ISIMA 1ère année

1er février 2011

Durée : 2 heures

Documents autorisés

PROGRAMMATION FONCTIONNELLE

Pensez à expliciter en français ce que doivent faire vos fonctions.
Vous pourrez bien sûr écrire des fonctions intermédiaires.

Exercice 1 :

On définit une liste profonde d'entiers comme une liste dont les éléments sont, soit des listes profondes d'entiers, soit des entiers. Par exemple : `(1 ((2) 3) (4 (5 (6) 7)) 8)`.

La profondeur d'un élément *e*, élément direct d'une liste *L*, est égale à la profondeur de *L* incrémentée de 1, la profondeur de la liste d'origine étant fixée à 0. Dans l'exemple ci-dessus, la profondeur de 1 est 1, celle de 2 est 3, celle de 3 est 2, etc.

Ecrire une fonction `profondeur` ayant comme arguments une liste profonde `L`, et telle que l'évaluation de l'expression `(profondeur L)` retourne une liste de couples contenant les éléments de `L` et leur profondeur.

Par exemple `(profondeur '(1 ((2) 3) (4 (5 (6) 7)) 8))` doit retourner `((1 1) (2 3) (3 2) (4 2) (5 3) (6 4) (7 3) (8 1))`.

Rappel : La fonction prédéfinie `list?` est un prédicat tel que l'évaluation de `(list? x)` retourne `#t` si `x` est une liste et `#f` sinon.

Solution :

```
(define profondeur (lambda (L) (pro L 0)))
(define pro (lambda (L p)
   (if (null? L) ()
       (if (list? (car L))
           (append (pro (car L) (+ p 1))
                   (pro (cdr L) p) )
           (cons (list (car L) p) (pro (cdr L) p)) ) ) ))
```

Problème :

Considérons une liste de nombres réels <u>distincts</u>, L, et un nombre réel x. L'objet de ce problème est d'écrire une fonction SP telle que l'évaluation de l'expression (SP L x) retourne une sous liste de L dont la somme des éléments est la plus proche possible de x.

Par exemple, l'évaluation de l'expression (SP '(0.30 0.22 0.21 0.26) 0.49) doit retourner la sous liste (0.22 0.26).

Première partie : On suppose que l'on dispose d'une fonction EP ayant comme argument une liste E, représentant un ensemble *E*, et telle que l'évaluation de l'expression (EP E) retourne une liste représentant l'ensemble des parties non vides de *E*. Par exemple l'évaluation de l'expression (EP '(1 2 3)) doit retourner ((1) (2) (3) (1 2) (1 3) (2 3) (1 2 3)).

Ecrire la fonction SP en <u>utilisant</u> la fonction EP (*on ne demande pas d'écrire la fonction* EP).

<u>Solution</u> :

```
(define SP (lambda (L x)
   (let ((EPL (EP L)))
        (SP2 (cdr EPL) x (car EPL)
             (abs (- (apply + (car EPL)) x)) ) ) ))
(define abs (lambda (x) (if (> x 0) x (- 0 x))))
(define SP2 (lambda (EPL x P d)
   (if (null? EPL) P
       (let ((d2 (abs (- (apply + (car EPL)) x))))
       (cond
         ((= d2 0) (car EPL))
         ((< d2 d) (SP2 (cdr EPL) x (car EPL) d2))
         ( else    (SP2 (cdr EPL) x P d)) ) ) ) ))
```

Deuxième partie : On considère une marge d'erreur, err, et on accepte que la fonction retourne une sous liste dont la somme des éléments n'est peut-être pas la plus proche de x, mais dont la différence en valeur absolue avec x est inférieure ou égale à err. On peut donc éviter de construire toutes les sous listes non vides de L (et donc d'utiliser la fonction EP) et retourner la première solution convenable rencontrée, même si elle n'est pas optimale.

Par exemple, avec L = (0.30 0.22 0.21 0.26), x = 0.49, et err = 0.05, on peut retourner la sous liste (0.30 0.22), même si la solution optimale est (0.22 0.26), car $| 0,52 - 0,49 | \leq$ err = 0,05.

On souhaite donc écrire une fonction `SPa`, ayant comme arguments `L`, `x`, et `err` (dans cet ordre), et qui implémente cette solution approximative.

Un principe général peut être de chercher une solution (c'est-à-dire une sous liste de `L`) de longueur k (en commençant par $k = 1$) et, en cas d'échec, de rappeler la fonction avec $k + 1$ jusqu'à ce que l'on trouve ou que l'on ait échoué avec toutes les valeurs de k entre 1 et la longueur de la liste (cette méthode évite de construire toutes les solutions et s'arrête à la première solution acceptables).

1°) Fonction de recherche d'une solution dans une liste de *k*-uplets

Ecrire une fonction `recherche` ayant comme arguments une liste de *k*-uplets, `Lkuplets`, un nombre réel `x` et une marge d'erreur, `err`, et telle que l'évaluation de l'expression `(recherche Lkuplets x err)` retourne le premier *k*-uplet de `Lkuplets` dont la somme S des éléments est telle que $| S - \mathtt{x} | \leq \mathtt{err}$, s'il en existe au moins un, et retourne `()` sinon. On pourra utiliser la fonction Scheme prédéfinie `abs` qui retourne la valeur absolue de son argument réel.

Solution :

```
(define recherche (lambda (Lkuplets x err)
   (if (null? Lkuplets) ()
       (if (<= (abs (- (apply + (car Lkuplets)) x)) err)
           (car Lkuplets)
           (recherche (cdr Lkuplets) x err) ) ) ))
```

2°) Construction d'une liste de (*k* + 1)-uplets à partir d'une liste de *k*-uplets

Supposons que la liste `L` est triée et prenons, pour expliquer la méthode de construction, un exemple simple : `L` = `(1 2 3 4)`.

Les 1-uplets sont `(1)`, `(2)`, `(3)` et `(4)`.

Les couples contenant `1` peuvent être construits en mettant `2`, `3`, ou `4` en tête de `(1)`. On obtient ainsi `(2 1)`, `(3 1)` et `(4 1)`.

Comme l'addition est commutative, il est inutile de construire (x, y) si l'on a déjà construit (y, x). Donc les couples contenant `2` peuvent être construits en mettant `3`, ou `4` en tête de `(2)`. On obtient ainsi `(3 2)` et `(4 2)`. Enfin, le couple contenant `3`, `(4 3)`, peut être construit en mettant `4` en tête de `(3)`.

On peut construire les triplets à partir des couples en appliquant le même principe : les triplets qui contiennent `(2 1)` peuvent être construits en mettant `3`, ou `4` en tête. Et cetera.

a) Ecrire une fonction `suite` ayant comme arguments une liste `L` non vide de nombres réels, triée par ordre croissant, et un élément `e` de cette liste, et telle que l'évaluation de l'expression `(suite L e)` retourne la liste des éléments de `L` placés après `e`.
Par exemple, l'évaluation de l'expression `(suite '(1 2 3 4) 2)` doit retourner `(3 4)`.

Solution :

```
(define suite (lambda (L e)
   (if (null? L) ()
        (if (= (car L) e) (cdr L) (suite (cdr L) e)) ) ))
```

b) Ecrire une fonction `construction` ayant comme arguments une liste `L` non vide de nombres réels, triée par ordre croissant, et une liste de *k*-uplets d'éléments de `L`, `Lkuplets`, et telle que l'évaluation de l'expression `(construction L Lkuplets)` retourne la liste des (k + 1)-uplets construits à partir des *k*-uplets de `L` conformément à la méthode expliquée ci-dessus.
Par exemple l'évaluation de l'expression

`(construction '(1 2 3 4) '((1) (2) (3) (4)))`

doit retourner `((2 1) (3 1) (4 1) (3 2) (4 2) (4 3))`,
et l'évaluation de l'expression

`(construction '(1 2 3 4) '((2 1) (3 1) (4 1) (3 2) (4 2) (4 3)))`

doit retourner `((3 2 1) (4 2 1) (4 3 1) (4 3 2))`.

Solution :

```
(define construction (lambda (L Lkuplets)
   (append-map (lambda (kuplet)
      (let ((L2 (suite L (car kuplet))))
      (map (lambda (x) (cons x kuplet)) L2) )
                Lkuplets ) ))
```

3°) Fonction `SPa`

a) Ecrire une fonction `SPa2` ayant comme arguments :

- une liste `L` non vide de nombres réels, triée par ordre croissant ;
- un nombre réel `x` ;
- un nombre réel `err` ;
- et une liste de *k*-uplets d'éléments de `L`, `Lkuplets` ;

et telle que l'évaluation de l'expression (SPa2 L x err Lkuplets) retourne un *k*-uplet de L dont la somme *S* des éléments est telle que $| S - x | \leq err$, en appliquant le principe décrit ci-dessus. On pourra utiliser la fonction Scheme prédéfinie length qui retourne la longueur de la liste passée en argument.

<u>Solution</u> :

```
(define SPa2 (lambda (L x err Lkuplets)
   (let ((k (length (car Lkuplets)))
         (n (length L))
         (solution (recherche Lkuplets x err)) )
   (if (null? solution)
      (if (< k n)
          (SPa2 L x err (construction L Lkuplets))
          () )
       solution ) ) ))
```

b) Ecrire la fonction SPa en utilisant la fonction SPa2. On suppose que l'on dispose d'une fonction tri qui permet de trier la liste L.

<u>Solution</u> :

```
(define SPa (lambda (L x err)
   (let ((LT (tri L)))
      (SPa2 LT x err (map list LT))))
```