Groupe	Nom	Prénom

Devoir sur table - Novembre 2011

LI101

Durée 1h30

Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.

Le sujet comporte 11 pages. Ne pas désagrafer les feuilles.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le barème apparaissant dans chaque boîte n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : 46 points.

La clarté des réponses et la présentation des programmes seront appréciées. Les questions peuvent être résolues de façon indépendante. Il est possible, voire utile, pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.

Pour vous faire gagner du temps, il ne vous est pas systématiquement demandé, en plus de la définition, la spécification entière d'une fonction. Bien lire ce qui est demandé : seulement la définition ? la signature et la définition ? la spécification et la définition ? seulement la spécification ?

Lorsque la signature d'une fonction vous est demandée, vous veillerez à bien préciser, avec la signature, les éventuelles hypothèses sur les valeurs des arguments.

Exercice 1

On rappelle qu'un entier naturel non nul est dit *premier* si et seulement si ses seuls diviseurs sont 1 et lui-même. On rappelle aussi que par convention cependant, 1 n'est pas un nombre premier. Le but de cet exercice est de définir un prédicat testant si un entier naturel est un nombre premier.

Question 1.1

Donnez la signature et une définition de la fonction `intervalle` qui, étant donnés deux entiers naturels n et p, construit la liste contenant les entiers naturels compris entre n et p inclus en ordre croissant. On supposera pour cela que n est inférieur ou égal à p. Par exemple :

`(intervalle 2 6)` → `(2 3 4 5 6)`

`(intervalle 3 3)` → `(3)`

Réponse. [3/46]

```
;;; intervalle : nat * nat -> LISTE[nat]
;;; (intervalle n p) renvoie la liste des entiers compris entre n et p inclus
;;; HYPOTHESE : n<=p
(define (intervalle n p)
  (if (= n p)
      (list n)
      (cons n (intervalle (+ n 1) p ))))
```

Question 1.2

Donnez une définition du prédicat `est-diviseur?: nat * nat -> bool` telle que `(est-diviseur? n p)` renvoie `#t` si et seulement si n est un diviseur de p. On fera l'hypothèse que n est un entier non nul. Par exemple :

Groupe	Nom	Prénom

`(est-diviseur? 2 4)` → `#t`

`(est-diviseur? 4 2)` → `#f`

`(est-diviseur? 2 3)` → `#f`

Réponse. [2/46]

```
;;; est-diviseur?: nat * nat -> bool
;;; (est-diviseur? n p) renvoie #t ssi n est un diviseur de p
;;; HYPOTHESE : n non nul
(define (est-diviseur? n p)
  (= (remainder p n) 0)
)
```

Question 1.3

À l'aide de la fonction précédente, écrivez la signature et une définition de la fonction `recup-diviseurs` qui, étant donnés un entier n et une liste d'entiers L, renvoie la liste des éléments de L qui sont des diviseurs de n. On supposera que la liste L ne contient pas 0. Par exemple :

`(recup-diviseurs 13 '())` → `()`

`(recup-diviseurs 13 '(3 1 8))` → `(1)`

`(recup-diviseurs 20 '(1 2 3 4 5 6 7 8 9))` → `(1 2 4 5)`

`(recup-diviseurs 2 '(3 4 5 6 7))` → `()`

Vous pourrez donner soit une définition récursive soit utiliser une fonctionnelle.

Réponse. [3/46]

```
;;; recup-diviseurs: nat * Liste[nat] -> Liste[nat]
;;; (recup-diviseurs n L) renvoie la liste des éléments de L divisant n
;;; HYPOTHESE : L ne contient pas 0
(define (recup-diviseurs n L)
  (if (pair? L)
      (if (est-diviseur? (car L) n)
  (cons (car L) (recup-diviseurs n (cdr L)))
  (recup-diviseurs n (cdr L)))
      (list)))
```

Autre réponse (fonctionnelle) :

```
;;; recup-diviseurs2: nat * Liste[nat] -> Liste[nat]
;;; (recup-diviseurs2 n L) renvoie la liste des éléments de L divisant n
;;; HYPOTHESE : L ne contient pas 0
(define (recup-diviseurs2 n L)
  (define (f x)
    (est-diviseur? x n)
    )
  (filter f L)
)
```

Question 1.4

Toujours à l'aide des fonctions précédentes, donnez la signature et une définition de la fonction `liste-diviseurs` qui, étant donné un entier naturel n non nul, renvoie la liste de ses diviseurs.

Groupe	Nom	Prénom

Indication : Les diviseurs de n sont les entiers compris entre 1 et n qui divisent n.

`(liste-diviseurs 12)` → `(1 2 3 4 6 12)`

`(liste-diviseurs 17)` → `(1 17)`

`(liste-diviseurs 1)` → `(1)`

Réponse. [2/46]

```
;;; liste-diviseurs: nat -> Liste[nat]
;;; (liste-diviseurs n) renvoie la liste des diviseurs de n
;;; HYPOTHESE : n non nul
(define (liste-diviseurs n)
  (recup-diviseurs n (intervalle 1 n))
)
```

Question 1.5

Donnez la signature et une définition du prédicat `liste-taille2?` qui teste si une liste L a exactement 2 éléments. Une attention particulière sera portée sur l'efficacité de la fonction.

`(liste-taille2? '())` → `#f`

`(liste-taille2? '("je" "tu" "il"))` → `#f`

`(liste-taille2? '("je" "tu"))` → `#t`

`(liste-taille2? '(#t))` → `#f`

`(liste-taille2? '(1 2 3))` → `#f`

`(liste-taille2? '(-1 7))` → `#t`

Réponse. [2/46]

```
;;; liste-taille2?: Liste[alpha] -> bool
;;; (liste-taille2? L) renvoie #t ssi L contient exactement 2 éléments
(define (liste-taille2? L)
  (and (pair? L) (pair? (cdr L)) (not (pair? (cddr L)))))
```

Question 1.6

En utilisant certaines des fonctions définies précédemment, donnez une définition du prédicat `est-premier?: nat -> bool` qui, étant donné un entier naturel n non nul, renvoie `#t` si n est un nombre premier.

`(est-premier? 12)` → `#f`

`(est-premier? 17)` → `#t`

`(est-premier? 1)` → `#f`

Réponse. [1/46]

```
;;; est-premier?: nat -> bool
;;; (est-premier? n) teste si n est premier ou non
;;; HYPOTHESE : n non nul
(define (est-premier? n)
  (liste-taille2? (liste-diviseurs n)))
```

Groupe	Nom	Prénom

Exercice 2

On appelle décomposition d'un entier naturel n strictement positif en q sommants, avec $q > 0$, une somme de q termes non nuls dont la valeur est n.

Par exemple, 5 est la décomposition de 5 en 1 sommant, $1+4$ et $2+3$ sont les décompositions de 5 en 2 sommants, $1+2+2$, $1+1+3$ sont les décompositions de 5 en 3 sommants. L'ensemble de ces décompositions représentent les décompositions de l'entier 5 en au plus 3 sommants.

On peut calculer le nombre de décompositions d'un naturel strictement positif n en au plus q sommants, avec $q > 0$, noté $d(n, q)$ dans la suite, à l'aide de la relation suivante :

$$d(n,q) = \begin{cases} 1 & \text{si } q = 1 \text{ ou } n = 1 \\ d(n, n-1) + 1 & \text{si } 1 < n \leq q \\ d(n, q-1) + d(n-q, q) & \text{sinon} \end{cases}$$

Question 2.1

Calculez la valeur de $d(5,3)$ à l'aide de la relation ci-dessus et donnez le nombre de calculs récursifs nécessaires à l'évaluation de $d(5,3)$.

Réponse. [2/46]

Il y a 5 décompositions comme donné en exemple ci-dessus.
Voici les calculs :

$$\begin{aligned} d(5,3) &= d(5,2) + d(2,3) \\ &= d(5,1) + d(3,2) + d(2,1) + 1 \\ &= 1 + d(3,1) + d(1,2) + 1 + 1 \\ &= 1 + 1 + 1 + 1 + 1 \\ &= 5 \end{aligned}$$

Il y a 7 appels récursifs.

Question 2.2

Donnez la signature et une définition de la fonction `nb-decomp` telle que `(nb-decomp n q)` calcule la valeur de $d(\mathtt{n}, \mathtt{q})$.

Réponse. [4/46]

```
;;;nb-decomp: nat * nat -> nat
;;;(nb-decomp n p) rend le nombre de decompositions de n en au plus q sommants
;;; HYPOTHESE : n et p sont strictement positifs
(define (nb-decomp n q)
  (cond
    ((= q 1) 1)
    ((= n 1) 1)
    ((>= q n) (+ (nb-decomp n (- n 1)) 1))
    (else (+ (nb-decomp n (- q 1)) (nb-decomp (- n q) q)))))
```

Groupe	Nom	Prénom

Exercice 3

Question 3.1

Donnez la signature et une définition de la fonction `min-liste` qui, étant donnée une liste non vide L de nombres, renvoie le minimum de L.

`(min-liste '(1))` → `1`

`(min-liste '(4 1 6))` → `1`

`(min-liste '(2.0 6.6 -3.5 0.1))` → `-3.5`

Réponse. [3/46]

```
;;; min-liste: LISTE[Nombre] -> Nombre
;;; (min-liste L) renvoie le plus petit element de L
;;; HYPOTHESE : L non vide
(define (min-liste L)
  (if (pair? (cdr L))
      (min (car L) (min-liste (cdr L)))
      (car L)))
```

On supposera désormais définie la fonction duale `max-liste` qui renvoie le maximum d'une liste non vide de nombres.

Question 3.2

Donnez une définition de la fonction `ecart-max: LISTE[Nombre] -> Nombre` qui, étant donnée une liste non vide L de nombres, renvoie l'écart entre le plus petit et le plus grand élément de L

`(ecart-max '(1))` → `0`

`(ecart-max '(4 1 6))` → `5`

`(ecart-max '(2.0 6.6 -3.5 0.1))` → `10.1`

Réponse. [1/46]

```
;;; ecart-max: LISTE[Nombre] -> Nombre
;;; (ecart-max L) renvoie l'ecart entre le plus petit et le plus grand element de L
;;; HYPOTHESE : L non vide
(define (ecart-max L)
  (- (max-liste L) (min-liste L)))
```

Question 3.3

Que pensez-vous de l'efficacité de la fonction `ecart-max` ? Peut-on faire mieux ? Justifiez.

Réponse. [1/46]
Complexité linéaire en la taille de la liste. Mais on fait 2 parcours de liste. On peut faire mieux en utilisant les couples.

On veut maintenant proposer une autre définition de la fonction `ecart-max` en se basant sur l'utilisation de couples de nombres.

Groupe	Nom	Prénom

Question 3.4

Donnez la signature et une définition de la fonction `min-max-liste` qui, étant donnée une liste non vide L de nombres, renvoie le couple (a, b) dans lequel a est le minimum et b le maximum de L. Votre fonction ne devra faire qu'un seul parcours de la liste L.

`(min-max '(1))` → `(1 1)`

`(min-max '(4 1 6))` → `(1 6)`

`(min-max '(2.0 6.6 -3.5 0.1))` → `(-3.5 6.6)`

Réponse. [5/46]

```
;;; min-max: LISTE[Nombre] -> COUPLE[Nombre Nombre]
;;; (min-max L) renvoie le couple (a,b) où a est le minimum et b le maximum de L
;;; HYPOTHESE : L non vide
(define (min-max L)
  (if (pair? (cdr L))
      (let* ((res-rec (min-max (cdr L)))
     (m (car res-rec))
     (M (cadr res-rec))
     (e (car L)))
(list (min e m) (max e M)))
      (list (car L) (car L))))
```

Question 3.5

Donnez une définition de la fonction `ecart-max2` de même spécification que la fonction `ecart-max` mais qui utilise la fonction `min-max`.

`(ecart-max2 '(1))` → `0`

`(ecart-max2 '(4 1 6))` → `5`

`(ecart-max2 '(2.0 6.6 -3.5 0.1))` → `10.1`

Réponse. [2/46]

```
;;; ecart-max2: LISTE[Nombre] -> Nombre
;;; (ecart-max2 L) renvoie l'ecart entre le plus petit et le plus grand element de L
;;; HYPOTHESE : L non vide
(define (ecart-max2 L)
  (let* ((res (min-max L))
 (a (car res))
 (b (cadr res)))
    (- b a)))
```

On s'intéresse maintenant à la normalisation d'une liste de nombres qui revient à normaliser chacun des nombres de la liste.
Soit a et b deux nombres tels que $a < b$. La fonction de normalisation d'un nombre x par a et b (avec $a \leq x$ et $x \leq b$) est définie par :

$$f(x) = \frac{x - a}{b - a}$$

Groupe	Nom	Prénom

Question 3.6

Proposez une définition de la fonction `normalise: Nombre * Nombre * Nombre -> Nombre` telle que `(normalise a b x)` calcule la valeur de `x` normalisée par `a` et `b`. On supposera que $a \leq x \leq b$ et $a \neq b$.

`(normalise 1 10 1)` → `0`

`(normalise 1 10 10)` → `1`

`(normalise 1 10 8)` → `7/9`

`(normalise 1 10 2)` → `1/9`

Réponse. [1/46]

```
;;; normalise: Nombre * Nombre * Nombre -> Nombre
;;; (normalise a b x) calcule la valeur de x normalisée par a et b
;;; HYPOTHESE : a <= x <= b et a/=b
(define (normalise a b x)
  (/ (- x a) (- b a)))
```

Question 3.7

En s'appuyant sur la fonction précédente, donnez une définition de la fonction `normalise-liste: Nombre * Nombre * LISTE[Nombre] -> LISTE[Nombre]` telle que `(normalise-liste a b L)` renvoie la liste des éléments de `L` normalisés par `a` et `b`. On supposera que tous les éléments de `L` sont compris entre `a` et `b` et que les nombres `a` et `b` sont différents.

Vous pourrez donner soit une définition récursive soit utiliser une fonctionnelle.

`(normalise-liste 1 10 (list))` → `()`

`(normalise-liste 1 10 (list 1 2 8 10))` → `(0 1/9 7/9 1)`

Groupe | Nom | Prénom

Réponse. [3/46]

```
;;; normalise-liste: Nombre * Nombre * LISTE[Nombre] -> LISTE[Nombre]
;;; (normalise-liste a b L) renvoie la liste des éléments de L  normalisées par a et b
;;; HYPOTHESE : a <= x <= b pour tout x de L et a /=b
(define (normalise-liste a b L)
  (if (pair? L)
      (cons (normalise a b (car L))
    (normalise-liste a b (cdr L)))
      (list)))
```

Autre réponse (définition interne) :

```
;;; normalise-liste2: Nombre * Nombre * LISTE[Nombre] -> LISTE[Nombre]
;;; (normalise-liste2 a b L) renvoie la liste des éléments de L  normalisées par a et b
;;; HYPOTHESE : a <= x <= b pour tout x de L et a /=b
(define (normalise-liste2 a b L)
  (define (normalise-liste-ab L)
    (if (pair? L)
(cons (normalise a b (car L))
      (normalise-liste-ab (cdr L)))
(list)))
  (normalise-liste-ab L))
```

Autre réponse (fonctionnelle) :

```
;;; normalise-liste3: Nombre * Nombre * LISTE[Nombre] -> LISTE[Nombre]
;;; (normalise-liste3 a b L) renvoie la liste des éléments de L  normalisées par a et b
;;; HYPOTHESE : a <= x <= b pour tout x de L et a /=b
(define (normalise-liste3 a b L)
  (define (f x)
    (normalise a b x))
  (map f L))
```

Exercice 4

Question 4.1

Écrivez la signature et une définition de la fonction `nb-diff` qui étant données deux listes `L1` et `L2` rend le nombre d'éléments de `L1` et de `L2` de même rang qui sont différents. On supposera que `L1` et `L2` ont le même nombre d'éléments.

`(nb-diff '() '())` → `0`

`(nb-diff '("a" "b" "c") '("A" "B" "C"))` → `3`

`(nb-diff '("a" "b" "c") '("a" "B" "C"))` → `2`

`(nb-diff '(1 2 3) '(4 5 6))` → `3`

`(nb-diff '(4 5 6) '(5 6 7))` → `3`

`(nb-diff '(4 5 6) '(6 5 4))` → `2`

`(nb-diff '(7 8 9 10 11) '(7 8 9 10 11))` → `0`

Groupe | Nom | Prénom

Réponse. [4/46]

```
;;; nb-diff : LISTE[Alpha] * LISTE[Alpha] -> nat
;;;(nb-diff L1 L2) rend le nombre de différences entre L1 et L2
;;; HYPOTHESE : L1 et L2 sont de même taille
(define (nb-diff L1 L2)
  (if (pair? L1)
      (if (not (equal? (car L1) (car L2)))
          (+ 1 (nb-diff (cdr L1) (cdr L2)))
          (nb-diff (cdr L1) (cdr L2)))
      0))
```

Question 4.2

Écrivez la signature et une définition de la fonction `insertion-avant` qui, étant donnés deux éléments `x` et `y` et une liste `L`, rend une liste `L` dans laquelle `x` a été ajouté devant la première occurrence de `y` si `y` apparaît dans `L`, à la fin sinon.

`(insertion-avant "a" "z" '("b" "c" "z" "d" "z"))` → `("b" "c" "a" "z" "d" "z")`

`(insertion-avant "a" "z" '("z" "z"))` → `("a" "z" "z")`

`(insertion-avant "a" "y" '("b" "c" "z"))` → `("b" "c" "z" "a")`

`(insertion-avant "a" "y" '())` → `("a")`

`(insertion-avant 1 2 '(5 4 3 2 1))` → `(5 4 3 1 2 1)`

Réponse. [3/46]

```
;;; insertion-avant : Alpha * Alpha * LISTE[Alpha] -> LISTE[Alpha]
;;; (insertion-avant x y L) rend la liste L dans laquelle x a été ajouté devant la premiere
;;; occurrence de y si y apparaît dans L, à la fin de L sinon
(define (insertion-avant x y L)
  (if (pair? L)
      (if (equal? (car L) y)
          (cons x L)
          (cons (car L) (insertion-avant x y (cdr L))))
      (list x)))
```

Question 4.3

Écrivez la signature et une définition de la fonction `suppr-nieme` qui étant donnés une liste `L` et `n` un entier naturel strictement positif rend la liste `L` privée de son `n`-ième élément si `L` contient au moins `n` éléments, `L` sinon.

`(suppr-nieme '("a" "b" "c" "d") 1)` → `("b" "c" "d")`

`(suppr-nieme '("a" "b" "c" "d") 2)` → `("a" "c" "d")`

`(suppr-nieme '("a" "b" "c" "d") 6)` → `("a" "b" "c" "d")`

`(suppr-nieme '() 2)` → `()`

`(suppr-nieme '(0 1 2 3 4 5 6) 4)` → `(0 1 2 4 5 6)`

Groupe | Nom | Prénom

Réponse. [4/46]

```
;;; suppr-nieme : LISTE[Alpha] * nat -> LISTE[Alpha]
;;; (suppr-nieme L n) rend la liste L privée de son n-ieme élément s'il existe, L sinon
;;; HYPOTHESE : n est strictement positif
(define (suppr-nieme L n)
  (if (pair? L)
      (if (= n 1)
          (cdr L)
          (cons (car L) (suppr-nieme (cdr L) (- n 1))))
      L))
```