Université Pierre et Marie Curie — Année 2001-2002
DEUG SPI seconde année — **Examen juin**
Option informatique — 13 juin 2002
2 heures

Remarques :

- *Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.*
- *Les questions peuvent être résolues de façon indépendante. Il est possible, voire même utile, pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.*
- *La clarté des réponses et la présentation des programmes seront appréciées. Toutes les fonctions apparaissant dans les réponses doivent être accompagnées de leur spécification.*
- *Le barême (total sur 20) n'est donné qu'à titre indicatif.*

Question 1 (1 point) :

Écrire une spécification de la fonction `premier-dans-reste` qui a comme définition Scheme :

```
(define (premier-dans-reste L)
  (member (car L) (cdr L)))
```

Donner aussi des exemples d'applications de cette fonction (avec les valeurs renvoyées).

```
;;; premier-dans-reste : LISTE[alpha] -> LISTE[alpha] + #f
;;; ERREUR lorsque la liste donnée est vide
;;; (premier-dans-reste L) renvoie le suffixe de "(cdr L)" débutant par
;;; "(car L)" ou #f lorsque "(car L)" n'apparaît pas dans "(cdr L)".
;;; Par exemple (premier-dans-reste '(4 5 4 2)) renvoie (4 2), (premier-dans-reste '(4 5 4)) renvoie (4),
;;; (premier-dans-reste '(4 5)) renvoie #f et (premier-dans-reste '(4)) renvoie #f.
```

Question 2 (2 points) :

Écrire une définition Scheme de la fonction `existe-doublon?` qui a comme spécification :

```
;;; existe-doublon? : LISTE[alpha] -> bool
;;; (existe-doublon? L) renvoie #t ssi il existe deux éléments de la liste "L" qui sont égaux.
;;; Par exemple, (existe-doublon? '(3 5 6 7 8 6)) renvoie #t (car 6 est un doublon)
;;; et (existe-doublon? '(3 5 6 7 8)) renvoie #f.
```

```
(define (existe-doublon? L)
  (if (pair? L)
      (or (premier-dans-reste L)      ; en fait renvoie le suffixe commençant à
          (existe-doublon? (cdr L)))  ; la seconde occurrence du premier doublon
      #f))
```

Question 3 (1 point) :

Écrire une définition Scheme de la fonction `cons-L` qui a comme spécification :

```
;;; cons-L : alpha * LISTE[LISTE[alpha]] -> LISTE[LISTE[alpha]]
;;; (cons-L e L) renvoie la liste obtenue en ajoutant la liste qui
;;; contient (uniquement) l'élément "e" devant la liste (de listes) "L".
;;; Par exemple, (cons-L 1 '((2 3) (4))) renvoie ((1) (2 3) (4))
```

```
(define (cons-L e L)
  (cons (list e) L))
```

Question 4 (3 points) :

Écrire une définition Scheme de la fonction `add-prefixe` qui a comme spécification :

```
;;; add-prefixe : alpha * LISTE[LISTE[alpha]] -> LISTE[LISTE[alpha]]
;;; (add-prefixe p L) renvoie la liste (de listes) obtenue en ajoutant l'élément "p"
;;; devant chaque élément de la liste (de listes) "L".
;;; Par exemple (add-prefixe 3 '((4 5) (6 7 8) ())) renvoie ((3 4 5) (3 6 7 8) (3))
```

```
(define (add-prefixe p L)
  ;; add-p : LISTE[alpha] -> LISTE[alpha]
  ;; (add-p L1) renvoie la liste obtenue en ajoutant l'élément "p" devant la liste "L1"
  (define (add-p L1)
    (cons p L1))
  (map add-p L))
```

La grammaire suivante décrit un nœud (d'un système de fichier par exemple) :

<noeud> → (*<nom>* *<contenu>*) RÉPERTOIRE
(*<nom>* *<taille>*) FICHIER

<contenu> → *<noeud>**

<nom> → *un symbole Scheme*

<taille> → *un entier naturel*

Un nœud possède un *nom* et il peut être
– un *répertoire* qui a un *contenu* constitué par un nombre quelconque (éventuellement 0) de nœuds
– ou un *fichier* qui a une certaine *taille*

Ainsi, le nœud `(a (b 5) (c (d 7) (e 9)))`, que l'on peut écrire

```
(a (b 5)
   (c (d 7)
      (e 9)))
```

pour être plus lisible, est un répertoire qui a pour nom *a* et qui contient un fichier de nom *b* et de taille *5* et un répertoire de nom *c*, ce dernier contenant deux fichiers de noms respectifs *d* et *e* et de tailles respectives *7* et *9*.

La barrière syntaxique est constituée des quatre fonctions :

`;;;` *fichier? : Noeud -> bool*
`;;;` *nom-noeud : Noeud -> Symbole*
`;;;` *taille-fichier : Fichier -> nat*
`;;;` *contenu-repertoire : Repertoire -> LISTE[Noeud]*

Question 5 (3 points) :

Donner une implantation de ces quatre fonctions.

```
;;; fichier? : Noeud -> bool
(define (fichier? noeud)
  (and (pair? (cdr noeud)) (number? (cadr noeud))))

;;; nom-noeud : Noeud -> Symbole
(define (nom-noeud noeud)
  (car noeud))

;;; taille-fichier : Fichier -> nat
(define (taille-fichier fichier)
  (cadr fichier))

;;; contenu-repertoire : Repertoire -> LISTE[Noeud]
(define (contenu-repertoire rep)
  (cdr rep))
```

Question 6 (3 points) :

Écrire une définition Scheme de la fonction `taille-noeud` qui a comme spécification :

```
;;; taille-noeud : Noeud -> nat
;;; (taille-noeud noeud) renvoie la somme des tailles des fichiers présents dans le noeud donné.
;;; Par exemple, (taille-noeud '(a (b 5) (c (d 7) (e 9)))) renvoie 21.
```

```
(define (taille-noeud noeud)
  (if (fichier? noeud)
      (taille-fichier noeud)
      (taille-contenu (contenu-repertoire noeud))))

;;; taille-contenu : LISTE[Noeud] -> nat
;;; (taille-contenu L) renvoie la somme des tailles des fichiers présents dans la liste de noeuds donnée.
(define (taille-contenu L)
  (if (pair? L)
      (+ (taille-noeud (car L))
         (taille-contenu (cdr L)))
      0))
```

Pour être nommés correctement, les différents nœuds contenus dans un répertoire doivent avoir des noms différents. Par exemple, les noms des différents nœuds contenus dans le répertoire `(a (b ...) (c ...))` sont corrects alors que les noms des différents nœuds contenus dans le répertoire `(a (b ...) (c ...) (b ...))` ne sont pas corrects (car `b` est présent deux fois).

Les noms présents dans un nœud sont valides lorsqu'il en est ainsi pour tous les répertoires présents dans ce nœud.

Question 7 (3 points) :

Écrire une définition Scheme de la fonction `noms-valide?` qui a comme spécification :

```
;;; noms-valide? : Noeud -> bool
;;; (noms-valide? noeud) renvoie #t ssi il n'existe pas de répertoire présent dans
;;; le noeud donné qui contient deux noeuds ayant le même nom.
```

Exemples :

```
(noms-valide? '(a (b 5)
                  (c (d 7)
                     (b 9))
                  (b 6)))  →  #f

(noms-valide? '(a (b (d 7)
                     (b 9))))  →  #t

(noms-valide? '(a (b (d 7))
                  (c (d 8))))  →  #t
```

```
(define (noms-valide? noeud)
  (if (fichier? noeud)
      #t
      (and (not (existe-doublon? (map nom-noeud
                                      (contenu-repertoire noeud))))
           (noms-valide-contenu? (contenu-repertoire noeud)))))

;;; noms-valide-contenu? : LISTE[Noeud] -> bool
;;; (noms-valide-contenu? L) renvoie #t ssi tous les noeuds "n" de la liste "L"
;;; sont tels que (noms-valide? n) renvoie #t.
(define (noms-valide-contenu? L)
  (if (pair? L)
      (and (noms-valide? (car L))
           (noms-valide-contenu? (cdr L)))
      #t))
```

Définition : une liste $(s_0 \ \ldots \ s_i \ s_{i+1} \ \ldots \ s_p)$ non vide de symboles est un *chemin* d'un nœud n – et nous dirons que ce chemin aboutit à un nœud de nom s_p – lorsqu'il existe une suite $(n_0, \ldots n_i, n_{i+1}, \ldots n_p)$ de nœuds telle que :

- n_0 est le nœud donné n,
- pour tout i strictement inférieur à p, n_i est un répertoire qui contient le nœud n_{i+1},
- pour tout i compris entre 0 et p (inclus), s_i est le nom du nœud n_i.

Remarque : les nœuds $n_0, \ldots n_i, n_{i+1}, \ldots n_{p-1}$ sont obligatoirement des répertoires, n_p pouvant être un répertoire ou un fichier.

Question 8 (4 points) :

Écrire une définition Scheme de la fonction `est-chemin?` qui a comme spécification :

```
;;; est-chemin? : LISTE[Symbole]/non vide/ * Noeud -> bool
;;; ERREUR lorsque le chemin donné est vide
;;; HYPOTHESE : "(noms-valide? noeud)" est vrai
;;; (est-chemin? chemin noeud) renvoie #t ssi "chemin" est un chemin du noeud "noeud"
```

Exemples :

```
(let ((le-noeud '(a (b 5)
                    (c (d 7)
                       (b 9))
                    (d 6))))
  (and (est-chemin? '(a c) le-noeud)
       (est-chemin? '(a c b) le-noeud)
       (not (est-chemin? '(a b c) le-noeud))
       (not (est-chemin? '(a e) le-noeud))))  →  #t
```

Utilisez-vous l'hypothèse ? Si oui, en quoi est-ce intéressant ?

```
(define (est-chemin? chemin noeud)
  (if (equal? (car chemin) (nom-noeud noeud))
      (est-chemin-aux? (cdr chemin) noeud)
      #f))

;;; est-chemin-aux? : LISTE[Symbole] * Noeud -> bool
;;; HYPOTHESE : (noms-valide? noeud) est vrai
;;; (est-chemin-aux? chemin noeud)  renvoie #t ssi "(cons (nom-noeud noeud) chemin)"
;;; est un chemin du noeud "noeud"
(define (est-chemin-aux? chemin noeud)
  (if (pair? chemin)
      (if (fichier? noeud)
          #f
          (est-chemin-L? chemin (contenu-repertoire noeud)))
      #t))

;;; est-chemin-L? : LISTE[Symbole]/non vide/ * LISTE[Noeud] -> bool
;;; HYPOTHESE : "(noms-valide-contenu? L)" est vrai
;;; (est-chemin-L? chemin L) renvoie #t ssi il existe un noeud "n" de "L" tel que
;;; "(est-chemin? chemin n)" est vrai.
(define (est-chemin-L? chemin L)
  (if (pair? L)
      (if (equal? (car chemin) (nom-noeud (car L)))
          (est-chemin-aux? (cdr chemin) (car L))
          (est-chemin-L? chemin (cdr L)))
      #f))
```

L'hypothèse permet, dans la fonction `est-chemin-L?`, de ne pas rappeler cette fonction sur `(cdr L)` lorsque `(equal? (car chemin) (nom-noeud (car L)))` est vrai d'où un gain de temps.

Question facultative (et difficile) :

Nous voudrions enfin savoir si un symbole donné est le nom d'un nœud présent dans un nœud donné. Plus précisément, étant donné un nœud et un symbole, nous voudrions donner la liste des chemins du nœud donné qui aboutissent à des nœuds ayant pour nom le symbole donné. Par exemple, en nommant `cherche` cette fonction :

```
(cherche 'b '(a (b (c 7)
                   (b 8))
                (c (d 7)
                   (b 9))))  →  ((a b) (a b b) (a c b))
```

Question 9 (4 points) :

Écrire une spécification et une définition Scheme de la fonction `cherche`.

```
;;; cherche : Symbole * Noeud -> LISTE[LISTE[Symbole]]
;;; (cherche nom noeud) renvoie la liste des chemins de "noeud" aboutissant à un noeud
;;; de nom "nom" (y compris le chemin (nom) lorsque le noeud donné a pour nom "nom").
(define (cherche nom noeud)
  (if (equal? nom (nom-noeud noeud))
      (cons-L nom (cherche-int nom noeud))
      (cherche-int nom noeud)))

;;; cherche-int : Symbole * Noeud -> LISTE[LISTE[Symbole]]
;;; (cherche-int nom noeud) renvoie la liste des chemins de "noeud" aboutissant
;;; à un noeud de nom "nom", excepté le chemin éventuel (nom)
(define (cherche-int nom noeud)
  (if (fichier? noeud)
      '()
      (add-prefixe (nom-noeud noeud)
                   (cherche-L nom (contenu-repertoire noeud)))))

;;; cherche-L : Symbole * LISTE[Noeud] -> LISTE[LISTE[Symbole]]
;;; (cherche-L nom L) renvoie la liste des chemins des différents noeuds de la liste "L"
;;; (y compris le chemin (nom) lorsque l'un des noeuds de "L" a pour nom "nom").
(define (cherche-L nom L)
  (if (pair? L)
      (append (cherche nom (car L))
              (cherche-L nom (cdr L)))
      '()))
```