Examen – Module d'informatique 1 – Janvier 2004
MIAS 1ère année – 1er semestre

Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme. Les téléphones doivent être éteints et rangés dans les sacs.

L'examen dure deux heures. Ce sujet comporte 8 pages.

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.

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

La clarté des réponses et la présentation des programmes seront appréciées. Sauf mention contraire, les fonctions qui apparaîtront dans vos réponses devront être accompagnées de leur spécification.

Ne pas désagrafer les feuilles.

Exercice 1

Le but de ce problème est de compléter logiquement des suites de nombres comme dans certains tests. Voici quelques exemples en partant des plus simples (les nombres attendus apparaissent à droite du point d'interrogation).

`1 2 3 4 5 6 ? 7 8 9` (traité question 1.7) `1 3 5 7 9 11 ? 13 15 17` (traité question 1.7) `1 2 4 7 11 16 ? 22 29 37 46` (traité question 1.9)

Fonctions de service

Question 1.1 – Écrire la signature et une définition d'un prédicat, nommé `au-moins-trois?`, vérifiant qu'une liste d'entiers contient au moins trois entiers. Ainsi

```
(au-moins-trois? '(1 0 2))  →  #T
(au-moins-trois? '(34))  →  #F
```

Réponse [1/50]

```
;;; au-moins-trois?: LISTE[int] -> bool
;;; (au-moins-trois? L) vérifie que L contient au moins trois entiers.
(define (au-moins-trois? L)
  (and (pair? L)
       (pair? (cdr L))
       (pair? (cddr L)) ) )
```

Question 1.2 – Écrire la signature et une définition d'un prédicat, nommé `tous-egaux-a?`, vérifiant que tous les éléments d'une liste d'entiers donnée sont égaux à un entier donné. Ainsi

```
(tous-egaux-a? 4 '(0 1 0 2))  →  #F
(tous-egaux-a? 4 '(4 4 4 4 4))  →  #T
(tous-egaux-a? 4 '(4 4 5 4))  →  #F
(tous-egaux-a? 4 '())  →  #T
```

Réponse [2/50]

```
;;; tous-egaux-a?: int * LISTE[int] -> bool
;;; (tous-egaux-a? x L) vérifie que tous les éléments de L sont égaux à x.
(define (tous-egaux-a? x L)
  (if (pair? L)
      (and (= x (car L))
           (tous-egaux-a? x (cdr L)) )
      #t ) )
```

Question 1.3 – Écrire la signature et une définition d'un prédicat, nommé `tous-egaux?`, vérifiant que tous les éléments d'une liste d'entiers non vide sont égaux. Ainsi

```
(tous-egaux? '(0 1 0 2))  → #F
(tous-egaux? '(4 4 4 4 4))  → #T
```

Réponse [2/50]

```
;;; tous-egaux?: LISTE[int]/non vide/ -> bool
;;; (tous-egaux? L) vérifie que L ne contient que des nombres égaux.
(define (tous-egaux? L)
  (tous-egaux-a? (car L) (cdr L)) )
```

Suites en progression arithmétique

Une suite d'entiers u_n est en progression arithmétique s'il existe une constante r (nommée la « raison ») telle que $u_{n+1} = u_n + r$.

Question 1.4 – Écrire la signature et une définition d'une fonction, nommée `progression-arithmetique`, prenant l'élément initial u_0, la raison r et un entier naturel n, et renvoyant la liste $(u_0\ u_1\ \ldots\ u_{n-1})$. Ainsi

```
(progression-arithmetique 0 1 3)  → (0 1 2)
(progression-arithmetique 10 2 4)  → (10 12 14 16)
```

Réponse [2/50]

```
;;; progression-arithmetique : int * int * nat -> LISTE[int]
;;; (progression-arithmetique u0 r n) renvoie les n premiers termes de la
;;; progression arithmétique de raison r et d'élément initial u0.
(define (progression-arithmetique u0 r n)
  (if (> n 0)
      (cons u0 (progression-arithmetique (+ u0 r) r (- n 1)))
      '() ) )
```

Étant donnée une liste d'entiers $(u_0\ u_1\ \ldots\ u_{n-1})$, on définit sa liste des différences comme étant la liste $(u_1 - u_0\ \ u_2 - u_1\ \ldots\ u_{n-1} - u_{n-2})$.

Question 1.5 –

1. Énoncez la relation liant les longueurs d'une liste et de sa liste des différences.
2. Quelle propriété vérifie la liste des différences d'une progression arithmétique ?
3. Écrire la signature et une définition de la fonction, nommée `liste-differences`, prenant une liste d'entiers naturels non vide et renvoyant la liste de leurs différences. Par exemple :

```
(liste-differences '(3 2 4 5))  → (-1 2 1)
(liste-differences '(1 2 3 4 5 6))  → (1 1 1 1 1)
(liste-differences '(1))  → ()
```

Section Numéro d'anonymat

Réponse [3/50]

1. La liste des différences a pour longueur la longueur de la liste argument moins un. En Scheme, cela donnerait

```
(= (+ 1 (length (liste-differences L))) (length L))
```

2. La liste des différences d'une progression arithmétique est une liste ne comportant que la raison, elle satisfait donc le prédicat `tous-egaux?`.

3.

```
;;; liste-differences : LISTE[int]/non vide/ -> LISTE[int]
;;; (liste-differences '(a b c ...)) renvoie (list (- b a) (- c b) ...)
(define (liste-differences liste)
  (if (pair? (cdr liste))
      (cons (- (cadr liste) (car liste))
            (liste-differences (cdr liste)) )
      '() ) )
```

Question 1.6 – Écrire la signature et une définition d'un prédicat, nommé `progression-arithmetique?`, vérifiant qu'une liste est en progression arithmétique. Ce prédicat devra également rendre `#f` lorsque la liste a moins de trois éléments. Ainsi

```
(progression-arithmetique? '(1 3))  → #F
(progression-arithmetique? '(1 3 5 7))  → #T
(progression-arithmetique? '(1 2 4 8 16))  → #F
```

Réponse [2/50]

```
;;; progression-arithmetique?: LISTE[int] -> bool
;;; Hypothèse : liste a au moins trois éléments
;;; (progression-arithmetique? liste) rend #t ssi liste est une liste
;;; en progression arithmétique
(define (progression-arithmetique? liste)
  (if (au-moins-trois? liste)
      (tous-egaux? (liste-differences liste))
      #f) )
```

Notons qu'il serait hasardeux de vouloir prolonger, en une progression arithmétique, une suite ayant moins de trois éléments.

Question 1.7 – Écrire la signature et une définition d'un semi-prédicat, nommé `prolongement-arithmetique`, prenant un entier naturel *n* et une liste d'entiers naturels. Si cette liste a au moins trois éléments et correspond à un début de suite en progression arithmétique, alors la valeur renvoyée est la liste initiale complétée des termes qui la suivent logiquement afin de former une liste comprenant exactement *n* termes ; sinon la fonction rend `#f`. Par exemple,

```
(prolongement-arithmetique 5 '(1 3))  → #F
(prolongement-arithmetique 6 '(1 3 5 7))  → (1 3 5 7 9 11)
(prolongement-arithmetique 3 '(1 3 5 7))  → (1 3 5)
(prolongement-arithmetique 10 '(1 2 4 8 16))  → #F
```

Réponse [2/50]

```
;;; prolongement-arithmetique: nat * LISTE[int] -> LISTE[int] + #f
;;; Hypothèse : liste a au moins trois éléments
;;; (prolongement-arithmetique n liste) rend une liste de longueur n
;;; dont le préfixe est liste et qui complète « logiquement »
;;; liste à condition que liste soit une progression arithmétique.
;;; C'est un semi-prédicat car la valeur finale est fausse si liste n'est
;;; pas une progression arithmétique.
(define (prolongement-arithmetique n liste)
  (if (progression-arithmetique? liste)
      (progression-arithmetique (car liste) (- (cadr liste) (car liste)) n)
      #f))
```

Section | Numéro d'anonymat

Suites dont la liste des différences est en progression arithmétique

La connaissance du premier terme et de la liste des différences permet de reconstruire toute la liste initiale. Par exemple, si le premier terme est `-1` et si la liste des différences est `(3 -3 5)` :

- le premier terme de la liste initiale est, bien sûr, `-1` ;
- le deuxième terme de la liste initiale est égal à $(-1)+3$, soit `2` ;
- le troisième terme de la liste initiale est égal à $2+(-3)$, soit `-1` ;
- le dernier terme de la liste initiale est égal à $(-1)+5$, soit `4`.

Plus graphiquement :

```
-1   ?   ?   ?     début de la liste initiale
   3  -3   5       liste des différences

-1   2   ?   ?     puisque 2=(-1)+3
   3  -3   5

-1   2  -1   ?     puisque -1=2+(-3)
   3  -3   5

-1   2  -1   4     puisque 4=(-1)+5
   3  -3   5
```

Question 1.8 – Écrire la signature et une définition d'une fonction, nommée `construction-additive`, qui, étant donnés un entier *a* et une liste *LD*, rend la liste dont le premier terme est *a* et dont la liste des différences est *LD*. Par exemple

```
(construction-additive -1 '(3 -3 5))  → (-1 2 -1 4)
```

Réponse [3/50]

```
;;; construction-additive : int * LISTE[int] -> LISTE[int]
;;; (construction-additive debut liste-diff) rend la liste dont le premier
;;; terme est debut et dont la liste des différences est liste-diff
(define (construction-additive debut liste-diff)
  (if (pair? liste-diff)
      (cons debut
            (construction-additive (+ debut (car liste-diff))
                                   (cdr liste-diff) ) )
      (list debut) ) )
```

Intéressons-nous maintenant à la liste `(1 2 4 7 11 16)`, exemple donné dans l'introduction. Ce n'est pas une progression arithmétique ; en revanche, la liste des différences, à savoir `(1 2 3 4 5)`, est une liste en progression arithmétique que l'on sait donc prolonger, par exemple en la liste `(1 2 3 4 5 6 7 8)`, ce qui permet de prolonger la liste initiale en la liste `(1 2 4 7 11 16 22 29 37)`.

Question 1.9 – Écrire la signature et une définition d'un semi-prédicat, `prolongement-arithmetique-2`, qui a comme données un entier naturel *n* et une liste d'entiers et qui rend la liste initiale complétée des termes qui la suivent logiquement afin de former une liste comprenant exactement *n* termes lorsque la liste des différences de la liste donnée est en progession arithmétique (et a au moins trois éléments) ; elle rend `#f` sinon. Par exemple,

```
(prolongement-arithmetique-2 10 '(1 2 4 7 11 16))  → (1 2 4 7 11 16 22 29 37 46)
(prolongement-arithmetique-2 10 '(1 2 17 34))  → #F
```

Réponse [3/50]

```
;;; prolongement-arithmetique-2 : nat * LISTE[int] -> LISTE[int] + #f
;;; (prolongement-arithmetique-2 n liste) rend une liste de longueur n
;;; dont le préfixe est liste et qui complète « logiquement » la
;;; liste à condition que sa liste des différences soit une liste en
;;; progression arithmétique. Elle rend #f sinon.
(define (prolongement-arithmetique-2 n liste)
  (let ((prol-ld (prolongement-arithmetique (- n 1)
    (liste-differences liste))))
    (if prol-ld
        (construction-additive (car liste) prol-ld)
        #f)))
```

Généralisations

On considère la fonction `prolongement-arithmetique-un-deux`, avec la définition suivante :

```
(define (prolongement-arithmetique-un-deux n liste)
  (if (au-moins-trois? liste)
      (let ((ld (liste-differences liste)))
        (if (tous-egaux? ld)
            (progression-arithmetique (car liste) (car ld) n)
            (let ((pld (prolongement-arithmetique (- n 1) ld)))
              (if pld
                  (construction-additive (car liste) pld)
                  #f) ) ) )
      #f) )
```

Question 1.10 – Quel est le résultat des évaluations des applications suivantes :
- `(prolongement-arithmetique-un-deux 8 '(1 2 3))`
- `(prolongement-arithmetique-un-deux 8 '(1 2 4 7 11))`
- `(prolongement-arithmetique-un-deux 8 '(1 2 17 34))`
- `(prolongement-arithmetique-un-deux 10 '(1 2 4))`

Réponse [1/50]

Résultat des évaluations des applications :
- (prolongement-arithmetique-un-deux 8 '(1 2 3)) → (1 2 3 4 5 6 7 8)
- (prolongement-arithmetique-un-deux 8 '(1 2 4 7 11)) → (1 2 4 7 11 16 22 29)
- (prolongement-arithmetique-un-deux 8 '(1 2 17 34)) → #F
- (prolongement-arithmetique-un-deux 10 '(1 2 4)) → #F

Question 1.11 – Donner la spécification de la fonction `prolongement-arithmetique-un-deux`.

Réponse [2/50]

Spécification de la fonction :

```
;;; prolongement-arithmetique-un-deux : nat * LISTE[int] -> LISTE[int] + #f
;;; (prolongement-arithmetique-un-deux n liste) rend une liste de longueur n
;;; dont le préfixe est liste et qui complète « logiquement » la liste à
;;; condition que liste soit en progression arithmétique ou que sa liste des
;;; différences soit une liste en progression arithmétique. Elle rend #f sinon.
```

La liste `(10 11 13 17 24 35 51)` a pour liste de différences `(1 2 4 7 11 16)`, qui a pour liste de différences `(1 2 3 4 5)`, qui est une liste en progression arithmétique.

Question 1.12 – Écrire une définition du semi-prédicat, nommé `prolongement-additif`, de même signature que `prolongement-arithmetique`, capable de compléter toute liste dont la liste des différences (itérée le bon

Section | Numéro d'anonymat

nombre de fois) est en progression arithmétique. Le semi-prédicat `prolongement-additif` répondra faux lorsque la liste donnée n'est pas prolongeable ou lorsqu'elle a moins de trois termes. Voici quatre exemples :

```
(prolongement-additif 8 '(1 3 5 7 9 11))  → (1 3 5 7 9 11 13 15)
(prolongement-additif 8 '(1 2 5 10 17 26))  → (1 2 5 10 17 26 37 50)
(prolongement-additif 8 '(1 2 4 9 19))  → (1 2 4 9 19 36 62 99)
(prolongement-additif 8 '(1 2 1 2 1 2))  → #F
```

Réponse [5/50]

```
(define (prolongement-additif n liste)
  (if (au-moins-trois? liste)
      (let ((ld (liste-differences liste)))
        (if (tous-egaux? ld)
            (progression-arithmetique (car liste) (car ld) n)
            (let ((pld (prolongement-additif (- n 1) ld)))
              (if pld
                  (construction-additive (car liste) pld)
                  #f ) ) ) )
      #f ) )
```

Comment prolonger la suite `(1 2 4 8 16)` ? On ne peut le faire avec les fonctions précédentes car c'est une suite en progression géométrique, où chaque terme est le double du précédent. Une progression géométrique est une suite telle que $u_{n+1} = ru_n$ où r (entier non nul) est nommé la « raison ».

Pour traiter ces suites, on pourrait écrire les fonctions `progression-geometrique`, `progression-geometrique?` et `liste-quotients`.

Question 1.13 – Généraliser les fonctions `progression-arithmetique` et `progression-geometrique` en la fonction `progression` telle que la propriété suivante soit vérifiée pour tout entier `a`, tout entier non nul `r` et tout entier naturel `n` :

```
(progression-arithmetique a r n) ≡ (progression + a r n)
(progression-geometrique a r n) ≡ (progression * a r n)
```

Réponse [3/50]

```
;;; progression : (int * int -> int) * int * int * nat -> LISTE[int]
;;; (progression op u0 r n) renvoie les n premiers termes de la
;;; progression arithmétique (lorsque op est égal à +) ou géométrique (lorsque
;;; op est égal à *) de raison r et d'élément initial u0.
(define (progression op u0 r n)
  (if (> n 0)
      (cons u0 (progression op (op u0 r) r (- n 1)))
      '() ) )
```

Exercice 2

Les arbres binaires de recherche vus en cours sont des cas particuliers des arbres binaires. Ce problème ne s'intéresse qu'aux arbres binaires d'étiquettes numériques.

Question 2.1 – Écrire un prédicat nommé `est-dans?` prenant un entier et un arbre binaire quelconque (mais étiqueté par des entiers), vérifiant si cet entier est l'une des étiquettes de cet arbre.

Réponse [3/50]

```
;;; est-dans? : Nombre * ArbreBinaire[Nombre] -> bool
(define (est-dans? n ab)
  (and (ab-noeud? ab)
       (or (= n (ab-etiquette ab))
           (est-dans? n (ab-gauche ab))
           (est-dans? n (ab-droit ab)) ) ) )
```

Section Numéro d'anonymat

Ce problème va s'intéresser aux arbres binaires de recherche mal fichus (ABRMF) dont les étiquettes sont toutes numériques (et toutes différentes) et qui sont ainsi définis :

Un ABRMF est un arbre binaire d'étiquettes numériques tel que :
- ses sous-arbres gauche et droit sont des ABRMF,
- soit l'étiquette de la racine est plus grande que toutes les étiquettes du sous-arbre gauche et plus petite que toutes les étiquettes du sous-arbre droit,
- soit l'étiquette de la racine est plus petite que toutes les étiquettes du sous-arbre gauche et plus grande que toutes les étiquettes du sous-arbre droit.

En d'autres termes, un ABRMF est presqu'un arbre binaire de recherche sauf que les sous-arbres gauche et droit sont parfois intervertis. Voici deux exemples :

```
       10                               10
      /  \                             /  \
    5      14                        30    8
   / \    /  \                      /  \
  7   3  16   11                   40   25
       \     /                            \
        4   13                             13
```

Question 2.2 – Écrire une fonction, nommée `arbre-orientation`, prenant un ABRMF et construisant l'arbre de ses orientations. L'arbre des orientations d'un ABRMF est un arbre de même structure sauf que ses étiquettes ne peuvent prendre que la valeur `normal` (resp. `inverse`) suivant que les étiquettes du sous-arbre gauche sont plus petites (resp. plus grandes) que l'étiquette de la racine. On impose que l'orientation soit `normal` lorsque les sous-arbres sont vides. Voici un exemple :

```
       10                      normal
      /  \                    /      \
    5      14          inverse        inverse
   / \    /  \         /     \        /     \
  7   3  16   11    normal normal normal inverse
     / \     /  \          /  \          /  \
        4   13               normal    normal
```

Réponse [4/50]

```
;;; arbre-orientation : ABRMF -> ArbreOrientation
(define (arbre-orientation abrmf)
  (if (ab-noeud? abrmf)
      (let ((gauche (ab-gauche abrmf)))
        (ab-noeud
         (if (ab-noeud? gauche)
             (if (< (ab-etiquette gauche) (ab-etiquette abrmf))
                 'normal
                 'inverse )
             (if (ab-noeud? droit)
                 (if (> (ab-etiquette droit) (ab-etiquette abrmf))
                     'normal
                     'inverse )
                 'normal ) )
         (arbre-orientation gauche)
         (arbre-orientation (ab-droit abrmf)) ) )
      (ab-vide) ) )
```

Question 2.3 – Écrire un prédicat, nommé `abrmf-est-dans?`, prenant un entier n, un ABRMF et son arbre d'orientations et vérifiant s'il existe une étiquette égale au nombre n dans l'ABRMF.

Section Numéro d'anonymat

Réponse [5/50]

```
;;; abrmf-est-dans? : Nombre * ABRMF * ArbreOrientation -> bool
(define (abrmf-est-dans? n abrmf orientations)
  (if (ab-noeud? abrmf)
      (cond
        ((= n (ab-etiquette abrmf)) #t)
        ((< n (ab-etiquette abrmf))
         (if (equal? 'normal (ab-etiquette orientations))
             (abrmf-est-dans? n (ab-gauche abrmf) (ab-gauche orientations))
             (abrmf-est-dans? n (ab-droit abrmf) (ab-droit orientations))))
        (else
         (if (equal? 'normal (ab-etiquette orientations))
             (abrmf-est-dans? n (ab-droit abrmf) (ab-droit orientations))
             (abrmf-est-dans? n (ab-gauche abrmf) (ab-gauche orientations)))))
      #f ) )
```

Question 2.4 – Tout arbre binaire de recherche est un ABRMF ! Mais l'inverse est faux. Écrire le prédicat `est-abr?` prenant un arbre d'orientations et vérifiant qu'il correspond à celui d'un arbre binaire de recherche.

Réponse [3/50]

```
;;; est-abr? : ArbreOrientation -> bool
(define (est-abr? orientations)
  (if (ab-noeud? orientations)
      (and (equal? 'normal (ab-etiquette orientations))
           (est-abr? (ab-gauche orientations))
           (est-abr? (ab-droit orientations)) )
      #t ) )
```

Question 2.5 – Écrire une fonction, nommée `abrmf->abr`, prenant un ABRMF et son arbre d'orientations et construisant l'arbre binaire de recherche équivalent.

```
        10                       normal                           10
       /  \                     /       \                        /  \
     5      14             inverse      inverse                5      14
   /  \    /  \            /     \       /     \             /  \    /  \
  7    3  16   11      normal normal normal inverse        3    7  11  16
      / \     / \              /  \            /  \       / \        \
         4  13                 normal       normal           4        13
```

Réponse [4/50]

```
;;; abrmf->abr : ABRMF * ArbreOrientation -> ArbreBinRecherche
(define (abrmf->abr abrmf orientations)
  (if (ab-noeud? orientations)
      (if (equal? 'normal (ab-etiquette orientations))
          (ab-noeud (ab-etiquette abrmf)
                    (abrmf->abr (ab-gauche abrmf) (ab-gauche orientations))
                    (abrmf->abr (ab-droit abrmf) (ab-droit orientations)) )
          (ab-noeud (ab-etiquette abrmf)
                    (abrmf->abr (ab-droit abrmf) (ab-droit orientations))
                    (abrmf->abr (ab-gauche abrmf) (ab-gauche orientations)) ) )
      abrmf ) )
```