MIAS 1 et MASS 1 – Programmation fonctionnelle

TP n° 5 : corrigé

```
; Exercice 1
; symbnomb : Liste --> Liste
;              l   |-> (0 0) si l est la liste vide
;                      (s+1 n) si l=(x . l1)
;                              et symbnomb(l1)=(s n)
;                              et x est un symbole
;                      (s n+1) si l=(x . l1)
;                              et symbnomb(l1)=(s n)
;                              et x est un nombre
;                      (s n)   si l=(x . l1)
;                              et symbnomb(l1)=(s n)
;                              et x n'est ni un symbole ni un nombre

(define symbnomb
  (lambda (l)
    (if (null? l)
        (list 0 0)
        (let* ((res (symbnomb (cdr l)))
               (s (car res))
               (n (cadr res))
               (x (car l)))
          (cond ((symbol? x) (list (+ s 1) n))
                ((number? x) (list s (+ 1 n)))
                (else res))))))

; Exercice 2
; separe-pair-impair : Liste --> Liste
;                          l   |-> (()()) si l est la liste vide
;                                  ( (x . lp) li ) si l=(x . l1)
;                                                  et separe-pair-impair(l1)=(lp li)
;                                                  et x est pair
;                                  ( lp (x . li) ) si l=(x . l1)
;                                                  et separe-pair-impair(l1)=(lp li)
;                                                  et x est impair

(define separe-pair-impair
  (lambda (l)
    (if (null? l)
        (list '() '())
        (let* ((res (separe-pair-impair (cdr l)))
               (lp (car res))
               (li (cadr res))
               (x (car l)))
          (if (even? x)
              (list (cons x lp) li)
              (list lp (cons x li)))))))

; Exercice 3
; truc : Liste --> Liste
;          l   |-> (0 0)     si l est la liste vide
;                  (p+1 m)   si l=(x . l1)
;                            et truc(l1)=(p m)
;                            et x>0 et x n'est pas multiple de 3
;                  (p+1 m+1) si l=(x . l1)
;                            et truc(l1)=(p m)
;                            et x>0 et x est multiple de 3
;                  (p m)     si l=(x . l1)
;                            et truc(l1)=(p m)
;                            et x<0 et x n'est pas multiple de 3

(define truc
  (lambda (l)
    (if (null? l)
        (list 0 0)
        (let* ((res (truc (cdr l)))
               (p (car res))
               (m (cadr res))
               (x (car l)))
          (cond ((and (> x 0) (= 0 (remainder x 3))) (list (+ p 1) (+ m 1)))
                ((> x 0) (list (+ p 1) m))
                ((= 0 (remainder x 3)) (list p (+ 1 m)))
                (else res))))))

; Exercice 4
; nbatomes : Liste --> Entier
;              l   |-> 0 si l est la liste vide
;                      1+nbatomes(l1) si l=(x . l1) est x est un atome
;                      nbatomes(l1)+nbatomes(l2) si l=(l1 . l2) est l1 est une liste non vide

(define nbatomes
  (lambda (l)
    (cond ((null? l) 0)
          ((atom? (car l)) (+ 1 (nbatomes (cdr l))))
          ((list? (car l)) (+ (nbatomes (car l))
                              (nbatomes (cdr l))))
          (else (nbatomes (cdr l))))))


; Exercice 5
; aplatir : Liste --> Liste
;             l   |-> () si l est la liste vide
;                     aplatir(l1).aplatir(l2) si l=(l1 . l2) et l1 est une liste
;                     (x . aplatir(l1))       si l=(x . l1) et x n'est pas une liste

(define aplatir
  (lambda (l)
    (cond ((null? l) '())
          ((list? (car l)) (append (aplatir (car l)) (aplatir (cdr l))))
          (else (cons (car l) (aplatir (cdr l)))))))


; Exercice 6
; sauf : Objet, Liste --> Liste
;          o  ,   l   |-> () si l est la liste vide
;                         sauf(o, l1) si l=(o . l1)
;                         (sauf(l1) . sauf(l2))  si l=(l1 . l2)
;                                                   et l1<>o est une liste
;                         (x . sauf(l1)) si l=(x . l1)
;                                        et x<>o n'est pas une liste

(define sauf
  (lambda (o l)
    (cond ((null? l) '())
          ((equal? o (car l)) (sauf o (cdr l)))
          ((list? (car l)) (cons (sauf o (car l))
                                 (sauf o (cdr l))))
          (else (cons (car l) (sauf o (cdr l)))))))
```

```
; Exercice 7
; enleve : Liste, Liste --> Liste
;            l1 ,   l2  |-> l2  si l1 est la liste vide
;                           enleve( ll1, sauf(x, l2) ) si l1=(x . ll1)

(define enleve
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (enleve (cdr l1) (sauf (car l1) l2)))))



; Exercice 8
; nblistes : Liste --> Entier
;              l   |-> 0 si l est la liste vide
;                      1+nblistes(l1)+nblistes(l2)  si l=(l1 . l2)
;                                                   et l1 est une liste
;                      nblistes(l1) si l=(x . l1)
;                                   et x n'est pas une liste

(define nblistes
  (lambda (l)
    (cond ((null? l) 0)
          ((list? (car l)) (+ 1 (nblistes (car l))
                              (nblistes (cdr l))))
          (else (nblistes (cdr l))))))



; Exercice 9
; profondeur : Liste --> Entier
;                l   |-> 1 si l est la liste vide
;                        max(1+profondeur(l1), profondeur(l2)) si l= (l1 . l2)
;                                                              et l1 est une liste
;                        profondeur(l1) si l=(x . l1) et x n'est pas une liste

(define profondeur
  (lambda (l)
    (cond ((null? l) 1)
          ((list? (car l)) (max (+ 1 (profondeur (car l)))
                                (profondeur (cdr l))))
          (else (profondeur (cdr l))))))



; Exercice 10
; infixprefix : ExprArithm --> ExprArithm
;                    e      |-> e si e est un symbole ou un nombre
;                               (op infixprefix(e1) infixprefix(e2)) si e=(e1 op e2)

(define infixprefix
  (lambda (e)
    (if (or (symbol? e) (number? e))
        e
        (list (cadr e)
              (infixprefix (car e))
              (infixprefix (caddr e))))))
```

```
; Exercice 11
; par rapport a la fonction de l'exercice 2,
; il faut prendre en compte le cas ou le premier element de la liste
; est une liste, auquel cas il faut appeler la fonction sur cette liste
; et le cas ou le premier element n'est ni une liste, ni un entier

(define separe-pair-impair2
  (lambda (l)
    (if (null? l)
        (list '() '())
        (let* ((res (separe-pair-impair2 (cdr l)))
               (lp (car res))
               (li (cadr res))
               (x (car l)))
          (cond ((list? x) (let* ((resx (separe-pair-impair2 x))
                                  (lpx (car resx))
                                  (lix (cadr resx)))
                             (list (append lpx lp)
                                   (append lix li))))
                ((not (integer? x)) res)
                ((even? x) (list (cons x lp) li))
                (else (list lp (cons x li)))))))))
```