# Examen – Module d'informatique 1 – Septembre 2000
# MIAS 1ère année – 1er semestre

Aucun document ni machine électronique n'est permis.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue (utiliser le dos de la feuille précédente si la réponse déborde des boîtes). Le barème (sur 50) apparaissant dans chaque boîte n'est donné qu'à titre indicatif.

La clarté des réponses et la présentation des programmes seront appréciées. Toutes les fonctions apparaissant dans les réponses doivent être accompagnées de leur spécification. Ne pas désagrapher les feuilles.

## Exercice 1

Cette section contient plusieurs exercices de regroupement de termes de listes suivant diverses modalités. Les deux premiers exercices préparent les suivants.

**Question 1.1** – Écrire une fonction `premiers` qui prend un entier naturel $n$ et une liste $(e_1 \quad e_2 \quad \ldots \quad e_p)$ et qui construit la liste des $n$ premiers termes de la liste reçue. Ainsi

```
(premiers 2 '(a b c d)) → (a b)
(premiers 5 '(a b c d)) → (a b c d)
```

Réponse [3/50]

```
;; Retourne la liste des n premiers termes d'une liste.
;;  EntierNaturel * alpha* -> alpha*
(define (premiers n liste)
  (if (pair? liste)
      (if (> n 0)
          (cons (car liste) (premiers (- n 1) (cdr liste)))
          '() )
      '() ) )
```

**Question 1.2** – Écrire une fonction `apres` qui prend un entier naturel $n$ et une liste $(e_1 \quad e_2 \quad \ldots \quad e_p)$ et qui construit une liste $(e_{n+1} \quad \ldots \quad e_p)$ c'est-à-dire la liste initialement reçue, privée de ses $n$ premiers termes si elle en a au moins autant reçus.

```
(apres 2 '(a b c d)) → (c d)
(apres 5 '(a b c d)) → ()
```

Section | Numéro d'anonymat

Réponse [3/50]

```
;; Retourne la liste de tous les termes d'une liste sauf les n premiers.
;;  EntierNaturel * alpha* -> alpha*
(define (apres n liste)
  (if (pair? liste)
      (if (> n 0)
          (apres (- n 1) (cdr liste))
          liste )
      '() ) )
```

**Question 1.3** – Écrire une fonction `grouper` qui prend un entier naturel non nul $n$, une liste $(e_1 \quad e_2 \quad \ldots \quad e_p)$ et qui construit la liste de ces mêmes termes regroupés par $n$. Prenez garde aux termes de la fin de la liste. Ainsi,

```
(grouper 2 '(a b c d)) → ((a b) (c d))
(grouper 2 '(a b c d e)) → ((a b) (c d) (e))
(grouper 6 '(a b c d)) → ((a b c d))
```

Réponse [4/50]

```
;; (e1 e2 ... eN eN+1 ... e2N e2N+1 ...) -> ((e1 ... eN)(eN+1 ... e2N) ...)
(define (grouper n liste)
  (if (pair? liste)
      (cons (premiers n liste) (grouper n (apres n liste)))
      '() ) )
```

**Question 1.4** – Écrire une fonction `repuorg` qui prend un entier naturel non nul $n$, une liste $(e_1 \quad e_2 \quad \ldots \quad e_p)$ et qui construit la liste de ces mêmes termes regroupés par $n$ en commençant par la fin. Prenez garde aux termes du début de la liste. Ainsi,

```
(repuorg 2 '(a b c d)) → ((a b) (c d))
(repuorg 2 '(a b c d e)) → ((a) (b c) (d e))
(repuorg 6 '(a b c d)) → ((a b c d))
```

Réponse [6/50]

Voici une solution utilisant la fonction `reverse` (on pourrait aussi l'écrire en une seule ligne !) :

```
;; (... e2N+1 e2N ... eN+1 eN ... e2 e1) -> (... (e2N ... eN+1)(eN ... e1))
(define (repuorg n liste)
  (let ((r (grouper n (reverse liste))))
    (reverse (map reverse r)) ) )
```

En voici une autre en style direct qui utilise `grouper` :

```
(define (repuorg2 n liste)
  (let ((debut (modulo (length liste) n)))
    (if (= debut 0)
        (grouper n liste)
        (cons (premiers debut liste)
              (grouper n (apres debut liste)) ) ) ) )
```

## Exercice 2

Soit la grammaire suivante définissant un mobile plein de cadeaux.

| | | | |
|---|---|---|---|
| <*mobile*> | → | `(CADEAU` <*cadeau*> `)` | Règle 1 |
| | | `(` <*mobile*> `BRANCHE` <*mobile*> `)` | Règle 2 |

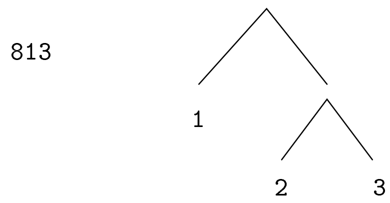
<*cadeau*> → <*entier naturel*>

Voici deux exemples de mobiles mettant en jeu, à eux deux, toutes les règles précédentes.

```
;; Exemples de mobiles :
(define mobile1
  '(CADEAU 813) )

(define mobile2
  '((CADEAU 1) BRANCHE ((CADEAU 2) BRANCHE (CADEAU 3))) )
```

Ces mobiles peuvent être figurés graphiquement comme suit :




Le mobile de gauche est réduit à un cadeau tandis que le mobile de droite est dit « branchu » parce qu'il possède deux « branches » contenant chacune un mobile.

**Question 2.1** – Écrire les fonctions `mobile-gauche` et `mobile-droite` accédant aux branches gauche et droite d'un mobile branchu. Ainsi

`(mobile-gauche mobile2)` → `(CADEAU 1)`

Réponse [2/50]

```
;; Accesseurs du mobile à deux branches
;;  mobile[branchu] -> mobile
(define (mobile-gauche s)
  (car s) )

(define (mobile-droite s)
  (caddr s) )
```

**Question 2.2** – Écrire le prédicat `est-cadeau?` qui reconnait si un mobile est formé suivant la règle 1.

`(est-cadeau? mobile1)` → `#T`
`(est-cadeau? mobile2)` → `#F`

Réponse [1/50]

```
;; Reconnaisseur d'un mobile réduit à un cadeau
;;  mobile -> bool
(define (est-cadeau? s)
  (equal? (car s) 'CADEAU) )
```

**Question 2.3** – Écrire la fonction `valeur-cadeau` qui retourne la valeur du cadeau d'un mobile satisfaisant le prédicat `est-cadeau?`.

`(valeur-cadeau mobile1)` → `813`

Section Numéro d'anonymat

Réponse [2/50]

```
;; Extracteur du cadeau
;;  mobile[non branchu] -> entierNaturel
(define (valeur-cadeau s)
  (cadr s) )
```

**Question 2.4** – Écrire une fonction `valeur-mobile` qui prend un mobile bien formé et qui calcule la somme des valeurs des cadeaux qu'il contient.

Réponse [3/50]

```
;; Retourne la somme des valeurs des cadeaux.
(define (valeur-mobile s)
  (if (est-cadeau? s)
      (valeur-cadeau s)
      (+ (valeur-mobile (mobile-gauche s))
         (valeur-mobile (mobile-droite s)) ) ) )
```

**Question 2.5** – Un mobile est à l'équilibre lorsque ses deux branches ont même valeur de cadeaux. Écrire le prédicat `a-l-equilibre?` testant si un mobile est à l'équilibre.

```
(a-l-equilibre? mobile2) → #F
(a-l-equilibre? '((CADEAU 4) BRANCHE ((CADEAU 1) BRANCHE (CADEAU 3)))) → #T
```

Réponse [4/50]

```
;; Vérifie qu'un mobile est à l'équilibre cad qu'il a autant de cadeaux à
;; gauche qu'à droite.
(define (a-l-equilibre? s)
  (or (est-cadeau? s)
      (= (valeur-mobile (mobile-gauche s))
         (valeur-mobile (mobile-droite s)) ) ) )
```

**Question 2.6** – Donner une spécification du prédicat suivant, donner également un exemple de mobile qui satisfait le prédicat et un exemple de mobile qui ne le satisfait pas.

```
(define (xyzzy? s)
  (or (est-cadeau? s)
      (and (xyzzy? (mobile-gauche s))
           (xyzzy? (mobile-droite s))
           (= (valeur-mobile (mobile-gauche s))
              (valeur-mobile (mobile-droite s)) ) ) ) )
```

Réponse [4/50]

Le prédicat `xyzzy?` vérifie qu'un mobile est bien équilibré à tout niveaux, c'est-à-dire qu'il est à l'équilibre et que ses branches gauche et droite sont bien équilibrées itou. Parmi les deux mobiles qui suivent :

```
((CADEAU 4) BRANCHE ((CADEAU 1) BRANCHE (CADEAU 3)))
((CADEAU 4) BRANCHE ((CADEAU 2) BRANCHE (CADEAU 2)))
```

Le premier est à l'équilibre mais pas bien équilibré ce qu'est, par contre, le second.

**Question 2.7** – Écrire une fonction nommée `collecter-cadeaux`, retournant la liste de toutes les valeurs des cadeaux d'un mobile.

Section | Numéro d'anonymat

Réponse [4/50]

```
;; Retourne la liste de tous les cadeaux d'un mobile.
(define (collecter-cadeaux s)
  (if (est-cadeau? s)
      (list (valeur-cadeau s))
      (append (collecter-cadeaux (mobile-gauche s))
              (collecter-cadeaux (mobile-droite s)) ) ) )
```

# Exercice 3

Cette section contient des exercices itérant des fonctions. On pourra écrire des fonctions communes aux solutions de ces questions.

**Question 3.1** – Écrire une fonction nommée `appliquer-n` prenant une fonction unaire $f$, un entier naturel $n$ et une valeur $x$ et calculant $\underbrace{f \circ f \circ \ldots \circ f}_{n}(x)$. On pourra par exemple écrire :

```
;; retourne l'entier naturel qui suit n.
;;  EntierNaturel -> EntierNaturel
(define (successeur n)
  (+ n 1) )
```

`(appliquer-n successeur 3 10)` → `13`

Réponse [3/50]

```
;; Appliquer n fois la fonction f au terme x.
;;  (alpha -> alpha) * EntierNaturel * alpha -> alpha
(define (appliquer-n f n x)
  (if (> n 0)
      (appliquer-n f (- n 1) (f x))
      x ) )
```

**Question 3.2** – Écrire une fonction nommée `iterer-sur-terme` qui prend une fonction $f$, une valeur $e$ et un entier naturel $n$ non nul et qui calcule la liste ($e \quad f(e) \quad f \circ f(e) \quad \ldots \quad \underbrace{f \circ f \ldots \circ f}_{n-1}(e)$). Ainsi,

```
;; Retourne le carré d'un entier naturel.
;;  EntierNaturel -> EntierNaturel
(define (carre n)
  (* n n) )
```

`(iterer-sur-terme carre 2 6)` → `(2 4 16 256 65536 4294967296)`

Section Numéro d'anonymat

Réponse [4/50]

Voici une solution en style direct :

```
;; Construit la liste de n termes : (e (f e) (f (f e)) ...)
;; f : alpha -> alpha
;; e : alpha
;; n : EntierNaturel[n>0]
(define (iterer-sur-terme f e n)
  (if (> n 1)
      (cons e
            (map f (iterer-sur-terme f e (- n 1))) )
      (list e) ) )
```

et encore deux autres moins cons-ommatrice :

```
(define (iterer-sur-terme3 f e n)
  (if (> n 1)
      (cons e
            (iterer-sur-terme3 f (f e) (- n 1)) )
      (list e) ) )


(define (iterer-sur-terme2 f e n)
  (define (iterer-vraiment fn n)
    (if (> n 0)
        (cons (fn e)
              (iterer-vraiment (composer f fn) (- n 1)) )
        '() ) )
  (define (identite x)
    x )
  (iterer-vraiment identite n) )
```

**Question 3.3** – Écrire une fonction nommée `iterer` prenant une fonction unaire $f$ et un nombre entier naturel $n$ et retournant la fonction $x \to \underbrace{f \circ f \circ \ldots \circ f}_{n}(x)$. On pourra, par exemple, écrire :

```
((iterer successeur 3) 10) → 13
```

Réponse [3/50]

```
;; Fabrique la nième itérée de la fonction f.
;; f : a -> a
;; n : entier naturel
(define (iterer f n)
  (define (appliquer x)
    (appliquer-n f n x) )
  appliquer )
```

**Question 3.4** – Écrire une fonction nommée `iterer-sur-liste` qui prend une fonction unaire $f$ et une liste de termes $(e_1 \quad e_2 \ldots e_n)$ et qui renvoie la liste $(f(e_1) \quad f \circ f(e_2) \quad \ldots \quad \underbrace{f \circ f \ldots \circ f}_{n}(e_n))$. Ainsi,

```
;; Retourne le carré d'un entier naturel.
;;  EntierNaturel -> EntierNaturel
(define (carre n)
  (* n n) )
```

```
(iterer-sur-liste carre '(1 2 3 4)) → (1 16 6561 4294967296)
```

Section Numéro d'anonymat

Réponse [4/50]

Voici une solution directe :

```
;; (e1 e2 e3 ...) -> (e1 (f e1) (f (f e2)) ...)
;;  (alpha -> alpha) * alpha* -> alpha*
(define (iterer-sur-liste f liste)
  (if (pair? liste)
      (cons (f (car liste))
            (iterer-sur-liste f (map f (cdr liste))) )
      '() ) )
```

et une autre plus compliquée dont la consommation (en appels à `cons`) est linéaire c'est-à-dire égale à la longueur de la liste initiale :

```
;; (alpha -> beta) * (gamma -> alpha) -> (gamma -> beta)
(define (composer f g)
  (define (resultat x)
    (f (g x)) )
  resultat )

(define (iterer-sur-liste2 f l)
  (define (iterer-vraiment fn l)
    (if (pair? l)
        (cons (fn (car l))
              (iterer-vraiment (composer f fn) (cdr l)) )
        '() ) )
  (iterer-vraiment f l) )
```