ISIMA 1ère année

6 septembre 2002
Durée : 2 heures
Documents autorisés

# ***PROGRAMMATION FONCTIONNELLE***

## **Exercice 1** :

Ecrire une fonction `parties` qui a comme argument une liste `E` représentant un ensemble et qui retourne l'ensemble des parties de `E` sous forme d'une liste de sous-listes, chaque sous-liste représentant une partie de `E` (l'ensemble vide sera représenté par la liste vide).
Par exemple, l'évaluation de `(parties '(1))` doit retourner `((1) ())`,
celle de `(parties '(1 2))` doit retourner `((2 1) (2) (1) ())`,
celle de `(parties '(1 2 3))` doit retourner `((3 2 1) (3 2) (3 1) (3) (2 1) (2) (1) ())`

Solution :
```
(define parties (lambda (E)
   (if E
      (let ((pcdr (parties (cdr E))))
         (append (map (lambda (p) (cons (car E) p)) pcdr) pcdr) )
      '(()) ) ))
```

## **Exercice 2** :

On appelle loi de composition interne sur un ensemble $E$ toute une application de $E \times E$ dans $E$. Etant donnés deux ensembles $E$ et $F$ munis de lois de composition interne notées respectivement "+" et "×", on appelle morphisme de $(E, +)$ dans $(F, \times)$ toute application $\mu : E \to F$ telle que : $\forall (x, y) \in E^2,\ \mu(x + y) = \mu(x) \times \mu(y)$.
Nous ne considérerons, dans cet exercice, que des ensembles finis, que nous représenterons par des listes dont les éléments sont tous distincts deux à deux.
Ecrire une fonction `morphisme` ayant cinq arguments :

- une liste `E` représentant un ensemble fini ;
- une fonction `opE` représentant une loi de composition interne sur `E` ;
- une liste `F` représentant un ensemble fini ;
- une fonction `opF` représentant une loi de composition interne sur `F` ;
- une fonction `mu` ayant comme argument un élément de la liste `E` et retournant un élément de la liste `F` ;

et telle que l'évaluation de `(morphisme E opE F opF mu)` retourne la valeur logique "vrai" si mu est un morphisme de (E, op1) dans (F, op2), et la valeur logique "faux" sinon.

Solution :
```
(define qqs (lambda (L P)
   (if L (and (P (car L) (qqs (cdr L) P)) #t) ))
(define morphisme (lambda (E opE F opF mu)
   (qqs E (lambda (x) (qqs F (lambda (y)
      (equal? (mu (opE x y)) (opF (mu x) (mu y))) )))) ))
```

## **Exercice 3** :

1°) Ecrire une fonction `C` ayant comme argument une liste d'entiers `L` et retournant une liste construite à partir de `L` en remplaçant toutes les successions de $n$ valeurs identiques $i$, pour $n \geq 3$, par une liste contenant $n$ et $i$.
Par exemple, l'évaluation de `(C '(0 0 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 1))` doit retourner `((3 0) (5 1) 0 0 1 0 1 0 (6 1))`.
2°) Ecrire une fonction `invC` ayant comme argument une liste obtenue par l'évaluation de la fonction `C` avec une liste d'entiers `L`, et retournant `L`.

Par exemple, l'évaluation de `(invC '((3 0) (5 1) 0 0 1 0 1 0 (6 1)))` doit retourner `'(0 0 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 1).`

3°) Ecrire une fonction `mapC` ayant comme argument une liste de listes d'entiers LL et retournant une liste construite à partir de `LL` en appliquant `C` à chaque sous-liste élément de LL ainsi qu'en remplaçant toutes les successions de *n* sous-listes identiques *l*, pour $n \geq 2$, par une liste contenant *n* et *l*.

Par exemple, l'évaluation de `(C '((0 0 0 1 1) (0 0 0 1 1) (0 0 0 1 1) (0 0 1 1 1) (0 0 1 1 1))` doit retourner `((3 ((3 0) 1 1)) (2 (0 0 (3 1))))`.

4°) Ecrire une fonction `invmapC` ayant comme argument une liste obtenue par l'évaluation de la fonction mapC avec une liste de listes d'entiers `LL`, et retournant `LL`.

Par exemple, l'évaluation de `(C '((3 ((3 0) 1 1)) (2 (0 0 (3 1)))))` doit retourner `((0 0 0 1 1) (0 0 0 1 1) (0 0 0 1 1) (0 0 1 1 1) (0 0 1 1 1).`

Solution :

1°)
```
(define C (lambda (L)
   (if (and L (cdr L) (cddr L))
      (let ((CcdrL (C (cdr L))))
         (if (list? (car CcdrL))
            (if (equal (cadar CcdrL) (car L))
               (cons (list (+ 1 (caar CcdrL)) (cadar CcdrL)) (cdr CcdrL))
               (cons (car L) CcdrL) )
            (if (and (cdr CcdrL)
                     (equal (carL) (car CcdrL))
                     (equal (car L) (cadr CcdrL)) )
               (cons (list 3 (car L)) (cddR CcdrL))
               (cons (car L) CcdrL) ) ) )
      L ) ))
```
2°)
```
(define invC (lambda (L)
   (if L
      (if (list? (car L))
         (append (faireliste (caar L) (cadr L)) (invC (cdr L)))
         (cons (car L) (invC (cdr L)) )
      () ) ))
(define faireliste (lambda (n e)
   (if (> n 0) (cons e (faireliste (- n 1) e)) ()) ))
```
3°)
```
(define mapC (lambda (LL)
   (C2 (map C LL)) ))
(define C2 (lambda (L)
   (if (and L (cdr L))
      (let ((CcdrL (C (cdr L))))
         (if (list? (car CcdrL))
            (if (equal (cadar CcdrL) (car L))
               (cons (list (+ 1 (caar CcdrL)) (cadar CcdrL)) (cdr CcdrL))
               (cons (car L) CcdrL) )
            (if (equal (carL) (car CcdrL))
               (cons (list 2 (car L)) (cdR CcdrL))
               (cons (car L) CcdrL) ) ) )
      L ) ))
```
4°)
```
(define invmapC (lambda (L)
   (map invC (invC L)) ))
```