Fonctionnelles

Exercice 1 :
Evaluer les expressions suivantes :

1. `(remove-if #'zerop '(0 1 0 2 0 3 0 0 0))`
2. `(remove-if-not (lambda (x) (= x 3)) '(0 1 2 3 0 1 2 3))`
3. `(mapcar (lambda (x) (* 2 x)) '(1 2 3))`
4. `(funcall (lambda (x y) (+ (* 2 x) y)) 2 3)`
5. `(assoc 'bleu '((rouge . red) (vert . green) (bleu . blue) (jaune . yellow)))`

Exercice 2 :

1. Ecrivez une fonction $list - abs - mapcar$ similaire à $list - abs - recursif$ vue au TD sur les listes (fonction qui retourne la liste des valeurs absolues des éléments d'une liste passée en argument) sauf qu'elle utilise mapcar.
2. Ecrivez de deux façons différentes une fonction $list - carres$ qui retourne les carrés des éléments de la liste fournie en paramètre. Les deux versions utilisent $mapcar$. La première version utilisera $labels$, la deuxième utilisera une $lambda$ expression.

Exercice 3 :

1. Ecrire la fonction qui, étant donnée une fonction f passée en paramètre, renvoie $f(1)$.
2. L'appliquer avec $f(x) = 3x^2 + 4.7$.

Exercice 4 :

1. Ecrire la fonction qui, étant donnée une valeur n, construit la fonction qui multiplie un nombre avec n.
2. En utilisant la fonction précédente, définir la fonction qui double son argument.

Exercice 5 :
Ecrire une fonction prenant une fonction (à un argument) et un entier n en arguments, et qui permette d'itérer n fois la fonction. Exemple :

```
(defparameter 3+ (iterer #'1+ 3))
(funcall 3+ 1) --> 4
```

Exercice 6 :

1. Définir la fonction *somme*, prenant en arguments une fonction d'une variable et une liste, et qui calcule la somme des images par la fonction des éléments de la liste :
   ```
   (defun somme (f L)
      ...)
   ```
2. De même, définir une fonction *produit* qui calcule le produit des images par une fonction des éléments d'une liste.
3. Définir une fonction $liste - itere$ permettant d'appliquer récursivement une fonction de deux variables aux éléments d'une liste, suivant le schéma :
   `(f x1 (f x2 ( ... (f xN b) ...)))`

4. Utiliser cette fonction pour définir les fonctions $mon-append$ et $mon-mapcar$ qui ont respectivement les mêmes fonctionnalités que *append* et *mapcar*, mais prenant uniquement deux arguments. Plus précisément, $mon-append$ réalise la concaténation de deux listes, et $mon-mapcar$ prend en argument une fonction et une liste, et renvoie la liste des images par la fonction des éléments de la liste.
5. Selon la même idée, écrire une fonction $produit-itere$ (utilisant la fonction $liste-itere$) permettant de calculer le produit de l'image par une fonction des éléments passés en argument. Exemple :
`(produit-itere #'sqrt '(4 9 25)) ; --> 30`
6. Ecrire une fonction $itere-liste$, qui applique une fonction de deux variables selon le schéma :
`(f ... (f (f b x1) x2) ... xN)`
7. Utiliser une de ces deux fonctions $liste-itere$ ou $itere-liste$ (la plus appropriée) pour écrire une fonction *renverse*, qui inverse l'ordre des éléments d'une liste.

Exercice 7 :

1. Fonctions traitant les éléments d'une liste `l` vérifiant un prédicat `pred`
   - Ecrire la fonction `combien` qui retourne le nombre d'éléments de la liste `l` vérifiant le prédicat `pred`.
     Exemples :
     ```
     (combien #'evenp '(1 5 7 6 2)) -> 2
     (combien #'numberp '(2 3 4 a b 5 t + 8)) -> 5
     ```
   - Ecrire une fonction qui retourne le premier élément de `l` vérifiant `pred`.
   - Ecrire une fonction qui retourne la liste des éléments de `l` vérifiant `pred`.
2. Extensions de `mapcar` et `apply`
   On donne la fonction $append-map$ suivante, elle pourra être utile dans les questions suivantes :
   ```
   (defun append-map (f l)
       ``returns the list (f(x1) f(x2) ... f(xn))''
       (apply #'append (mapcar f l)))
   ```

   Voici un exemple d'utilisation :
   ```
   (append-map (lambda (x) (list x (* x x)))
   '(1 2 3 4))  -> (1 1 2 4 3 9 4 16)
   ```

   - Parfois on a besoin d'un `map` sélectif, lorsqu'on ne veut appliquer `f` qu'aux éléments d'une liste vérifiant un certain prédicat et obtenir au final une liste ne contenant que les valeurs modifiées par `f`. Ecrire la fonction correspondante `(map-select f L pred)` en utilisant `append-map`.
     Exemple :
     ```
     (map-select  (lambda (x) (/ 1 x))
             '(a 2 0 4 10)
             (lambda (x) (and (numberp x) (not (zerop x)))))  -> (1/2 1/4 1/10)
     ```
   - Ecrire une version simplifiée de `remove-if(pred l)`. Cette fonction reconstruit la liste `l` sans les éléments qui vérifient `pred`. La tester avec $pred(x) \equiv (x = 0)$.

Exercice 8 :
En Lisp, un vecteur peut être représenté par la liste de ses n éléments $(a1\ a2\ a3\ a4\ ..\ an)$ et une matrice peut être représentée par la liste dont les éléments (sur n lignes et n colonnes) sont $((a11\ a12\ ..\ a1n)..(an1\ an2\ ..\ ann))$

1. Ecrire une fonction *prod_scalaire* qui retourne le produit scalaire de deux vecteurs
2. Ecrire une fonction *mat_vect* qui retourne le produit d'un vecteur par une matrice
3. Ecrire une fonction *mat_mat* qui retourne le produit de deux matrices
4. Ecrire une fonction *transposition* qui retourne la transposée d'une matrice

Exercice 9 :
Ecrire une fonction fixpoint généralisant le calcul de la racine carrée vu en TD2 par la méthode du point fixe. Cette fonction doit admettre 3 paramètres : une fonction de D dans D, un prédicat sur D*D indiquant l'arrêt du calcul, et un élement de départ du domaine D. Ecrire à nouveau la fonction racine carrée en utilisant fixpoint.