ISIMA 1ère année — 30 août 2012

Durée : 2 heures

Documents autorisés

PROGRAMMATION FONCTIONNELLE

Exercice 1 :

Ecrire une fonction `precedent` ayant comme arguments une liste `L` et une variable quelconque `x` et telle que l'évaluation de l'expression `(precedent x L)` retourne `()` dans les cas où `x` n'appartient pas à `L` ou bien est le premier élément de `L`, et retourne l'élément qui précède `x` dans `L` sinon.

Par exemple l'évaluation de l'expression `(precedent 'd '(a b c d e))` doit retourner le symbole `c`.

Solution :

```
(define precedent (lambda (x L)
   (if (or (null? L) (null? (cdr L))) ()
       (if (equal? x (cadr L)) (car L)
           (precedent x (cdr L)) ) ) ))
```

Exercice 2 :

Soit `table` une fonction ayant comme argument un opérateur binaire `op` et une liste `L` = (x_1, …, x_n) d'éléments x_k de type compatible avec `op` (par exemple si `op` est la multiplication, les x_k sont des nombres), et telle que l'évaluation de l'expression `(table op L)` retourne la liste de tous les triplets (x_i, x_j, x_i `op` x_j) pour $1 \le i, j \le n$.

Par exemple, `(table * '(1 2 3))` doit retourner

`((1 1 1) (1 2 2) (1 3 3) (2 1 2) (2 2 4) (2 3 6) (3 1 3) (3 2 6) (3 3 9))`.

Ecrire la fonction `table` en utilisant des schémas d'application itératifs de type `map` ou `append_map`.

Solution :

```
(define table (lambda (op L)
```

```
   (append_map (lambda (xi)
       (map (lambda (xj) (list xi xj (op xi xj))) L) )
                L ) ))
```

Exercice 3 :

1°) Ecrire une fonction `maximum` ayant comme arguments une fonction numérique d'une variable numérique, `f`, et une liste non vide de nombres, `L`, du type (x_1, …, x_n), et telle que l'évaluation de l'expression `(maximum f L)` retourne $\max_{i\in\{1,\ldots,n\}} f(x_i)$.

Solution :

Version recursive classique :

```
(define maximum (lambda (f L)
   (if (null? (cdr L))
       (f (car L))
       (max (f (car L)) (maximum f (cdr L))) ) ))
```

Version recursive terminale :

```
(define maximum (lambda (f L) (maxrt f (cdr L) (f (car L)))))
(define maxrt (lambda (f L max)
   (if (null? L) max
       (let ((fxi (f (car L))))
          (if (> fxi max) (maxrt f (cdr L) fxi)
                          (maxrt f (cdr L) max) ) ) ) ))
```

2°) Ecrire une fonction `argmax` ayant comme arguments une fonction numérique d'une variable numérique, `f`, et une liste non vide de nombres, `L`, du type (x_1, …, x_n), et telle que l'évaluation de l'expression `(argmax f L)` retourne $i_{\max} \in \{1, ..., n\}$ tel que $f(x_{i_{\max}}) = \max_{i\in\{1,\ldots,n\}} f(x_i)$.

Solution :

Version recursive classique :

```
(define argmax (lambda (f L)
   (if (null? (cdr L)) 1
      (if (>= (f (car L)) (maximum f (cdr L))) 1
         (+ 1 (argmax f (cdr L))) ) ) ))
```

Version recursive terminale, nettement plus efficace :

```
 (define argmax (lambda (f L)
   (argmaxrt f (cdr L) (f (car L)) 1 2)))
(define argmaxrt (lambda (f L max argmax i)
   (if (null? L) argmax
       (let ((fxi (f (car L))))
          (if (> fxi max) (maxrt f (cdr L) fxi i (+ i 1))
                          (maxrt f (cdr L) max argmax (+ i 1))
           ) ) ) ))
```

Exercice 4 :

On appelle multi-ensemble d'un ensemble *E*, ou collection, un ensemble *M* de couples (e, n_e), où *e* désigne un élément de *E* et n_e le nombre, strictement positif, d'occurrences de *e* dans *E*.

Par exemple une bibliothèque qui contient des livres dont les références sont représentées par les symboles `e1`, `e2`, ..., et qui possède 9 exemplaires de `e1`, 3 de `e2`, ... pourra être représentée par la liste `((e1 9) (e2 3) ... )`.

1°) Ecrire une fonction `ajouter` ayant comme argument un couple `c`, de type (e, n_e), et une liste de couples `Coll`, représentant une collection, et telle que l'évaluation de l'expression `(ajouter c coll)` retourne une liste de couples qui représente la collection obtenue en ajoutant n_e occurrences de *e* dans la collection `Coll`.

Par exemple si la variable `Bib` représente la collection `((e1 9) (e2 3) ... )`, l'appel de l'expression `(ajouter '(e2 4) Bib)` retourne `((e1 9) (e2 7) ... )`, c'est-à-dire la collection obtenue en ajoutant 4 exemplaires du livre `e2` dans la bibliothèque `Bib`.

Remarque : Il se peut que *e* n'appartienne pas à la collection représentée par `Coll` (par exemple si l'on achète de nouveaux livres).

Solution :

```
(define ajouter (lambda (c Coll)
   (if (null? Coll) (list c)
      (if (equal? (car c) (caar Coll))
          (cons (list (caar Coll) (+ (cadar Coll) (cadr c)))
                (cdr Coll) )
          (cons (car Coll) (ajouter c (cdr Coll))) ) ) ))
```

2°) Ecrire, à l'aide d'un schéma de réduction, une fonction `fusion` ayant comme arguments deux listes de couples `M1` et `M2` représentant des collections, et telle que l'évaluation de

l'expression `(fusion M1 M2)` retourne une liste de couples qui représente la collection obtenue en ajoutant toutes les occurrences des éléments de `M1` dans la collection `M2`.

Solution :

Version récursive :

```
(define fusion (lambda (C1 C2)
   (if (null? C1) C2
       (ajouter (car C1) (fusion (cdr C1) C2)) ) ))
```

D'où la solution avec le schéma récursif :

```
(define SR (lambda (L I R)
   (if (null? L) I
       (R (car L) (SR (cdr L) I R)) ) ))
(define fusion (lambda (C1 C2) (SR C1 C2 ajouter))
```

Version itérative :

```
(define fusion (lambda (C1 C2)
   (if (null? C1) C2
       (fusion (cdr C1) (ajouter (car C1) C2)) ) ))
```

D'où la solution avec le schéma itérative :

```
(define SI (lambda (L A C)
      (if (null? L) A
          (SI (cdr L) (C (car L) A) C)) ))
(define fusion (lambda (C1 C2) (SI C1 C2 ajouter))
```