ISIMA 1ère année

4 septembre 2007
Durée : 2 heures
Documents autorisés

PROGRAMMATION FONCTIONNELLE

Exercice 1 :

Toutes les fonctions de cet exercice doivent être écrites en faisant appel à la fonction `map`, et sans utiliser une autre fonction de l'exercice.

1°) a) Ecrire une fonction `mapf1LC` ayant comme arguments une fonction `f1` d'un seul argument et une liste de couples `LC` de type `((`x_1`,` y_1`), …, (`x_n`,` y_n`))`, où x_1, …, x_n et y_1, …, y_n, sont du type de l'argument de `f1`, et telle que l'évaluation de l'expression `(mapf1LC f1 LC)` retourne une liste de type `((f1(`x_1`), f1(`y_1`)), …, (f1(`x_n`), f1(`y_n`)))`.

b) Ecrire une fonction `mapf1L1L2` ayant comme arguments une fonction `f1` d'un seul argument et deux listes `L1` et `L2` respectivement de type `(`x_1`, …, `x_n`)` et `(`y_1`, …, `y_n`)`, où x_1, …, x_n et y_1, …, y_n, sont du type de l'argument de `f1`, et telle que l'évaluation de l'expression `(mapf1L1L2 f1 L1 L2)` retourne une liste de type :

`((f1(`x_1`), f1(`y_1`)), …, (f1(`x_n`), f1(`y_n`)))`.

Remarque : Ecrire `mapf1L1L2` directement, sans utiliser `mapf1LC`.

2°) a) Ecrire une fonction `mapf2LC` ayant comme arguments une fonction `f2` de deux arguments et une liste de couples `LC` de type `((`x_1`,` y_1`), …, (`x_n`,` y_n`))`, où x_1, …, x_n sont du type du premier argument de `f2`, et y_1, …, y_n, sont du type du deuxième argument de `f2`, et telle que l'évaluation de l'expression `(mapf2LC f2 LC)` retourne une liste de type `(f2(`x_1`,` y_1`), …, f2(`x_n`,` y_n`))`.

b) Ecrire une fonction `mapf2L1L2` ayant comme arguments une fonction `f2` de deux arguments et deux listes `L1` et `L2` respectivement de type `(`x_1`, …, `x_n`)`, où x_1, …, x_n sont du type du premier argument de `f2`, et `(`y_1`, …, `y_n`)`, où y_1, …, y_n, sont du type du deuxième argument de `f2`, et telle que l'évaluation de l'expression `(mapf2L1L2 f2 L1 L2)` retourne une liste de type `(f2(`x_1`,` y_1`), …, f2(`x_n`,` y_n`))`.

Remarque : Ecrire `mapf2L1L2` directement, sans utiliser `mapf2LC`.

3°) a) Ecrire une fonction `mapfCLC` ayant comme arguments une fonction `fC` d'un seul argument, qui est un couple de valeurs, et une liste de couples `LC` de type

((x_1, y_1), …, (x_n, y_n)), où (x_1, y_1), …, (x_n, y_n) sont du type de l'argument de `fC`, et telle que l'évaluation de l'expression `(mapfCLC fC LC)` retourne une liste de type (`fC`((x_1, y_1)), …, `fC`((x_n, y_n))).

b) Ecrire une fonction `mapfCL1L2` ayant comme arguments une fonction `fC` d'un seul argument, qui est un couple de valeurs, et deux listes `L1` et `L2` respectivement de type (x_1, …, x_n), où x_1, …, x_n sont du type du premier élément de l'argument de `fC`, et (y_1, …, y_n), où y_1, …, y_n, sont du type du deuxième élément de l'argument de `fC`, et telle que l'évaluation de l'expression `(mapfCL1L2 fC L1 L2)` retourne une liste de type (`fC`((x_1, y_1)), …, `fC`((x_n, y_n))).

Remarque : Ecrire `mapfCL1L2` directement, sans utiliser `mapfCLC`.

Solution :

1°) a)
```
(define mapf1LC (lambda (f1 LC)
      (map (lambda (C) (list (f (car C)) (f (cadr C)))) LC) ))
```
b)
```
(define mapf1L1L2 (lambda (f1 L1 L2)
    (map (lambda (x y) (list (f x) (f y))) L1 L2) ))
```
2°) a)
```
(define mapf2LC (lambda (f2 LC)
      (map (lambda (C) (f2 (car C) (cadr C))) LC) ))
```
b)
```
(define mapf2L1L2 (lambda (f2 L1 L2) (map f2 L1 L2)))
```
3° a)
```
(define mapfCLC (lambda (fC LC) (map fC LC)))
```
b)
```
(define mapfCL1L2 (lambda (fC L1 L2)
     (map (lambda (x y) (fC (list x y))) L1 L2) ))
```

Exercice 2 : Partitions d'un ensemble en *k* parties

Dans cet exercice, on représente un ensemble par une liste. Par exemple l'ensemble {1, 2, 3, 4} est représenté par la liste `(1 2 3 4)` (les éléments de la liste pouvant figurer dans un ordre quelconque).

On appelle partie d'un ensemble *E* tout sous ensemble de *E*.

Une partition d'un ensemble *E* est un ensemble de parties de *E*, (P_1, …, P_k), $1 \le k \le$ Card (*E*), vérifiant les trois propriétés suivantes :

- $\forall i \in \{1, \ldots, k\}, P_i \neq \varnothing$;
- $\forall (i, j) \in \{1, \ldots, k\}^2, i \neq j \Rightarrow P_i \cap P_j = \varnothing$;
- $\bigsqcup_{i \in \{1, \ldots, k\}} P_i = E$.

Une partition d'un ensemble *E* sera donc représentée par une liste de sous listes. Par exemple `((1) (2 4) (3))` est une partition de `(1 2 3 4)`.

L'objet de cet exercice est d'écrire une fonction ayant comme argument une liste `E` représentant un ensemble et retournant la liste de toutes les partitions de `E` en *k* parties. Des exemples sont donnés dans la seconde partie de cet exercice.

1°) Ecrire une fonction `distribuer` ayant comme arguments un élément `x` d'un ensemble, et une partition `Pa` de ce même ensemble, et telle que l'évaluation de l'expression `(distribuer x Pa)` retourne une liste de partitions où dans chacune d'entre elles, `x` est inséré en tête d'une des parties de `Pa`.
Par exemple l'évaluation de `(distribuer 1 '((2) (3) (4)))` doit retourner la liste de partitions `( ((1 2) (3) (4)) ((2) (1 3) (4)) ((2) (3) (1 4)) )`.

Solution :
```
(define distribuer (lambda (x Pa)
   (if (cdr Pa)
      (let ((distcdr (distribuer x (cdr Pa))))
         (cons (cons (cons x (car Pa)) (cdr Pa))
               (map (lambda (P) (cons (car Pa) P)) distcdr)
          ) )
      (list (list (cons x (car Pa)))) ) ))
```

2°) Ecrire une fonction `part` ayant comme argument une liste `E`, représentant un ensemble, et un entier `k`, et telle que l'évaluation de l'expression `(part E k)` retourne l'ensemble de toutes les partitions de `E` en `k` parties. Si `n` désigne la longueur de `E`, l'évaluation de `(part E k)` avec `k` > `n` doit retourner `()`.

Par exemple :
- l'évaluation de `(part '(1 2 3) 1)` doit retourner `(((1 2 3)))` ;
- l'évaluation de `(part '(1 2 3) 3)` doit retourner `(((1) (2) (3)))` ;
- l'évaluation de `(part '(1 2 3) 2)` doit retourner
  `( ((1) (2 3)) ((1 2) (3)) ((2) (1 3)) )`.

Le dernier exemple suggère que l'ensemble des partitions de `E` en `k` parties peut être construit comme l'union de deux ensembles :
- le premier ensemble est construit en insérant le singleton contenant `(car E)` en tête de chaque partition de l'ensemble des partitions de `(cdr E)` en `k` - 1 parties. Attention ! Dans l'exemple ci-dessus l'ensemble des partitions de `(2 3)` en une partie ne contient qu'un seul élément : la partition `((2 3))`.
- le second ensemble est construit en "distribuant" `(car E)` dans chaque partition de l'ensemble des partitions de `(cdr E)` en `k` parties. Attention ! Dans l'exemple ci-dessus l'ensemble des partitions de `(2 3)` en 2 parties ne contient qu'un seul élément : la partition `((2) (3))`.

Solution :
```
(define part (lambda (E k)
   (let ((n (length E)))
   (cond ((> k n) ())
         ((= k n) (list (map list E)))
         ((= k 1) (list (list E)))
```

```
      ( else (let ((Pcdrk-1 (part (cdr E) (- k 1)))
                   (Pcdrk (part (cdr E) k)) )
(append
   (map (lambda (P) (cons (list (car E)) P)) Pcdrk-1)
   (append-map (lambda (P) (distribuer (car L) P)) Pcdrk)
) )) ) ) ))
```