Université de La Réunion — lundi 5 juin 2000

DEUG 1^{ere} année – MIAS 1 et MASS 1

UE4 – Informatique 2 – Examen terminal

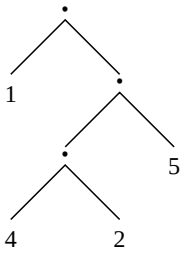
Durée : 2 heures – Aucun document n'est autorisé. 1

Répondre uniquement dans les cadres prévus à cet effet

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

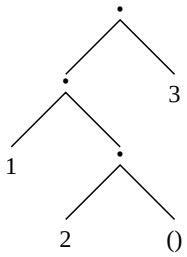
Partie 1 : Paires pointées et listes

Définition 1 : Une paire pointée multi–niveaux est dite *complète sur* $\mathbb{N}$ si toutes les feuilles de sa représentation arborescente (où car est representé par le sous–arbre gauche et cdr est représenté par le sous–arbre droit) appartiennent à $\mathbb{N}$.
Le schéma suivant montre les représentations arborescentes d'un exemple de paire pointée complète sur $\mathbb{N}$ et de deux contre-exemples.

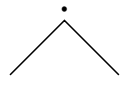


(1 . ((4 . 2) . 5))

un exemple de paire pointée complète sur N



((1 2) . 3)



(1 . a)

deux objets qui ne sont pas des paires pointées complètes sur N

Définition 2 : On appelle *feuillage d'entiers* un objet qui est, soit un entier, soit une paire pointée complète sur $\mathbb{N}$.

(4 pts) **Exercice 1 :** Écrivez en Scheme un prédicat feuillage? qui teste si un objet est un feuillage d'entiers.

Un objet o *est un feuillage si c'est un entier ou bien si c'est une paire pointée dont le* car *et le* cdr *sont des feuillages.*

```
(define feuillage?
   (lambda (o)
      (or (integer? o)
          (and (pair? o)
               (feuillage? (car o))
               (feuillage? (cdr o))))))
```

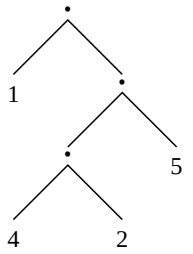
Tournez SVP $\Longrightarrow$

(3 pts) **Exercice 2 :** Écrivez en Scheme une fonction liste-feuilles qui prend en argument un feuillage d'entiers et renvoie la liste des feuilles de sa représentation arborescente (de gauche à droite).
Par exemple `(liste-feuilles '(1 . ((1 . 2) . 2)))` doit renvoyer `(1 1 2 2)`.

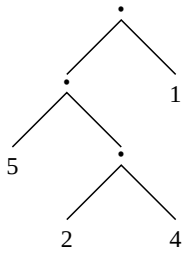
Si le feuillage `f` *donné en argument est un entier alors on renvoie la liste contenant cet entier sinon on renvoie la liste obtenue par concaténation de la liste des feuilles du* `(car f)` *et de la liste des feuilles du* `(cdr f)`.

```
(define liste-feuilles
   (lambda (f)
      (if (integer? f)
          (list f)
          (append (liste-feuilles (car f)) (liste-feuilles (cdr f))))))
```

(3 pts) **Exercice 3 :** Écrivez en Scheme une fonction sym qui prend en argument un feuillage d'entiers et renvoie le feuillage d'entiers symétrique. Par exemple `(sym '(1 . ((4 . 2) . 5)))` doit renvoyer `((5 . (2 . 4)) . 1)`.



un feuillage d'entiers
(1 . ((4 . 2) . 5))



et le feuillage symétrique
((5 . (2 . 4)) . 1)

Si le feuillage `f` *donné en argument est un entier on renvoie cet entier sinon on construit la paire pointée dont le* `car` *est le feuillage symétrique du* `(cdr f)` *et dont le* `cdr` *est le feuillage symétrique du* `(car f)`.

```
(define sym
   (lambda (f)
      (if (integer? f)
          f
          (cons (sym (cdr f)) (sym (car f))))))
```

Université de La Réunion — lundi 5 juin 2000

DEUG 1^{ere} année – MIAS 1 et MASS 1

UE4 – Informatique 2 – Examen terminal

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

Répondre uniquement dans les cadres prévus à cet effet

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

Partie 2 : Type abstrait de données

On définit, au moyen des fonctions suivantes, le type abstrait de données $\mathcal{S}$ qui permet de manipuler les suites strictement croissantes finies d'entiers naturels :

- les fonctions génératrices : $\begin{cases} \textbf{suite-vide} & : & \rightarrow & \mathcal{S} \\ \textbf{ajouter-terme} & : & \mathcal{S} \times \mathbb{N} & \rightarrow & \mathcal{S} \end{cases}$
- les fonctions caractéristiques : $\begin{cases} \textbf{est-vide?} & : & \mathcal{S} & \rightarrow & \{\#f, \#t\} \\ \textbf{min} & : & \mathcal{S} & \rightarrow & \mathbb{N} \\ \textbf{enlever-min} & : & \mathcal{S} & \rightarrow & \mathcal{S} \end{cases}$

Exercice 1 : En supposant que l'on code une suite croissante à l'aide d'une liste dont les éléments sont rangés par ordre croissant, écrivez les fonctions génératrices et caractéristiques données ci-dessus.

(0,5 pt) 1a. La fonction **suite-vide** qui ne prend pas d'argument et renvoie une suite vide :

C'est une fonction sans paramètre qui renvoie la liste vide.

```
(define sym
   (lambda () '()))
```

(0,5 pt) 1b. La fonction **est-vide?** qui teste si une suite est vide ou pas :

C'est tout simplement la fonction `null?`.

```
(define est-vide? null?)
```

(0,5 pt) 1c. La fonction **min** qui renvoie le plus petit terme d'une suite :

Le plus petit élément étant au début de la suite, il s'agit de la fonction `car`.

```
(define min car)
```

(0,5 pt) 1d. La fonction **enlever-min** qui retire d'une suite son plus petit terme.

Il faut retirer le premier élément de la suite, c'est la fonction `cdr`.

```
(define enleve-min cdr)
```

Tournez SVP $\Longrightarrow$

(2 pts) 1e. La fonction **ajouter-terme**, qui fonctionne comme suit: pour tout $S \in \mathcal{S}$ et tout $x \in \mathbb{N}$, (**ajouter-terme** S x) ajoute à la suite S le nouveau terme x. Attention, le résultat doit être une suite strictement croissante.

Si la suite est vide ou si lélément `x` *à ajouter est plus petit que le premier élément de la suite, il suffit de placer x au début de la liste, sinon il faut l'ajouter au bon endroit dans le* `(cdr s)`.

```
(define ajouter-terme
   (lambda (s x)
      (if (or (null? s) (<= x (car s)))
          (cons x s)
          (cons (car s) (ajouter-terme (cdr s) x)))))
```

Exercice 2 : Toutes les fonctions écrites dans cet exercice doivent être indépendantes de l'implantation du type abstrait de données. Vous les définirez en utilisant les fonctions génératrices et caractéristiques mentionnées plus haut.

(3 pts) 2a. Ecrivez la fonction **union** qui prend en argument deux suites croissantes $S1$ et $S2$ et qui renvoie une suite croissante dont les termes sont l'union de ceux de $S1$ et $S2$.

Exemple : l'union des suites 1, 2, 4, 6, 9 et 1, 3, 5, 9 est la suite 1, 2, 3, 4, 5, 6, 9.

Attention, on n'a pas le droit d'utiliser la fonction `member` *car on ne manipule pas des listes. Pour savoir si un entier est dans une suite, il faut comparer celui-ci au plus petit élément de la suite : fonction* `min`.

```
(define union
   (lambda (s1 s2)
      (cond ((est-vide? s1) s2)
            ((est-vide? s2) s1)
            ((= (min s1) (min s2)) (union (enlever-min s1) s2))
            ((< (min s1) (min s2)) (ajouter-terme (union (enlever-min s1) s2)
                                                  (min s1)))
            (else (ajouter-terme (union s1 (enlever-min s2)) (min s2))))))
```

(3 pts) 2b. Ecrivez la fonction **intersection** qui prend en argument deux suites croissantes $S1$ et $S2$ et qui renvoie une suite croissante dont les termes sont l'intersection de ceux de $S1$ et $S2$.

Exemple : l'intersection des suites 1, 2, 4, 6, 9 et 1, 3, 6, 9 donne la suite 1, 6, 9.

```
(define intersection
   (lambda (s1 s2)
      (cond ((or (est-vide? s1) (est-vide? s2)) (suite-vide))
            ((= (min s1)(min s2)) (ajouter-terme (intersection (enlever-min s1)
                                                               (enlever-min s2))
                                                 (min s1)))
            ((< (min s1)(min s2)) (intersection (enlever-min s1) s2))
            (else (intersection s1 (enlever-min s2))))))
```

Université de La Réunion lundi 5 juin 2000

DEUG 1^{ere} année – MIAS 1 et MASS 1

UE4 – Informatique 2 – Examen terminal

Durée : 2 heures – Aucun document n'est autorisé. **3**

Répondre uniquement dans les cadres prévus à cet effet

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

Partie 3 : Complexité en temps

Remarque importante : Dans les exercices qui suivent, vous préciserez clairement en fonction de quelle mesure des données vous évaluez les complexités en temps. Vous expliciterez tout aussi clairement les équations de récurrence permettant de calculer ces complexités et après les avoir résolues vous donnerez leur ordre de grandeur en utilisant la notation O.
On vous demande d'évaluer la complexité en temps dans le pire des cas. C'est à dire, en fonction d'une mesure des données, le temps maximum consommé pour évaluer un appel à la fonction.

(4 pts) **Exercice 1 :** Évaluer la complexité en temps dans le pire des cas de la fonction suivante qui prend comme paramètres un objet et une liste plate.

```
; estdans : Objet, Liste --> Booleen
;              x ,   l   |-> #t si x est dans l
;                           #f sinon

(define estdans
   (lambda (x l)
      (cond ((null? l) #f)
            ((equal? x (car l)) #t)
            (else (esdans x (cdr l))))))
```

On évalue la complexité en temps en fonction de la longueur de la liste `l`*. Notons n la longueur de la liste, C_e le temps maximum (qui est indépendant de* `l`*) nécessaire pour évaluer le corps de la fonction sans l'appel récursif et $T_e(n)$ le temps consommé pour évaluer un appel à* `est-dans` *avec un liste de longueur n.*

Si la liste est vide, aucun appel récursif n'est nécessaire et le temps comsommé est C_e.
Si la liste n'est pas vide ($n > 0$), le temps consommé est celui consommé par l'évaluation du corps de la fonction auquel il faut ajouter le temps consommé pour évaluer l'appel `(est-dans x (cdr l))`.
On obtient les équations :

$$\begin{cases} T_e(0) = C_e \\ T_e(n) = C_e + T_e(n-1) \end{cases}$$

T_e est une suite arithmétique de premier terme C_e et de raison C_e, on a donc:

$$T_e(n) = (n+1)C_e = O(n)$$

Tournez SVP $\Longrightarrow$

(6 pts) **Exercice 2 :** Évaluer la complexité en temps dans le pire des cas de la fonction suivante qui prend comme paramètres deux listes plates.

```
; truc : Liste, Liste --> Liste
;          l1 ,   l2  |-> remplace tous les elements de l1 qui sont dans l2 par des 1
;                         et tous les element de l1 qui ne sont pas dans l2 par des 0
(define truc
   (lambda (l1 l2)
       (cond ((null? l1) '())
             ((estdans (car l1) l2) (cons 1 (truc (cdr l1) l2)))
             (else (cons 0 (truc (cdr l1) l2))))))
```

Ici la complexité en temps dépend de la longueur de `l1` *qui est le paramètre d'induction de la fonction* `truc` *mais aussi de la longueur de la liste* `l2` *car il faut prendre en compte le temps consommé par les appels à la fonction* `estdans`*, temps qui dépend de la longueur de* `l2`*.*
Notons n_1 la longueur de la liste `l1`*, n_2 la longueur de* `l2`*, C_t le temps maximum (indépendant de* `l1` *et* `l2`*) nécessaire pour évaluer le corps de la fonction sans l'appel à* `estdans` *ni l'appel récursif à* `truc` *et notons $T_t(n_1, n_2)$ le temps consommé pour évaluer un appel à* `truc` *avec des listes* `l1` *et* `l2` *de longueur n_1 et n_2.*

Si la liste `l1` *est vide, quelque soit la longueur de* `l2`*, le temps consommé est C_t.*
Si `l1` *n'est pas vide, le temps consommé est C_t auquel il faut ajouter le temps consommé par l'appel* `(estdans (car l1) l2)` *et celui consommé par l'appel récursif* `(truc (cdr l1) l2)`*.*

On obtient les équations :

$$\begin{cases} T_t(0, n_2) = C_t \\ T_t(n_1, n_2) = C_t + T_e(n_2) + T_t(n_1 - 1, n_2) = C_t + (n_2 + 1)C_e + T_t(n_1 - 1, n_2) \end{cases}$$

On résoud cette récurrence par sommation et on obtient

$$T_t(n_1, n_2) = (n_1 + 1)C_t + n_1(n_2 + 1)C_e = O(n_1 \times n_2)$$

Université de La Réunion — lundi 5 juin 2000

DEUG 1^{ere} année – MIAS 1 et MASS 1

UE4 – Informatique 2 – Examen terminal

Durée : 2 heures – Aucun document n'est autorisé. **4**

Répondre uniquement dans les cadres prévus à cet effet

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

Partie 4 : Ordre supérieur

(4 pts) **Exercice 1 :** Donnez le résultat de l'évaluation des expressions suivantes :

```
>(apply append '((1) ((a)) (a(b(c)))))
(1 (a) a (b (c)))
>(map cdr '((1 2) (2 (3 a))))
((2) ((3 a)))
>(map car '((2 3 4) () (a (b))))
erreur à cause de l'appel à car sur ()
>(map * '(1 2) '(5 6) '(3 2))
(15 24)
>(map (lambda (x) (or (odd? x) (>0 x))) '(-7 9 4 0 ))
(#t #t #f #f)
>((lambda (a b c) ( / (* a a ) (- b c) ) ) 4 6 4 )
8
>(apply * (map (lambda (x) 2) '(2 (1 1) 0 (4 (6 5))) ))
16
>(apply + (map (lambda (l) (+ (cadr l) 1) ) '((1 2) (2 3) (3 4))))
12
```

(3 pts) **Exercice 2 :** Ecrivez, en utilisant la fonction prédéfinie `map`, une fonction non récursive `remplacer` qui prend comme paramètre une liste plate d'entiers `l` et qui remplace tous les entiers pairs de `l` par 0 et tous les entiers impairs par 1.

Par exemple `'(remplacer '(1 2 3 4 5 6 7 8))` doit renvoyer la liste `'(1 0 1 0 1 0 1 0)`.

Il faut appliquer à tous les éléments de la liste `l` *la fonction qui prend comme paramètre un entier et renvoie 0 si celui est pair ou 1 s'il est impair.*

```
(define remplacer
   (lambda (l)
      (map (lambda (x) (if (even? x) 0 1))   l)))
```

Tournez SVP $\Longrightarrow$

(3 pts) **Exercice 3 :** En utilisant les fonctions prédéfinies `apply` et `map`, écrire une fonction `moyennes`, non récursive, qui prend comme paramètre une liste `ll` de listes de nombres et renvoie la liste des moyennes des nombres de chaque liste présente dans `ll`.

Par exemple : `(moyennes '((10 20) (12 14 16 14) (8 10 12)) )` doit renvoyer la liste `'(15 14 10)`.

Il faut appliquer à chaque liste de nombres une fonction qui calcule la moyenne de ces nombres. Cette moyenne est obtenue en faisant la somme des nombres de la liste et en divisant par la taille de la liste.

```
(define moyennes
   (lambda (ll)
      (map (lambda (l) (/ (apply + l) (length l)))   ll)))
```