Université de La Réunion septembre 2000

DEUG 1^{ere} année – MIAS 1 et MASS 1

UE4 – Informatique – 2^{e} session

Durée : 2 heures – Aucun document n'est autorisé.

Répondre uniquement dans les cadres prévus à cet effet

Nom :	N° étudiant :	Signature
Prénom(s) :		
Date de naissance :		

Partie 1 : Échauffements

(3 pts) **Exercice 1.1 :** Écrivez en scheme une fonction `2nd-degre` qui prend en paramètres les coefficients d'une équation du second degré $ax^2 + bx + c = 0$, et qui renvoie la liste des racines **réelles** de cette équation. Cette fonction doit prendre en compte tous les cas (entre autre si $a = 0$ on est ramené à une équation du premier degré).

```
(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)
        (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)))))))))
```

(1 pt) **Exercice 1.2 :** Écrivez en scheme une fonction `pardeux` qui groupe les éléments d'une liste plate par deux.
Exemple : `(pardeux '(a 2 4 f h 5 x))` renvoie `((a 2)(4 f)(h 5)(x))`

```
(define pardeux
  (lambda (l)
    (cond ((null? l) '())
          ((null? (cdr l)) (list l))
          (else (cons (list (car l) (cadr l))
                      (pardeux (cddr l)))))))
```

(1 pt) **Exercice 1.3 :** Écrivez en scheme un prédicat `tous-egaux?` qui prend une liste plate et qui est vrai si tous les éléments de la liste sont les mêmes.

```
(define tous-egaux?
  (lambda (l)
    (or (null? l)
        (null? (cdr l))
        (and (equal? (car l) (cadr l))
             (tous-egaux? (cdr l))))))
```

(1 pt) **Exercice 1.4 :** Écrivez en scheme une fonction `aplatir` qui prend une liste quelconque l et renvoie une liste plate formée des atomes de l.

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

(1 pt) **Exercice 1.5 :** Écrivez en scheme une fonction **récursive terminale** qui permet de renverser une liste plate.

```
(define renverse
  (lambda (l)
    (define aux
      (lambda (l1 r)
        (if (null? l1)
            r
            (aux (cdr l1) (cons (car l1) r)))))
    (aux l '())))                                                       .
```

(1 pt) **Exercice 1.6 :** Écrivez en scheme une fonction **récursive terminale** qui permet de calculer x^n (on supposera que $x \neq 0$ et $n \geq 0$).

```
(define puissance
  (lambda (x n)
    (define aux
      (lambda (k xk)
        (if (= k n)
            xk
            (aux (+ k 1) (* xk x)))))
    (if (= 0 n)
        1
        (aux 1 x))))                                                    .
```

(2 pts) **Exercice 1.7 :** Écrivez en scheme une fonction `separe-pair-impair` qui prend une liste plate d'entiers et renvoie une liste formée de la liste des nombres pairs et de la liste des nombres impairs. **Cette fonction ne doit effectuer qu'un seul parcourt de la liste.**
Exemple `(separe-pair-impair '(1 2 3 4 5 6 7))` renvoie `((2 4 6)(1 3 5 7))`

```
(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)))))))                                .
```

Partie 2 : Tri par sélection

Le tri par sélection d'une liste `l` d'entiers consiste à sélectionner le plus petit entier `m` de `l` (ou l'un des plus petits entiers si `m` apparaît plusieurs fois), et à le placer au début de la liste obtenue en triant par sélection la liste `l` privée de `m`.

(1 pt) **Exercice 2.1 :** Écrivez en scheme une fonction `selectmin` qui renvoie le plus petit élément d'une liste d'entiers.
Exemple : `(selectmin '(4 7 8 2 3 5 9))` renvoie `2`

```
(define selectmin
  (lambda (l)
    (cond ((null? (cdr l)) (car l))
          ((< (car l) (cadr l)) (selectmin (cons (car l) (cddr l))))
          (else (selectmin (cdr l))))))
```

(1 pt) **Exercice 2.2 :** Écrivez en scheme une fonction `otemin` qui prend une liste d'entiers et renvoie cette liste privée de son plus petit élément (ou d'un de ses plus petits éléments). On pourra utiliser la fonction `selectmin`.
Exemple : `(otemin '((4 7 8 2 3 5 9))` renvoie `(4 7 8 3 5 9)`

```
(define otemin
  (lambda (l)
    (define ote-m
      (lambda (m l)
        (cond ((= m (car l)) (cdr l))
              (else (cons (car l)
                          (ote-m m (cdr l)))))))
  (ote-m (selectmin l) l)))
```

(2 pts) **Exercice 2.3 :** Écrivez une fonction `select-ote` qui prend une liste `l` d'entiers et renvoie le plus petit élément et la liste `l` privée de ce plus petit élément. Cette fonction ne doit effectuer qu'**un seul** parcours de `l` et ne doit bien evidemment **pas** faire appel aux fonctions `selectmin` et `otemin`.
Exemple : `(select-sup '(4 7 8 2 3 5 9))` renvoie `(2 (4 7 8 3 5 9))`

```
(define select-ote
  (lambda (l)
    (if (null? (cdr l))
        (list (car l) '() )
        (let* ((res (select-ote (cdr l)))
               (m (car res))
               (rl (cadr res)))
          (if (< (car l) m)
              (list (car l)
                    (cons m rl))
              (list m
                   (cons (car l) rl)))))))
```

(1 pt) **Exercice 2.4 :** Écrivez en scheme une foncfion `tri-selection` qui prend une liste d'entiers et renvoie une liste, contenant les mêmes entiers, triée en utilisant le principe du tri par selection.

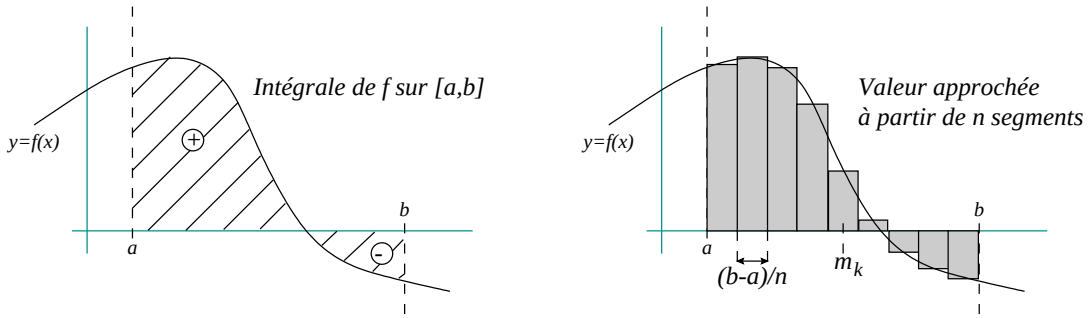
```
(define tri-selection
  (lambda (l)
    (cond ((null? l) l)
          (else (let ((res (select-ote l)))
                  (cons (car res)
                        (tri-selection (cadr res))))))))
```

Partie 3 : Ordre supérieur

(2 pts) **Exercice 3.1 :** Spécifiez et écrivez en scheme une fonction `somme` qui permet de calculer $\sum_{i=1}^{n} f(i)$, f étant une fonction entière quelconque.

```
; somme : Fonction, Entier --> Reel
;             f    ,   n    |-> f(1) si n=1
;                              f(n)+somme(f, n-1)   si n>1
     (define somme
        (lambda (f n)
           (if (= n 1)
               (f 1)
               (+ (f n) (somme f (- n 1))))))                                .
```

(3 pts) **Exercice 3.2 :** On peut obtenir une valeur approchée de l'intégrale d'une fonction réelle (supposée intégrable) f sur un intervalle $[a, b]$ en calculant l'aire algébrique de la surface délimitée par l'axe des abscices, la courbe représentative de f et les droites $y = a$ et $y = b$, c'est à dire l'aire hachurée du dessin ci-dessous à gauche. Calculer une valeur approchée de $\int_a^b f$ revient donc à calculer une valeur approchée de cette aire.



Pour calculer une valeur approchée de l'aire, on découpe l'intervalle $[a, b]$ en n segments de longueur $(b - a)/n$ et on calcule la somme des surfaces de n rectangles dont les largeurs sont égales à $(b - a)/n$ et les hauteurs sont données par les valeurs de f au milieu de chaque segment (dessin de droite).
L'abscice du milieu du k^{e} segment est $m_k = a + (k - 1)\frac{b-a}{n} + \frac{1}{2}\frac{b-a}{n}$ (abscice du début du k^{e} segment augmentée de la moitié de la longueur d'un segment).
L'aire du k^{e} rectangle A_k est donc $A_k = \frac{b-a}{n} \times f(m_k)$
Pour un nombre de rectangles n, la valeur approchée de l'intégrale $\int_a^b f$ est donc donnée par la somme :

$$\sum_{k=1}^{n} A_k = \frac{b-a}{n} \times \sum_{k=1}^{n} f(m_k)$$

Écrivez une fonction `integrale` qui prend comme paramètres une fonction `f` et un nombre de rectangles `n` et renvoie comme résultat **une fonction** qui calcule la valeur approchée de l'intégrale de `f` entre deux bornes `a` et `b`.
La spécification de cette fonction est :

$$\begin{array}{lccc} \textit{integrale :} & \text{Fonction}, & \text{Entier} & \longrightarrow & \text{Fonction} \\ & f & n & \longmapsto & a, b \longmapsto \sum_{k=1}^{n} A_k \end{array}$$

```
(define integrale
  (lambda (f n)
    (lambda (a b)
      (let ((seg (/ (- b a) n)))
        (* seg
          (somme (lambda (k) (f (+ a (* (/ (- (* 2 k) 1) 2) seg)) ) )
                 n))))))                                                   .
```