ENSEIRB-Matmeca
2019-2020

1ère année - Filière Info
PG104

TD n°7 - Map et reduce

Exercice 1: Calculs de dérivées / Fonctions d'ordre supérieur

1. Écrire la fonction qui donne une approximation de la dérivée.

$$F'(x) = \frac{F(x+h) - F(x-h)}{2h} + O(h)$$

2. Généraliser aux dérivées n-ièmes en utilisant la fonction précédente.

Vous devriez remarquer que l'application répétitive de l'approximation de la dérivée n'est pas un algorithme efficace pour trouver la n-ième dérivée.

3. En utilisant les formules suivantes :

$$F''(x) = \frac{F(x+h) - 2F(x) + F(x-h)}{h^2} + O(h^2)$$

$$F^{(3)}(x) = \frac{-F(x-2h) + 2F(x-h) - 2F(x+h) + F(x+2h)}{2h^3} + O(h^2)$$

écrire la fonction qui donne une approximation de la première à la troisième dérivée. Comparer les résultats avec ceux de la dernière fonction.

Exercice 2: Bibliothèque standard

Deviner le résultat puis évaluer les expressions suivantes :

1. `((lambda (x y) (+ (* 2 x) y)) 2 3)`
2. `(filter positive? '(0 1 0 2 0 3 0 0 0))`
3. `(filter (lambda (x) (= x 3)) '(0 1 2 3 0 1 2 3))`
4. `(map (lambda (x) (* 2 x)) '(1 2 3))`

Exercice 3: Utilisation des maps

1. Écrire une fonction `list-abs-map` qui retourne la liste des valeurs absolues des éléments d'une liste passée en argument, en utilisant `map`.
2. Écrire de deux façons différentes une fonction `list-squares` qui retourne les carrés des éléments de la liste fournie en paramètre, en utilisant `map`.
   — la première version utilisera `letrec` avec une fonction nommée ;
   — la seconde utilisera directement une `lambda` expression anonyme.

Exercice 4: Fonctions avec prédicats

Écrire la fonction `how-many` qui retourne le nombre d'éléments de la liste `l` vérifiant le prédicat `pred`. On pourra faire une version sans map et une avec map.

Exemple :

```
(how-many even?   '(1 5 7 6 2))         ;; → 2
(how-many number? '(2 3 4 a b 5 t + 8)) ;; → 5
```

Exercice 5: Vecteurs et matrices

En Scheme, un vecteur peut être représenté par la liste de ses n éléments :

$$(a_1\ a_2\ a_3\ a_4\ \ldots\ a_n)$$

... et une matrice peut être représentée par la liste dont les éléments (sur n lignes et n colonnes) :

$$((a11\ a12\ \ldots\ a1n)\ldots(an1\ an2\ \ldots\ ann))$$

1. Écrire une fonction `scalar-product` qui retourne le produit scalaire de deux vecteurs ;
2. Écrire une fonction `transpose` qui retourne la transposée d'une matrice.
3. Écrire une fonction `mat-vect` qui retourne le produit d'un vecteur par une matrice.
4. Écrire une fonction `mat-mat` qui retourne le produit de deux matrices.
5. Pour aller plus loin... Écrire une fonction `has-path` qui prend en argument une matrice carrée et teste si elle contient au moins une ligne, colonne ou diagonale dont tous les éléments sont non nuls (une telle fonction peut être utilisée pour vérifier les conditions de gain dans des jeux comme le *tic-tac-toe*).

Remarque : les fonctions peuvent être écrites de façon récursive mais c'est fastidieux ... et il est possible de faire plus finement en utilisant `map` et `apply`. Pour simplifier, aucune vérification de longueur ne sera faite.

Exercice 6: Composition de fonctions / Pliages

1. Définir la fonction `map-and-sum`, 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.
2. De même, définir une fonction `map-and-prod` qui calcule le produit des images par une fonction des éléments d'une liste.
3. Définir deux fonctions `iter-from-right` et `iter-from-left` permettant d'appliquer récursivement une fonction de deux variables aux éléments d'une liste, suivant les schémas respectifs :

   `(f x1 (f x2 ... (f xn b) ...))`   `(f xn ... (f x2 (f x1 b)) ... )`
4. Utiliser une de ces fonctions pour définir les fonctions `my-append` et `my-map` qui ont respectivement les mêmes fonctionnalités que `append` et `map`, mais prenant uniquement deux arguments.
5. Écrire une fonction `prod-iterate` (toujours en utilisant une des fonctions ci-dessus) calculant le produit $\prod_i f(x_i)$.

   Exemple : `(prod-iterate sqrt '(4 9 25)) ; → 30`
6. Utiliser une de ces fonctions pour écrire une fonction `my-reverse`, qui inverse l'ordre des éléments d'une liste.

Exercice 7: Extensions de map et apply

On donne la fonction `append-map` suivante, parfois aussi appelée `concatMap`, qui pourra être utile dans les questions suivantes :

```
;; Returns the concatenation of the list (f(x1) f(x2) ... f(xn))
(define (append-map f l)
    (apply append (map f l)))
```

Exemple :

```
(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.

1. Écrire 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 (number? x) (not (zero? x)))))  ;; → (1/2 1/4 1/10)
   ```

2. Écrire une fonction `(remove-if pred l)` qui implémente un comportement inverse de la fonction `filter` (qu'il ne faut pas utiliser ici) , i.e. qui reconstruit la liste `l` sans les éléments qui vérifient `pred`. La tester avec $pred(x) \equiv (x > 0)$.