Examen – Module d'informatique 1 – Janvier 2001
MIAS 1ère année – 1er semestre

Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.

L'examen dure deux heures.

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 barême (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. Ne pas désagrafer les feuilles.

Exercice 1

La grammaire suivante décrit un labyrinthe :

<*labyrinthe*> → (<*numero*> `vers` <*suivants*>)

<*numero*> → *`un entier naturel`*

<*suivants*> → <*labyrinthe*>*

Un labyrinthe est supposé formé d'une pièce (identifiée par un numéro) dont les portes mènent vers d'autres labyrinthes. Toutes les pièces sont supposées identifiées par des numéros différents.

La barrière syntaxique autour des labyrinthes se compose de trois fonctions : un constructeur `labyrinthe` et deux sélecteurs `labyrinthe-numero` et `labyrinthe-suivants`. Le sélecteur `labyrinthe-numero` prend un labyrinthe et renvoie le numéro de la première pièce du labyrinthe tandis que le sélecteur `labyrinthe-suivants` renvoie la liste des labyrinthes auxquels mène la première pièce du labyrinthe.

Par exemple, le labyrinthe `(1 vers (2 vers (3 vers)))` comporte trois pièces : la première mène à la pièce 2 qui mène à la pièce 3 qui est un cul-de-sac. Le labyrinthe `(1 vers (2 vers) (3 vers))` comporte également trois pièces mais la première mène aux pièces 2 et 3 qui sont toutes deux des culs-de-sac.

Question 1.1 – On suppose, pour l'instant, disposer du constructeur de labyrinthe suivant : `labyrinthe` dont la spécification est :

;;; *labyrinthe : nat * LISTE[Labyrinthe] -> Labyrinthe*
;;; *(labyrinthe n labs) renvoie le labyrinthe formé de la pièce n, menant*
;;; *aux labyrinthes labs.*

Quelle est la valeur de l'expression suivante :

```
(let* ((L1 (labyrinthe 1 '()))
       (L2 (labyrinthe 2 '()))
       (L3 (labyrinthe 3 (list L1 L2)))
       (L4 (labyrinthe 4 (list))))
  (labyrinthe 5 (list L3 L4)) )
```

Section | Numéro d'anonymat

Réponse [2/50]

La valeur est :

```
(5 vers (3 vers (1 vers) (2 vers)) (4 vers))
```

Question 1.2 – Écrire une définition possible du constructeur `labyrinthe`.

Réponse [3/50]

```
;;; labyrinthe : nat * LISTE[Labyrinthe] -> Labyrinthe
;;; (labyrinthe n labs) renvoie le labyrinthe formé de la pièce n, menant
;;; aux labyrinthes labs.
(define (labyrinthe n labs)
  (cons n (cons 'vers labs)) )
```

Écrire les fonctions `labyrinthe-numero` et `labyrinthe-suivants` de la barrière d'abstraction.

Réponse [2/50]

```
;;; labyrinthe-numero: Labyrinthe -> nat
;;; (labyrinthe-numero d) rend le numéro de la première pièce du labyrinthe.
(define (labyrinthe-numero d)
  (car d) )

;;; labyrinthe-resultats: Labyrinthe -> LISTE[Labyrinthe]
;;; (labyrinthe-resultats d) rend la liste des labyrinthes auxquels mène
;;; la première pièce du labyrinthe.
(define (labyrinthe-suivants d)
  (cddr d) )
```

Question 1.3 – Soit la fonction `tx` ainsi définie :

```
(define (tx d)
  (define (txs ds)
    (if (pair? ds)
        (+ (tx (car ds))
           (txs (cdr ds)) )
        0 ) )
  (+ 1 (txs (labyrinthe-suivants d))) )
```

Que vaut cette fonction appliquée au labyrinthe de la question 1 ? Écrire la spécification de la fonction `tx` ainsi que celle de sa fonction interne :

Réponse [5/50]

```
(tx '(5 vers (3 vers (1 vers) (2 vers)) (4 vers))) → 5

;;; tx: Labyrinthe -> nat
;;; (tx d) calcule le nombre de pièces présentes dans le labyrinthe d.

  ;; txs: LISTE[Labyrinthe] -> nat
  ;; (txs ds) calcule le nombre de pièces présentes dans les labyrinthes ds.
```

Rappelons qu'écrire `(cons` α β`)` est beaucoup plus simple (et efficace) que `(append (list` α`)` β`)`.

Question 1.4 – Écrire une fonction, nommée `liste-pieces`, qui prend un labyrinthe et rend la liste de tous les numéros de pièces qui s'y trouvent. L'ordre de collecte n'est pas important. Ainsi

Section Numéro d'anonymat

```
(liste-pieces '(1 vers (2 vers (4 vers)) (3 vers))) → (1 2 4 3)
```

Réponse [4/50]

```
;;; liste-pieces: Labyrinthe -> LISTE[nat]
;;; (liste-pieces d) rend la liste de tous les numeros de pièces se trouvant
;;; dans le labyrinthe d.
(define (liste-pieces d)
  (define (collecte ds)
    (if (pair? ds)
        (append (liste-pieces (car ds))
                (collecte (cdr ds)) )
        '() ) )
  (cons (labyrinthe-numero d)
        (collecte (labyrinthe-suivants d)) ) )

(define (liste-pieces2 d)
  (cons (labyrinthe-numero d)
        (reduce append '() (map liste-pieces2 (labyrinthe-suivants d))) ) )
```

Question 1.5 – Écrire un semi-prédicat nommé `sous-labyrinthe` qui prend un numéro de pièce n ainsi qu'un labyrinthe d et qui renvoie le labyrinthe partant de cette pièce n quelque part dans d. Le semi-prédicat `sous-labyrinthe` renvoie faux s'il n'y a pas de pièce n dans le labyrinthe d. Par exemple,

```
(sous-labyrinthe 3 '(1 vers (2 vers (4 vers)) (3 vers))) → (3 vers)
(sous-labyrinthe 5 '(1 vers (2 vers (4 vers)) (3 vers))) → #F
```

Réponse [4/50]

```
;;; sous-labyrinthe: nat * Labyrinthe -> Labyrinthe + #f
;;; (sous-labyrinthe n d) renvoie le sous labyrinthe dans d de pièce n.
(define (sous-labyrinthe n d)
  (define (recherche ds)
    (if (pair? ds)
        (or (sous-labyrinthe n (car ds))
            (recherche (cdr ds)) )
        #f ) )
  (if (= n (labyrinthe-numero d))
      d
      (recherche (labyrinthe-suivants d)) ) )
```

Exercice 2

Ce problème traite de NPRAG c'est-à-dire de nombres premiers récursifs à gauche. Le nombre 1 est considéré comme un NPRAG. Un entier naturel, noté $\overline{a_n a_{n-1} \ldots a_1 a_0}$ en base 10 avec a_n différent de zéro, est un NPRAG s'il est premier et si tous les nombres s'écrivant $\overline{a_i \ldots a_0}$ sont des NPRAG pour $0 \leq i \leq n$. Voici les NPRAG supérieurs à 10 et inférieurs à 1000 (il y en a bien d'autres) :

```
1  2  3  5  7

 11  31  41  61  71  13  23  43  53  73  83  17  37  47  67  97

 211  311  811  911  131  331  431  631  241  541  641  941  461  661
 761  271  571  971  113  313  613  223  523  823  443  643  743  353  653
 853  953  173  373  673  773  283  383  683  883  983  317  617  137  337
 937  347  547  647  947  167  367  467  967  197  397  797  997
```

Section	Numéro d'anonymat

Vous pourrez utiliser par la suite le prédicat nommé `premier?` qui possède la spécification suivante :

;;; *premier?: nat -> bool*
;;; *(premier? n) reconnaît si n est un nombre premier.*

Question 2.1 – Écrire une fonction sans argument, nommée `NPRAG-chiffres`, qui rend la liste des NPRAG inférieurs à 10.

Réponse [3/50]

Voici deux solutions :

```
;;; NPRAG-chiffres: -> LISTE[nat]
;;; (NPRAG-chiffres) rend la liste des NPRAG inférieur à 10.
(define (NPRAG-chiffres1)
  ;; De fait, ce sont les chiffres premiers et c'est simple à faire de tête:
  (list 1 2 3 5 7) )

(define (NPRAG-chiffres2)
  ;; Énumérer tous les chiffres entre debut (inclus) et fin (exclus):
  (define (iota debut fin)
    (if (< debut fin)
        (cons debut (iota (+ debut 1) fin))
        '() ) )
  ;; et ne retenir que ceux qui sont premiers:
  (cons 1 (filtre premier? (iota 2 10))) )
```

Question 2.2 – Écrire une fonction, nommée `majorant`, qui prend un entier naturel n non nul et rend l'entier 10^k tel que $10^{k-1} \leq n < 10^k$. Ainsi

`(majorant 23)` $\rightarrow$ `100`

Réponse [4/50]

```
;;; majorant: nat -> nat
;;; (majorant p) calcule la puissance de 10 immédiatement supérieure à p.
(define (majorant p)
  (define (compare maj)
    (if (< p maj)
        maj
        (compare (* 10 maj)) ) )
  (compare 10) )
```

Question 2.3 – Écrire une fonction, nommée `prefixe-nombre`, qui prend un chiffre c différent de zéro et un entier naturel n. En supposant que l'entier naturel n s'écrive $\overline{a_i a_{i-1} \ldots a_1 a_0}$, la fonction `prefixe-nombre` rend le nombre s'écrivant $\overline{c a_i a_{i-1} \ldots a_1 a_0}$. Ainsi,

`(prefixe-nombre 1 23)` $\rightarrow$ `123`

Réponse [3/50]

```
;;; prefixe-nombre: nat * nat -> nat
;;; (prefixe-nombre c p) rend le nombre dont l'écriture en base décimale débute
;;; par le chiffre c et s'achève par ceux de p.
(define (prefixe-nombre c p)
  (+ p (* c (majorant p))) )
```

Section Numéro d'anonymat

Question 2.4 – Écrire, à l'aide de la fonction `prefixe-nombre`, une fonction, nommée `NPRAG-suivants`, qui prend un NPRAG n et qui rend tous les NPRAG que l'on peut obtenir en préfixant n par un chiffre quelconque. Ainsi

```
(NPRAG-suivants 1) → (11 31 41 61 71)
(NPRAG-suivants 43) → (143 443 643 743)
```

Réponse [4/50]

Voici deux solutions :

```
;;; NPRAG-suivants: NPRAG -> LISTE[NPRAG]
;;; (NPRAG-suivants p) rend la liste des NPRAG obtenus en préfixant le NPRAG
;;; p par un chiffre.
(define (NPRAG-suivants1 p)
  (define (itere c)
    (if (< c 10)
        (let ((cp (prefixe-nombre c p)))
          (if (premier? cp)
              (cons cp (itere (+ c 1)))
              (itere (+ c 1)) ) )
        '() ) )
  (itere 1) )

(define (NPRAG-suivants2 p)
  (define (prefixe c)
    (prefixe-nombre c p) )
  (filtre premier?
          (map prefixe '(1 2 3 4 5 6 7 8 9)) ) )
```

Question 2.5 – Examiner la fonction suivante :

```
(define (deploiement deployeur lp)
  (if (pair? lp)
      (let* ((p  (car lp))
             (ps (deployeur p)) )
        (cons p (deploiement deployeur (append (cdr lp) ps))) )
      '() ) )
```

Que vaut le programme suivant ?

```
(define (multiple-si-petit n)
  (if (< n 10)
      (list (* 2 n) (* 3 n))
      '() ) )
(deploiement multiple-si-petit '(2 5))
```

Écrire une spécification pour la fonction `deploiement`.

Réponse [6/50]

L'expression a pour valeur :

```
(2 5 4 6 10 15 8 12 12 18 16 24)
```

Voici une spécification :

```
;;; deploiement: (alpha -> LISTE[alpha]) * LISTE[alpha] -> LISTE[alpha]
;;; (deploiement deployeur lp) applique la fonction deployeur à tous les
;;; éléments de la liste lp ainsi qu'aux résultats obtenus.
```

Section Numéro d'anonymat

Question 2.6 – Écrire une fonction, nommée `NPRAG-tous`, qui rend la liste de tous les NPRAG (il y en a un nombre fini). On pourra utiliser (sans la définir) la fonction précédente. L'ordre des NPRAG n'est pas important.

Réponse [5/50]

```
;;; NPRAG-tous: -> LISTE[nat]
;;; (NPRAG-tous) rend la liste de tous les NPRAG.
(define (NPRAG-tous)
  (deploiement NPRAG-suivants (NPRAG-chiffres)) )
```

Question 2.7 – Un NPRAG est extrémal lorsqu'il ne peut être préfixé par un chiffre et ainsi former un nouveau NPRAG. Écrire une fonction, nommée `NPRAG-extremaux`, qui rend la liste de tous les NPRAG extrémaux. Cette fonction pourra utiliser (sans la définir) la fonction précédente. L'ordre des NPRAG n'est pas important.

Réponse [5/50]

```
;;; NPRAG-extremaux: -> LISTE[nat]
;;; (NPRAG-extremaux) rend la liste de tous les NPRAG extrémaux cad les
;;; NPRAG que l'on ne peut préfixer par un chiffre pour former un nouvel NPRAG.
(define (NPRAG-extremaux)
  (define (extremal? p)
    (null? (NPRAG-suivants p)) )
  (filtre extremal? (NPRAG-tous)) )
```

