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.

Réponse [5/50]

```
;;; premiers: LISTE[LISTE[int]/non vide/] -> LISTE[int]
;;; (premiers L) rend la liste de tous les premiers termes des termes de L.
(define (premiers1 L)
  (map car L) )

(define (premiers2 L)
  (if (pair? L)
      (cons (car (car L)) (premiers2 (cdr L)))
      '() ) )
```

Notez le type de l'argument de `premiers`, ce n'est pas équivalent à `LISTE[LISTE[int]]/non vide/ -> LISTE[int]`.

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)
```

Section | Numéro d'anonymat

Réponse [3/50]

```
;;; map2: (alpha * beta -> gamma) * LISTE[alpha] * LISTE[beta] -> LISTE[gamma]
;;; (map2 f (list e1 e2 ... eP) (list f1 f2 ... fQ)) rend (list (f e1 f1)
;;; (f e2 f2) ... (f eR fR)) avec R = min(P, Q).
(define (map2 f L1 L2)
  (if (and (pair? L1)
           (pair? L2) )
      (cons (f (car L1) (car L2))
            (map2 f (cdr L1) (cdr L2)) )
      '() ) )
```

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))`

Réponse [3/50]

```
;;; associations: LISTE[alpha] * LISTE[beta] -> LISTE[NUPLET[alpha beta]]
;;; (associations cles valeurs) rend la liste d'association associant les clés
;;; de type alpha aux valeurs de type beta.
(define (associations cles valeurs)
  (map2 list cles valeurs) )
```

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.

Réponse [1/50]

```
;;; au-moins-2-termes?: LISTE[alpha] -> bool
;;; (au-moins-2-termes? L) vérifie que L a au moins deux termes.
(define (au-moins-2-termes? L)
  (and (pair? L)
       (pair? (cdr L)) ) )
```

Attention, une proposition de solution telle que

```
(define (au-moins-2-termes? L)
  (if (pair? L) (pair? (cdr L))) )
```

n'est pas correcte puisque, lorsque la liste est vide, le résultat est n'importe quoi et non pas `#t`.

La proposition suivante est pléonastique :

```
(define (au-moins-2-termes? L)
  (if (pair? L)
      (if (pair? (cdr L))
          #t
          #f )
      #f ) )
```

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)`

Section	Numéro d'anonymat

(alea1 '(pile face pile)) $\rightarrow$ (#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)`.

Réponse [1/50]

(alea1 '(pile face face pile pile pile face face face face)) $\rightarrow$ (#T #F)

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

Réponse [3/50]

```
;;; alea1: LISTE[Symbole] -> LISTE[booleen]
;;; (alea1 L) construit une liste de booléens équiprobables à partir d'une
;;; liste de pile ou face obtenues à partir d'une pièce possiblement truquée.
(define (alea1 L)
  (if (au-moins-2-termes? L)
      (let ((un   (car L))
            (deux (cadr L)) )
        (if (equal? un deux)
            (alea1 (cddr L))
            (cons (equal? un 'pile)
                  (alea1 (cddr L)) ) ) )
      '() ) )
```

Pour en savoir plus: Si une pièce de monnaie est truquée (mais on ne sait pas comment) et que la probabilité de sortir pile est p (et donc la probabilité de sortir face est $1-p$). La pièce étant truquée, p n'est pas égal à $1/2$ et on ne connait même pas p. Pour fabriquer de l'aléatoire non biaisé, on reprend une idée attribuée à Von Neumann qui est d'utiliser la symétrie. On considère les couples formés de deux lancers consécutifs. Si le couple est (pile, face) alors on produit le booléen `#t`, si le couple est (face, pile) on produit le booléen `#f` et si le couple est autre, on l'élimine : il ne produit rien. La probabilité d'émettre un booléen Vrai est égale à la probabilité d'émettre un booléen Faux (elle est égale à $p(1-p)$), on a donc constitué une suite de booléens équiprobables à partir d'une pièce truquée.

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,

(reste-alea1 '(pile face pile pile face face face pile face pile)) $\rightarrow$ (pile face)
(reste-alea1 '(pile pile pile pile)) $\rightarrow$ (pile pile)
(reste-alea1 '(pile pile pile pile face)) $\rightarrow$ (pile pile)

Section Numéro d'anonymat

Réponse	[3/50]

```
;;; reste-alea1: LISTE[Symbole] -> LISTE[Symbole]
;;; (reste-alea1 L) construit une liste de symboles pile ou face à partir des paires
;;; répétées de tels symboles.
(define (reste-alea1 L)
  (if (au-moins-2-termes? L)
      (let ((un   (car L))
            (deux (cadr L)) )
        (if (equal? un deux)
            (cons un (reste-alea1 (cddr L)))
            (reste-alea1 (cddr L)) ) )
      '() ) )
```

Pour en savoir plus: Cette fonction permet d'exploiter les couples répétés que n'exploitait pas la fonction `alea1` ce qui permet d'extraire un peu plus d'aléatoire de la liste de lancers initiale. En effet, `alea1` n'émet un booléen que lorsque les lancers changent de sens. La probabilité de cet événement n'est que de $2p(1-p)$ ce qui veut dire que, si la pièce n'est pas truquée, alors on ne sort qu'un seul booléen tous les deux lancers : la perte est donc de 50%.
L'idée, due à Yuval Peres, est d'utiliser les couples éliminés (pile, pile) ou (face, face) pour engendrer un peu plus d'aléatoire. Chaque fois qu'un couple est éliminé, on conserve le lancer répété et l'on constitue ainsi une nouvelle liste de symboles `pile` ou `face` sur laquelle on pourra encore appliquer `alea1`.

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)
```

Réponse	[3/50]

```
;;; alea2: LISTE[Symbole] -> LISTE[booleen]
;;; (alea2 L) construit une liste de booléens aléatoires à partir d'une liste de
;;; pile ou face obtenues à partir d'une pièce possiblement truquée.
(define (alea2 L)
  (append (alea1 L)
          (alea1 (reste-alea1 L)) ) )
```

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.

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

Section Numéro d'anonymat

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

;;; *echiquier-vide: -> Echiquier*
;;; *(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.

Réponse [2/50]

```
(cocher (echiquier-vide) 1 1 'X)
(cocher (cocher (echiquier-vide) 1 1'X ) 2 2 'O)
(cocher (cocher (cocher (echiquier-vide) 1 1 'X) 2 2 'O) 2 1 'X)
```

Attention, une proposition de solution telle que :

```
(and (cocher (echiquier-vide) 1 1 'X)
     (cocher (echiquier-vide) 2 2 'O) )
```

est incorrecte car elle vise à produire deux échiquiers indépendants chacun doté d'une unique marque (de plus, un échiquier ne pouvant être égal à `#f` seul le premier échiquier est créé et rendu comme valeur de la forme `and`).

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`).

Réponse [1/50]

```
;;; autre-marque: Marque -> Marque
;;; (autre-marque m) calcule la marque de l'autre joueur (X si O et O si X).
(define (autre-marque m)
  (if (equal? m 'O) 'X 'O) )
```

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.*

Section	Numéro d'anonymat

```
(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]
     (string-append (colonne 1) (colonne 2) (colonne 3))  )
  ;; expression de (echiquier->paragraphe echiquier):
  [1,5/50]
  (paragraphe (list (ligne 1) (ligne 2) (ligne 3)))  )
```

```
(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 " ") ) ) )
    (string-append (colonne 1) (colonne 2) (colonne 3)) )
  (paragraphe (list (ligne 1) (ligne 2) (ligne 3))) )
```

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` ?

Réponse [1/50]
Les fonctions `coup-marque`, `coup-i` et `coup-j` sont des accesseurs ou sélecteurs qui extraient de l'information à partir d'un coup. La fonction `coup` est un constructeur.

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
```

Section Numéro d'anonymat

```
;;; 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]

```
      (coup m (car coords) (cadr coords))
                                                        )
    (map coup-m liste-coords) ) )
```

```
(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)
      (coup m (car coords) (cadr coords)) )
    (map coup-m liste-coords) ) )
```

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)
```

Réponse [4/50]

```
;;; liste-coups-possibles: Echiquier * Marque -> LISTE[Coup]
;;; (liste-coups-possibles echiquier m) liste les seuls coups possibles sur
;;; un échiquier donné.

(define (liste-coups-possibles echiquier m)
  ;; possible?: Coup -> bool
  ;; (possible? c) vérifie que la case du coup prévu est libre.
  (define (possible? c)
    (not (marque echiquier (coup-i c) (coup-j c))) )
  (filtre possible?
          (liste-coups-imaginables m) ) )
```

Sur l'échiquer exemple, l'expression donne :

```
(liste-coups-possibles (echiquier-exemple) 'O) ≡
(list (coup 'O 1 2) (coup 'O 1 3) (coup 'O 2 3) (coup 'O 3 2))
```

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
```

Section Numéro d'anonymat

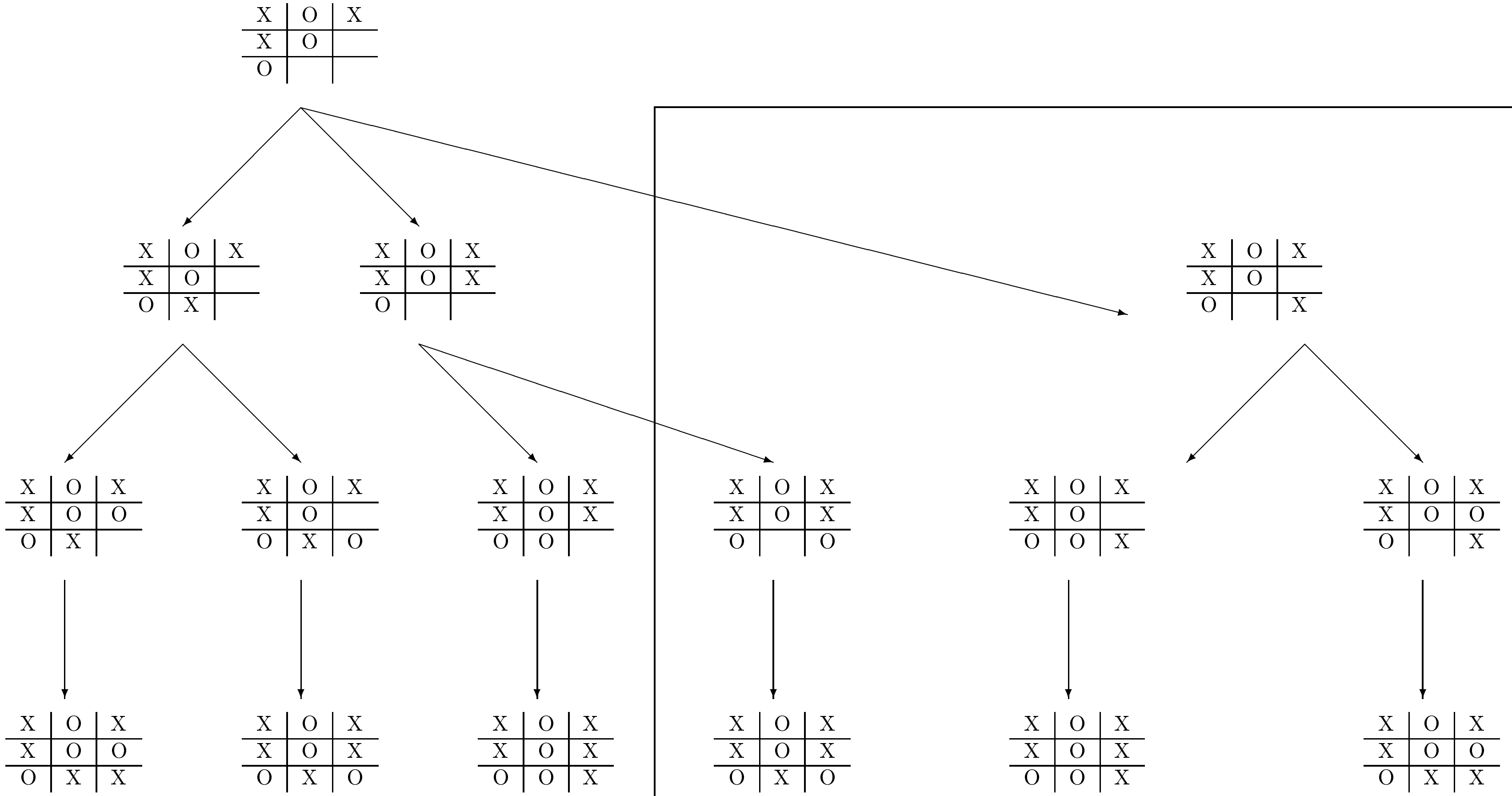
; ; ; *(gagne? echiquier m) vérifie sur l'échiquier «echiquier» si le joueur de marque «m» a gagné.*

Réponse [2/50]

```
; (define (liste-coups-utiles echiquier joueur)
;   (define (possible? coup)
;     (not (marque echiquier (coup-i coup) (coup-j coup))) )
;   (if (gagne? echiquier (autre-marque joueur))
;       '()
;       (filtre possible?
;               (liste-coups-imaginables joueur) ) ) )
(define (liste-coups-utiles echiquier joueur)
  (if (gagne? echiquier (autre-marque joueur))
      '()
      (liste-coups-possibles echiquier joueur) ) )
```

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.



[3/50]

Section

Numéro d'anonymat

TSVP

Section	Numéro d'anonymat

Pour en savoir plus: Voici au passage la fonction `arbre-parties` qui prend un échiquier et un joueur et qui construit l'arbre des parties partant de cet échiquier où c'est joueur qui doit jouer en premier. Ainsi, l'arbre précédemment esquissé est-il obtenu par `(arbre-parties e 'O)` ou `e` est l'échiquier représenté en haut.

```
;;; arbre-parties: Echiquier * Marque -> ArbreGeneral[Echiquier]
;;; (arbre-parties echiquier m) calcule l'arbre de toutes les séquences
;;; d'échiquiers. L'étiquette est l'échiquier que le joueur de marque m reçoit
;;; et sur lequel (s'il n'a pas gagne) il va jouer (cocher une case).
(define (arbre-parties echiquier m)
  (define (depliement coup)
    (arbre-parties
     (cocher echiquier (coup-i coup) (coup-j coup) (coup-marque coup))
     (autre-marque m) ) )
  (ag-noeud echiquier
            (map depliement
                 (liste-coups-possibles echiquier m) ) ) )
```

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 ?

Réponse [3/50]

```
;;; elaguer: ArbreGeneral[Echiquier] -> ArbreGeneral[Echiquier]
;;; (elaguer arbre-parties) élaguer les branches suivant une position
;;; gagnée pour l'un ou l'autre des joueurs.
(define (elaguer arbre-parties)
  (let ((echiquier (ag-etiquette arbre-parties)))
    (if (or (gagne? echiquier 'O)
            (gagne? echiquier 'X) )
        (ag-noeud echiquier (list))
        (ag-noeud echiquier
                  (map elaguer (ag-foret arbre-parties)) ) ) ) )
```

Le troisième échiquier du bas en partant de la gauche est inutile car la partie a déjà été gagnée par le joueur O juste au-dessus.

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.

Section Numéro d'anonymat

Réponse [3/50]

```
;;; peut-gagner?: ArbreGeneral[Echiquier] * Marque -> bool
;;; (peut-gagner? arbre-parties m) vérifie que le joueur de marque m peut
;;; gagner (probablement si l'autre joueur joue très très mal).
(define (peut-gagner? arbre-parties m)
  ;; peut-encore-gagner?: ArbreGeneral[Echiquier] -> bool
  ;; (peut-encore-gagner? a) vérifie que le joueur m peut encore gagner avec a.
  (define (peut-encore-gagner? arbre-parties)
    (peut-gagner? arbre-parties m) )
  (or (gagne? (ag-etiquette arbre-parties) m)
      (existe? peut-encore-gagner? (ag-foret arbre-parties)) ) )

;;; existe?: (alpha -> bool) * LISTE[alpha] -> bool
;;; (existe? p L) vérifie qu'au moins un terme de L satisfait le prédicat p.
(define (existe? p L)
  (if (pair? L)
      (or (p (car L))
          (existe? p (cdr L)) )
      #f ) )
```

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.

Réponse [3/50]

```
;;; nombre-peut-gagner: ArbreGeneral[Echiquier] * Marque -> nat
;;; (nombre-peut-gagner arbre-parties m) compte le nombre de parties que peut
;;; gagner le joueur ayant la marque m (probablement si l'autre joueur joue très très mal).
(define (nombre-peut-gagner arbre-parties m)
  (define (nombre-peut-encore-gagner partie somme)
    (+ (nombre-peut-gagner partie m)
       somme ) )
  (if (gagne? (ag-etiquette arbre-parties) m)
      1
      (reduce nombre-peut-encore-gagner
              0
              (ag-foret arbre-parties) ) ) )
```