(defun palindrome(L)

    (if (listp L)

        (equal (reverse L) L)

        (print "Erreur : L n'est pas une liste")

    )

)

Vérifier si la liste passée est un palindrome revient à vérifier si la liste de départ est équivalente à son inverse. Ici, nous utilisons la fonction native au LISP « reverse », nous aurions également pu utiliser nos fonctions développées précédemment.

# Exercice 2

(defun list-triple-couple(L)

    (if (listp L)

        (mapcar (lambda (x) (list x (\* 3 x))) L)

        (print "Erreur : L n'est pas une liste")

    )

)

Pour chaque élément de la liste L, nous effectuerons l’expression renvoyant une liste composée de x et de son triple.

# Exercice 3

(defun cles(a-list)

    (if (listp a-list)

        (mapcar (lambda (assoc)

            (car assoc)

        ) a-list)

    )

)

Les clés des associations représentant le premier élément de l’association (correspondant à une liste), pour chaque association nous allons prendre le premier élément à savoir sa clé ce qui retournera finalement la liste des clés.

(defun creation(listeCles listeVal)

    (grouper listeCles listeVal)

)

Pour pouvoir créer une liste d’associations, on va se servir de notre fonction « grouper » pour réunir les clés et les valeurs entre-elles sous forme d’associations distinctes.

(defun my-assoc (cle a-list)

    (if (listp a-list)

        (mapcar (lambda (x)

            (if (not (equal (member cle x) nil))

                (member cle x)

            )

        ) a-list)

        (print "Erreur : a-list n'est pas une liste")

    )

)

Dans chaque association de a-list, nous allons chercher si la clé passée en paramètre est présente, auquel cas on retourne l’association correspondante.

Aussi, nous avons vérifié quand c’était nécessaire les différents tests sur les paramètres (taille définie, paramètre est bien une liste).

# Exercice 4

(defun bonformat(tombe)

    (if (and (listp tombe) (equal (length tombe) 5) (listp (caddr tombe)))

        t

        nil

    )

)

(defun nom (tombe)

    (if (bonformat tombe)

        (car tombe)

        "Erreur format données tombe"

    )

)

(defun an-inhum (tombe)

    (if (bonformat tombe)

        (nth 1 tombe)

        "Erreur format données tombe"

    )

)

(defun num (tombe)

    (if (bonformat tombe)

        (cadr (caddr tombe))

        "Erreur format données tombe"

    )

)

(defun rangee (tombe)

    (if (bonformat tombe)

        (car (caddr tombe))

        "Erreur format données tombe"

    )

)

(defun debut-loc (tombe)

    (if (bonformat tombe)

        (nth 3 tombe)

        "Erreur format données tombe"

    )

)

(defun duree-loc (tombe)

    (if (bonformat tombe)

        (car (last tombe))

        "Erreur format données tombe"

    )

)

(defun emplacement (tombe)

    (if (bonformat tombe)

        (nth 2 tombe)

        "Erreur format données tombe"

    )

)

(defun qui-est-la (emplacement cimetiere)

    (if (bonformat (car cimetiere))

        (if (equal emplacement (emplacement (car cimetiere)))

            (nom (car cimetiere))

            (qui-est-la emplacement (cdr cimetiere))

            )

        (if (listp (nth 1 cimetiere))

            (qui-est-la emplacement (nth 1 cimetiere))

            "Emplacement non attribué"

        )

    )

)

(defun prevoyant? (tombe)

    (> (an-inhum tombe) (debut-loc tombe))

)

(defun nb-prevoyants (cimetiere)

    (let ((cim (cadr cimetiere))

          (nb 0))

        (setq nb 0)

        (loop for i from 0 to (- (list-length cim) 1)

            do

                (if (prevoyant? (nth i cim))

                    (setq nb (+ nb 1))

                    (setq nb nb)

                )

        )

        nb

    )

)

(defun annuaire (cimetiere rangee)

    (let ((cim (cadr cimetiere))

          (noms '()))

        (loop for i from 0 to (- (list-length cim) 1)

            do

                (if (equal rangee (rangee (nth i cim)))

                    (setq noms

                        (push

                            (nom (nth i cim )) noms)

                            )

                )

                (setq noms noms)

        )

        noms

    )

)

(defun doyen-benjamin (cimetiere )

    (let\* ((cim (cadr cimetiere))

          (doyen (car cim))

          (benjamin (car cim)))

        (loop for i from 1 to (- (list-length cim) 1)

            do

                (if (> (an-inhum doyen) (an-inhum (nth i cim)))

                    (setq doyen

                        (nth i cim)

                    )

                    (setq doyen doyen)

                )

                (if (< (an-inhum benjamin) (an-inhum (nth i cim)))

                    (setq benjamin (nth i cim))

                    (setq doyen doyen)

                )

        )

        (list doyen benjamin)

    )

)