Examen – Module d'informatique 1 – Janvier 2003
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 11 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. Toutes les fonctions apparaissant dans les réponses doivent être accompagnées de leur spécification sauf lorsqu'indiqué différemment.

Ne pas désagrafer les feuilles.

Exercice 1

Question 1.1 – On considère une fonction nommée `premiers` prenant une liste de listes (non vides) d'entiers et retournant la liste de leurs premiers termes. Ainsi

```
(premiers '((1 2 3) (11 22) (5 6)))  → (1 11 5)
```

Écrire la signature de la fonction `premiers` puis écrire successivement deux définitions de la fonction `premiers`, l'une utilisant un itérateur (une fonctionnelle) vu en cours, l'autre ne l'utilisant pas.

[5/50]

Section Numéro d'anonymat

Question 1.2 – Écrire une fonction nommée `map2` (la signature et la définition seulement) prenant une fonction f (acceptant deux arguments) et deux listes L1 et L2. Cette fonction construit une nouvelle liste dont les termes sont les résultats des applications de f aux termes successifs et de même rang de L1 et L2. Si les listes diffèrent par la taille, les termes superflus ne sont pas pris en compte. Ainsi,

```
(map2 - '(11 22 33) '(1 2 3))  → (10 20 30)
(map2 max '(1 2) '(10 1 12))  → (10 2)
```

[3/50]

Question 1.3 – Écrire une fonction nommée `associations` (la signature et la définition seulement) prenant une liste de clés et une liste de valeurs et construisant la liste d'associations associant ces clés à ces valeurs. On supposera que ces listes ont même taille. Ainsi,

```
(associations '(un deux trois) '(1 2 3))  → ((un 1) (deux 2) (trois 3))
```

[3/50]

Section Numéro d'anonymat

Exercice 2

Soit une pièce de monnaie dont on ne sait comment elle est truquée. On lance cette pièce un grand nombre de fois et on enregistre si elle est tombée sur pile ou face dans une grande liste ne contenant que les symboles `pile` ou `face`. On désire fabriquer, à partir de cette liste, une liste de booléens aléatoires équiprobables.

Question 2.1 – Écrire un prédicat `au-moins-2-termes?` prenant une liste en argument et vérifiant si cette liste possède au moins deux termes.

[1/50]

Soit la fonction nommée `alea1` prenant une liste de symboles `pile` ou `face` et calculant une liste de booléens : on considère les symboles deux par deux et, pour chaque couple, lorsqu'il est constitué des symboles `pile face`, on produira le booléen `#t`, et lorsqu'il est constitué des symboles `face pile` on produira le booléen `#f`, dans tous les autres cas on ne produit rien. Ainsi,

```
(alea1 '(pile face pile pile face face face pile face pile))  →  (#T #F #F)
(alea1 '(pile face pile))  →  (#T)
```

Question 2.2 – Quelle est la valeur de la fonction `alea1` appliquée à la liste `(pile face face pile pile pile face face face face)`.

[1/50]

Question 2.3 – Écrire une fonction nommée `alea1` (la signature et la définition seulement) implantant la spécification énoncée plus haut.

[3/50]

Question 2.4 – Écrire une fonction (la définition seulement) nommée `reste-alea1` prenant une liste de symboles `pile` ou `face` et calculant une nouvelle liste de symboles `pile` ou `face` définie comme suit : on considère les symboles deux par deux et, pour chaque couple, lorsqu'il est constitué de deux fois le même symbole, on produira ce symbole non répété. Ainsi,

Section Numéro d'anonymat

```
(reste-alea1 '(pile face pile pile face face face pile face pile))  → (pile face)
(reste-alea1 '(pile pile pile pile))  → (pile pile)
(reste-alea1 '(pile pile pile pile face))  → (pile pile)
```

[3/50]

Question 2.5 – Écrire une fonction nommée `alea2` prenant une liste de symboles `pile` ou `face` et calculant une liste de booléens : ceux obtenus à partir d'`alea1` suivis par ceux obtenus en appliquant `alea1` sur la nouvelle liste calculée par `reste-alea1`. Ainsi

```
(alea2 '(pile face))  → (#T)
(alea2 '(pile face face))  → (#T)
(alea2 '(pile pile face face))  → (#T)
(alea2 '(pile pile face face pile))  → (#T)
(alea2 '(face pile pile pile face face face pile))  → (#F #F #T)
```

[3/50]

Exercice 3

On désire analyser le jeu dit du *Tic-Tac-Toe* qui se joue sur un échiquier 3x3. Les joueurs sont identifiés par leur marque : les symboles O et X, ils jouent chacun leur tour et O commence. À chaque tour, un joueur doit cocher avec sa marque (O ou X) l'une des cases restées vide. Un joueur gagne lorsqu'il réalise un alignement horizontal, vertical ou diagonal de 3 de ses marques. La partie est nulle lorsque toutes les cases sont prises et qu'aucun alignement n'a été réalisé.

Les marques appartiendront au type `Marque` constitué des symboles `X` et `O`. Par abus de notation, « joueur » et « marque » seront confondus dans toute la suite.

Chaque case de l'échiquier est repérée par deux entiers naturels (abscisse et ordonnée) variant entre 1 et 3.

L'échiquier sera manipulé au travers d'une barrière d'abstraction dont voici les spécifications :

`;;;` *echiquier-vide: -> Echiquier*

Section | Numéro d'anonymat

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

FIG. 1 – Repérage des cases dans l'échiquier

```
;;; (echiquier-vide) rend un échiquier vide.

;;; cocher: Echiquier * nat/1à3/ * nat/1à3/ * Marque -> Echiquier
;;; ERREUR si la case est déjà marquée.
;;; (cocher echiquier i j m) rend un échiquier égal en tout point à
;;; l'argument echiquier sauf qu'en (i,j) se trouve la marque m.

;;; marque: Echiquier * nat/1à3/ * nat/1à3/ -> Marque + #f
;;; (marque echiquier i j) rend la marque de la case (i,j) ou #f s'il
;;;  n'y a pas de marque sur cette case.
```

Voici quelques échiquiers correspondant à un début de partie :

X		

X		
	O	

X		
X	O	

X		
X	O	
O		

X		
X	O	
O		X

FIG. 2 – Un début de partie (et l'échiquier exemple pour la suite)

Question 3.1 – Écrire, pour chacun des trois premiers échiquiers de la figure 2, une expression qui le construit.

[2/50]

Question 3.2 – Écrire une fonction nommée `autre-marque` prenant en argument une marque (resp. le symbole `X` ou `O`) et calculant l'autre marque (resp. le symbole `O` ou `X`).

[1/50]

Section   Numéro d'anonymat

Question 3.3 – Compléter la fonction `echiquier->paragraphe` suivante dont voici un exemple d'emploi (on suppose que la fonction `echiquier-exemple` rend l'échiquier de droite de la figure 2) :

```
(echiquier->paragraphe (echiquier-exemple))  → "
X
XO
O X
"
```

```
;;; echiquier->paragraphe: Echiquier -> Paragraphe
;;; (echiquier->paragraphe echiquier) rend un paragraphe representant l'échiquier.

(define (echiquier->paragraphe echiquier)
  ;; ligne: nat -> string
  ;; (ligne i) rend la chaîne correspondant à la i-ème ligne de l'échiquier.
  (define (ligne i)
    ;; colonne: nat -> string
    ;; (colonne j) rend la chaîne correspondant au contenu de la case i,j de l'échiquier.
    (define (colonne j)
      (let ((c (marque echiquier i j)))
        (cond ((equal? c 'X) "X")
              ((equal? c 'O) "O")
              (else " ") ) ) )
     ;; expression de (ligne i):
```

[1,5/50]

)

```
  ;; expression de (echiquier->paragraphe echiquier):
```

[1,5/50]

)

Section Numéro d'anonymat

Les coups des joueurs seront représentés par des valeurs de type `Coup` dont voici la barrière d'abstraction :

```
;;; coup: Marque * nat/1à3/ * nat/1à3/ -> Coup
;;; avec Marque = O ou X
;;; (coup m i j) représente un coup de marque m en case i,j.

;;; coup-marque: Coup -> Marque
;;; (coup-marque c) rend la marque du joueur effectuant le coup.

;;; coup-i: Coup -> nat/1à3/
;;; (coup-i c) rend la ligne où le joueur joue.

;;; coup-j: Coup -> nat/1à3/
;;; (coup-j c) rend la colonne où le joueur joue.
```

Question 3.4 – À quelle(s) grande(s) famille(s) de fonction(s) (constructeur, reconnaisseur, ...) appartiennent les fonctions `coup`, `coup-marque`, `coup-i` et `coup-j` ?

[1/50]

Question 3.5 – Compléter la fonction `liste-coups-imaginables` suivante :

```
;;; liste-coups-imaginables: Marque -> LISTE[Coup]
;;; (liste-coups-imaginables m) liste les neuf valeurs imaginables du
;;; type Coup pour un joueur m donné.

(define (liste-coups-imaginables m)
  (let ((liste-coords '((1 1) (1 2) (1 3)
                        (2 1) (2 2) (2 3)
                        (3 1) (3 2) (3 3) )))
    ;; coup-m: NUPLET[nat/1à3/ nat/1à3/] -> Coup
    ;; (coup-m (list i j)) rend le coup de marque m en i, j.
    (define (coup-m coords)
      ;; expression de (coup-m coords):
                                                        [2/50]
      )
    (map coup-m liste-coords) ) )
```

Section Numéro d'anonymat

Question 3.6 – Donnez les spécifications de la fonction `liste-coups-possibles` et de la fonction interne `possible?` ainsi définie :

```
(define (liste-coups-possibles echiquier m)
  (define (possible? c)
    (not (marque echiquier (coup-i c) (coup-j c))) )
  (filtre possible?
          (liste-coups-imaginables m) ) )
```

Donnez également la valeur attendue pour l'expression :

```
(liste-coups-possibles (echiquier-exemple) 'O)
```

[4/50]

Question 3.7 – Écrire une fonction nommée `liste-coups-utiles` (la définition seulement) qui prend un échiquier et un joueur et qui calcule la liste des seuls coups possibles que peut jouer ce joueur sur cet échiquier. Cette liste sera bien sûr vide si l'autre joueur vient de gagner sur cet échiquier.

On supposera disposer de la fonction `gagne?` de spécification :

;;; *gagne?: Echiquier * Marque -> bool*
;;; *(gagne? echiquier m) vérifie sur l'échiquier «echiquier» si le joueur de marque «m» a gagné.*

[2/50]

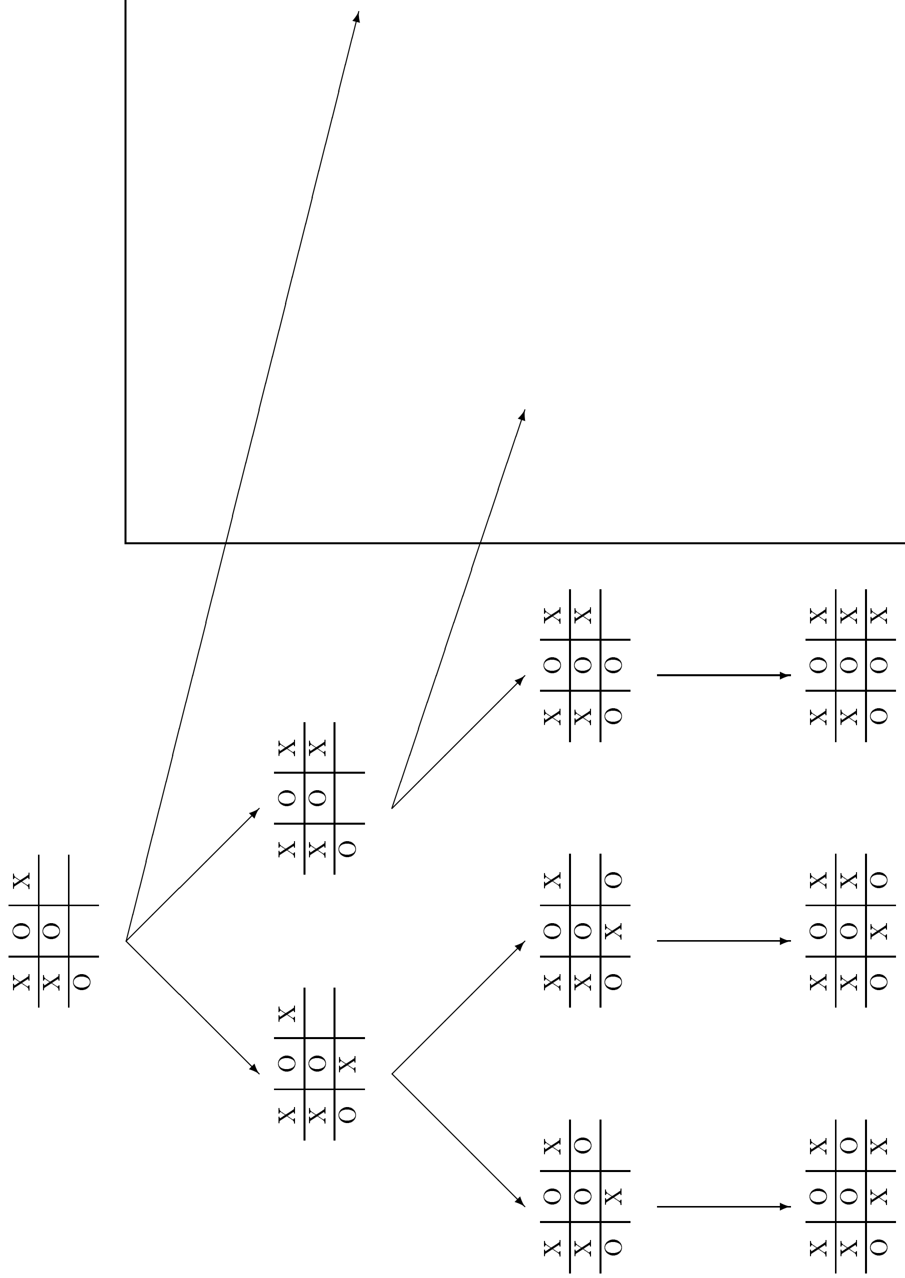
Question 3.8 – On souhaite représenter toutes les parties possibles par un « arbre des parties » qui est un arbre général dont l'étiquette est l'échiquier qu'un joueur reçoit. Les fils de ce nœud sont les arbres des parties correspondant à chaque coup possible que ce joueur peut jouer. Les feuilles de cet arbre sont toutes des échiquiers totalement remplis. Étant donné un arbre général, vous avez le droit d'utiliser toutes les fonctions de la barrière d'abstraction des arbres généraux.

Voici par exemple une représentation d'un arbre de parties que vous aurez à compléter sur ses parties manquantes.

Section

Numéro d'anonymat

[3/50]



Section Numéro d'anonymat

Question 3.9 – On souhaite élaguer l'arbre des parties de toutes les prolongations de parties gagnées. Écrire la fonction nommée `elaguer` (la signature et la définition seulement) qui prend un arbre de parties et calcule un arbre de parties tel que les nœuds, qui ne sont pas des feuilles, ne soient pas des parties gagnées. Y-a-t'il une partie élaguable sur la partie imprimée de l'arbre des parties précédent ?

[3/50]

Question 3.10 – Écrire un prédicat nommé `peut-gagner?` (la signature et la définition seulement) qui prend un arbre de parties et une marque et qui vérifie qu'il existe au moins une partie où le joueur possédant cette marque gagne.

[3/50]

Section Numéro d'anonymat

Question 3.11 – Écrire une fonction nommée `nombre-peut-gagner` (la signature et la définition seulement) qui prend un arbre de parties et une marque et qui rend le nombre de parties de cet arbre où le joueur possédant cette marque gagne.

[3/50]