MIAS 1 et MASS 1 – Programmation fonctionnelle

TP n° 3: corrigé

```
; Exercice 1

(cons 1 (cons 2 '()))
(cons (cons 1 '()) (cons 2 '()))
(cons 1 (cons (cons 2 3) '()))
(cons '() (cons (cons (cons 1 2) (cons 3 '()))  '()))
(cons 1 (cons 2 (cons 3 (cons (cons (cons 4 '()) '()) '()))))


; Exercice 2

(cadr (cadadr '(chou (genou (caillou joujou)))))
(caaar '(((joujou) hibou)))
(caaddr '(chou genou (joujou) caillou))
(caadr (cadadr '(chou (genou (hibou (joujou))))))
(cadr (cadadr '(chou (genou (hibou joujou)))))
(cdr (cadadr '(chou (genou (hibou . joujou)))))


; Exercice 3
; second : Liste --> Objet
;             l  |-> b    si l=(a b . l')

(define second
  (lambda (l) (cadr l)))


; Exercie 4
; echange : Liste --> Objet
;             l   |-> (b a . l')  si l=(a b . l')

(define echange
  (lambda (l)
    (cons (cadr l)
          (cons (car l)(cddr l)))))


; Exercice 5
; truc : Reel --> Liste
;          x  |-> (le carre de x est y) avex y=x*x

(define truc
  (lambda (x) (list 'le 'carre 'de x 'est (* x x))))


; Exercice 6

(define question
  (lambda (l)
    (cons (echange l) (cons '? '()))))
```

```
; Exercice 7
; deux? : Objet --> Booleen
;           o   |-> vrai si o est une liste d'exactement 2 elements
;
; une liste d'exactement 2 elements est une paire pointee dont
; le cdr est une paire pointee et le cddr est la liste vide

(define deux?
  (lambda (o)
    (and (pair? o)
         (pair? (cdr o))
         (null? (cddr o)))))


; Exercice 8
; minmax : Liste   --> Liste
;            l     |-> (x y) si l=(a b c) et x=min(a,b,c) et y=max(a,b,c)

(define minmax
  (lambda (l)
    (let* ((a (car l))
           (b (cadr l))
           (c (caddr l))
           (min (cond ((and (<= a b) (<= a c)) a)
                      ((<= b c) b)
                      (else c)))
           (max (cond ((and (>= a b) (>= a c)) a)
                      ((>= b c) b)
                      (else c))) )
      (list min max))))


; Exercice 9
; On represente les fractions rationnelles a l'aide de paires pointees
; d'entiers de la forme (nominateur . denominateur).
; Dans les specifications des fonctions qui suivent les parametres
; f1 et f2 sont des fractions qu'on suppose representees respectivement
; par les paires pointees d'entiers f1=(a . b) et f2=(c . d)

; Somme de deux fractions
; +fraction :  Fraction,  Fraction  -->   Fraction
;                f1    ,    f2      |->   f1+f2 = (ad+bc . bd)

(define +fraction
  (lambda (f1 f2)
    (cons (+ (* (car f1) (cdr f2))
             (* (cdr f1) (car f2)))
          (* (cdr f1) (cdr f2)))))

; Difference de deux fractions
; -fraction : Fraction, Fraction  -->   Fraction
;               f1    ,   f2      |-> f1-f2=(ad-bc . bd)

(define -fraction
  (lambda (f1 f2)
    (cons (- (* (car f1) (cdr f2))
             (* (cdr f1) (car f2)))
          (* (cdr f1) (cdr f2)))))
```

```
; Produit de deux fractions
; *fraction : Fraction, Fraction  -->    Fraction
;               f1    ,   f2(     |->  f1*f2=(ac . bd)

(define *fraction
  (lambda (f1 f2)
    (cons (* (car f1) (car f2))
          (* (cdr f1) (cdr f2)))))

; Quotient de deux fractions
; qfraction : Fraction, Fraction  -->    Fraction
;               f1    ,   f2(     |->  f1/f2=(ad . bc)

(define qfraction
  (lambda (f1 f2)
    (cons (* (car f1) (cdr f2))
          (* (cdr f1) (car f2)))))

; Egalite de deux fractions : deux fractions a/b et c/d sont egales si ad=bc
; =fraction? : Fraction, Fraction --> Booleen
;                f1    ,   f2     |-> vrai si ad=bc, faux sinon

(define =fraction?
  (lambda (f1 f2)
    (= (* (car f1) (cdr f2))
       (* (cdr f1) (car f2)))))


; Exercice 10
; 2nd-degre : Reel, Reel, Reel --> Liste
;              a  ,  b  ,  c   |-> la liste des solutions reelles de ax^2+bx+c=0

(define 2nd-degre
  (lambda (a b c)

    ; on definit une fonction locale pour resoudre le cas
    ; ou a = 0
    (define premier-degre
      (lambda (b c)
        (if (= 0 b)
            (if (= 0 c)
                (list 'infini)
                '())
            (list  (- (/ c b))) )))

    ; corps de la fonction second-degree
    (if (= a 0)
        (premier-degre b c)
        ; on definit une variable locale : delta
        (let ((delta (- (* b b) (* 4 a c))))
          (cond ((< delta 0) '())
                ((= 0 delta) (list (- (/ b (* 2 a)))) )
                (else (list (/ (- (* b b) (sqrt delta)) (* 2 a))
                            (/ (+ (* b b) (sqrt delta)) (* 2 a))))
                )))))
```