Examen – LI101 – janvier 2005

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 12 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 60) 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

Ce problème s'intéresse à des listes de nombres.

Question 1.1 – Écrire la signature et la définition d'un prédicat, nommé `decroissante?`, prenant une liste de nombres et vérifiant si ces nombres décroissent (non nécessairement strictement).

```
(decroissante? '(10 5 1 -2 -4))  → #T
(decroissante? '(10 10 9))  → #T
(decroissante? '(10 8 9))  → #F
(decroissante? '())  → #T
```

Réponse [4/60]

En voici une première version :

```
(define (decroissante-2? lx)
  ;; decr?: Nombre * LISTE[Nombre]/non vide/ -> bool
  (define (non-vide-decr? lx)
    (if (pair? (cdr lx))
        (and (>= (car lx) (cadr lx))
             (non-vide-decr? (cdr lx)) )
        #t ) )
  (if (pair? lx)
      (non-vide-decr? lx)
      #t ) )
```

et une seconde avec accumulateur (la variable `dernier`) :

```
;;; decroissante?: LISTE[Nombre] -> bool
;;; (decroissante? lx) vérifie que lx est une liste de nombres décroissants.
(define (decroissante? lx)
  ;; decr?: Nombre * LISTE[Nombre] -> bool
  (define (decr? dernier lx)
    (if (pair? lx)
        (and (>= dernier (car lx))
             (decr? (car lx) (cdr lx)) )
        #t ) )
  (if (pair? lx)
      (decr? (car lx) (cdr lx))
      #t ) )
```

À propos, il est inefficace d'écrire `(> (length L) 0)` à la place de `(pair? L)` !

Question 1.2 – Écrire la signature et la définition d'une fonction, nommée `liste-differences`, prenant une liste d'au moins deux nombres et calculant la liste de leurs différences. Si la liste de nombres est notée $(u_0\ u_1\ \ldots\ u_{n-1})$ alors la liste des différences est la liste $(u_0 - u_1\ \ u_1 - u_2\ \ldots\ u_{n-2} - u_{n-1})$.

```
(liste-differences '(10 5 1 -2 -4))  → (5 4 3 2)
```

Réponse [4/60]

Voici deux solutions :

```
;;; liste-differences: LISTE[Nombre]/au moins 2 termes/ -> LISTE[Nombre]
;;; (liste-differences lx) prend la liste lx (e1 e2 e3 ...) et calcule
;;; la liste des différences ( e1-e2 e2-e3 e3-e4 ...).
(define (liste-differences lx)
  (if (pair? (cdr lx))
      (cons (- (car lx) (cadr lx))
            (liste-differences (cdr lx)) )
      '() ) )

(define (liste-differences-2 lx)
  (let ((difference (- (car lx) (cadr lx))))
    (if (pair? (cddr lx))
        (cons difference (liste-differences-2 (cdr lx)))
        (list difference) ) ) )
```

Question 1.3 – On dira qu'une liste de nombres rétrécit si elle est formée de nombres dont les différences diminuent (au sens large) en valeur absolue. Écrire la signature et la définition d'un prédicat, nommé `retrecissante?`, prenant une liste de nombres contenant au moins deux termes et vérifiant que cette liste rétrécit.

```
(retrecissante? '(1 2 3 4))  → #T
(retrecissante? '(10 5 1 -2 -4))  → #T
(retrecissante? '(10 12 9))  → #F
(retrecissante? '(10 11))  → #T
```

Section | Numéro d'anonymat

Réponse [3/60]

Voici deux solutions dont l'une utilise les fonctions précédentes : on calcule la liste des différences et on regarde si elle décroît.

```
;;; retrecissante?: LISTE[Nombre]/au moins 2 termes/ -> bool
;;; (retrecissante? lx) vérifie que la différence (en valeur absolue) entre deux
;;; termes consécutifs de lx diminue strictement.
(define (retrecissante? lx)
  ;; diminue?: Nombre * LISTE[Nombre] -> bool
  (define (diminue? delta lx)
    (if (pair? lx)
        (if (pair? (cdr lx))
            (let ((d (abs (- (car lx) (cadr lx)))))
              (if (< d delta)
                  (diminue? d (cdr lx))
                  #f ) )
            #f )
        #t ) )
  (diminue? (abs (- (car lx) (cadr lx))) (cdr lx)) )
(define (retrecissante? lx)
  (decroissante? (map abs (liste-differences lx))) )
```

Exercice 2

Ce problème s'intéresse aussi à des listes de nombres.

Question 2.1 – Écrire une définition du prédicat, nommé `positive-negative?`, prenant une liste de nombres et vérifiant que cette liste est formée d'un nombre positif ou nul suivi d'un nombre strictement négatif suivi d'un nombre positif ou nul et ainsi de suite. Ainsi

```
(positive-negative? '())  → #T
(positive-negative? '(1))  → #T
(positive-negative? '(1 -2))  → #T
(positive-negative? '(1 2 3))  → #F
(positive-negative? '(1 -2 3))  → #T
(positive-negative? '(-1 2 -3))  → #F
(positive-negative? '(1 -2 3 -4 5))  → #T
```

Section Numéro d'anonymat

Réponse [4/60]

Voici une solution reposant sur un couple de fonctions mutuellement récursives :

```
;;; positive-negative?: LISTE[Nombre] -> bool
;;; (positive-negative? lx) vérifie que lx est une liste de nombres
;;; alternativement positif puis négatif.
(define (positive-negative? lx)
  ;; pn?: LISTE[Nombre] -> bool
  ;; (pn? lx) vérifie que lx est une liste de nombres alternativement
  ;; positif puis négatif.
  (define (pn? lx)
    (if (pair? lx)
        (and (> (car lx) 0)
             (np? (cdr lx)) )
        #t ) )
  ;; np?: LISTE[Nombre] -> bool
  ;; (np? lx) vérifie que lx est une liste de nombres alternativement
  ;; négatif puis positif.
  (define (np? lx)
    (if (pair? lx)
        (and (< (car lx) 0)
             (pn? (cdr lx)) )
        #t ) )
  (pn? lx) )
```

Et voici une autre solution plus classique :

```
(define (positive-negative-4? lx)
  (if (pair? lx)
      (and (>= (car lx) 0)
           (if (pair? (cdr lx))
               (and (< (cadr lx) 0)
                    (positive-negative-4? (cddr lx)) )
               #t ) )
      #t ) )
```

Question 2.2 – Écrire une définition du prédicat, nommé `alternee?`, prenant un nombre (que l'on qualifiera de *pivot*) et une liste de nombres et vérifiant que cette liste est formée de nombres alternativement égaux ou plus grands puis strictement plus petits que ce pivot. Ainsi

```
(alternee? 5 '(15 2 16 3 17))  → #T
(alternee? 5 '(1 15 2 16 3 17))  → #F
```

Section Numéro d'anonymat

Réponse [4/60]

Voici les deux mêmes types de solutions à base de récursion mutuelle ou bien en style direct :

```
;;; alternee?: Nombre * LISTE[Nombre] -> bool
;;; (alternee? pivot lx) vérifie que lx est une liste de nombres alternativement
;;; plus grand puis plus petit que pivot.
(define (alternee? pivot lx)
  ;; au-dessus?: LISTE[Nombre] -> bool
  ;; (au-dessus? lx) vérifie que lx est une liste de nombres alternativement
  ;; plus grand puis plus petit que pivot.
  (define (au-dessus? lx)
    (if (pair? lx)
        (and (>= (car lx) pivot)
             (au-dessous? (cdr lx)) )
        #t ) )
  ;; au-dessous?: LISTE[Nombre] -> bool
  ;; (au-dessous? lx) vérifie que lx est une liste de nombres alternativement
  ;; plus petit puis plus grand que pivot.
  (define (au-dessous? lx)
    (if (pair? lx)
        (and (< (car lx) pivot)
             (au-dessus? (cdr lx)) )
        #t ) )
  (au-dessus? lx) )

(define (alternee-2? pivot lx)
  (if (pair? lx)
      (and (>= (car lx) pivot)
           (if (pair? (cdr lx))
               (and (< (cadr lx) pivot)
                    (alternee-2? pivot (cddr lx)) )
               #t ) )
      #t ) )
```

On peut également utiliser `positive-negative?` à condition de translater les nombres.

```
(define (alternee-5? pivot lx)
  (define (translate x)
    (- x pivot) )
  (positive-negative? (map translate lx)) )
```

Question 2.3 – Écrire, à l'aide du prédicat `alternee?`, une nouvelle définition du prédicat `positive-negative?`.

Réponse [2/60]

```
(define (positive-negative-2? lx)
  (alternee? 0 lx) )
```

Question 2.4 – Écrire une spécification et une définition de la fonctionnelle, nommée `alternativement?`, ayant la signature suivante.

`;;;` *alternativement?: (alpha -> bool) * LISTE[alpha] -> bool*

Cette fonctionnelle généralise les prédicats précédents. Cette fonctionnelle prend en arguments un prédicat et une liste. Elle vérifie que la liste comporte un terme satisfaisant le prédicat puis un terme ne le satisfaisant pas et ainsi de suite.

```
(alternativement? odd? '(1 2 3 4 5))  → #T
(alternativement? string? '("1" 2 "33" #T))  → #T
(alternativement? string? '("a" "bb"))  → #F
```

Section Numéro d'anonymat

Réponse [5/60]

```
;;; alternativement?: (alpha -> bool) * LISTE[alpha] -> bool
;;; (alternativement? p lx) vérifie que les termes de rang impair (le premier,
;;; le troisième, etc.) de lx satisfont p et que les termes de rang pair (le
;;; second, le quatrième, etc.) ne satisfont pas p.
(define (alternativement? p lx)
  (define (oui? lx)
    (if (pair? lx)
        (and (p (car lx))
             (non? (cdr lx)) )
        #t ) )
  (define (non? lx)
    (if (pair? lx)
        (and (not (p (car lx)))
             (oui? (cdr lx)) )
        #t ) )
  (oui? lx) )
```

À propos, il est inutile de remplacer `(p (car L))` par `(equal? #t (p (car L)))` ou même par `(equal? (car (map p L)) #t)`!

Question 2.5 – Écrire, à l'aide de la fonctionnelle `alternativement?`, une nouvelle définition du prédicat `positive-negative?`.

Réponse [2/60]

```
(define (positive-negative-3? lx)
  (define (positif-ou-nul? x)
    (>= x 0) )
  (alternativement? positif-ou-nul? lx) )
```

Question 2.6 – Écrire, à l'aide de la fonctionnelle `alternativement?`, une nouvelle définition du prédicat `alternee?`.

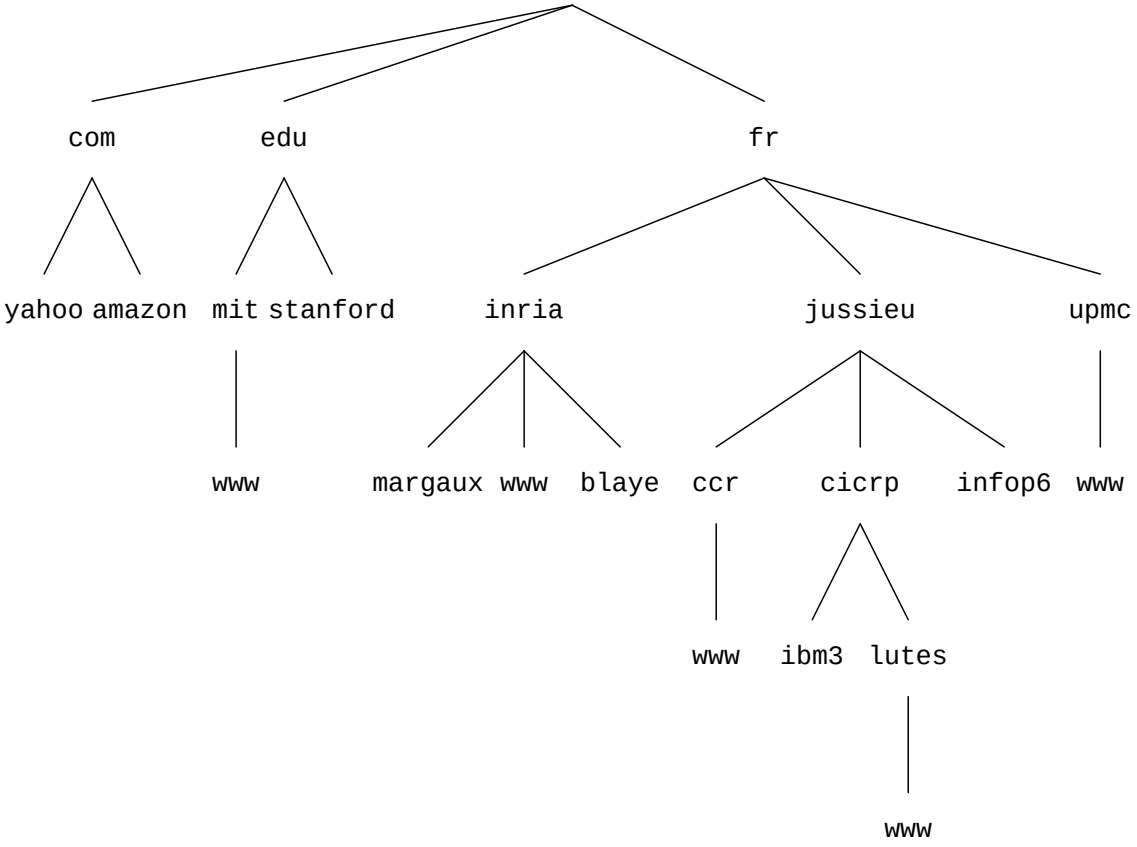
Réponse [5/60]

```
(define (alternee-4? pivot lx)
  (define (plus-grand x)
    (>= x pivot) )
  (alternativement? plus-grand lx) )
```

Exercice 3

Cet exercice travaille autour de l'arbre des noms de domaines et de machines d'Internet. Cet arbre que l'on nommera un « arbre Internet » sera représenté par un arbre général dont les étiquettes sont des chaînes de caractères, la racine a une étiquette qui est la chaîne vide. Voici l'arbre qui sera utilisé dans les exemples qui suivent et qui est produit par l'expression `(ai)`. Dans ce dessin, les étiquettes sont imprimées sans guillemets.

Section Numéro d'anonymat



L'arbre Internet a des nœuds qui peuvent être des noms de domaines ou des noms de machines. Un nom de domaine ou de machine comme `ccr.jussieu.fr` est représenté par le nœud obtenu en partant de la racine de l'arbre Internet et en suivant les arêtes menant à `fr`, `jussieu` puis `ccr`. Ce chemin suivi dans l'arbre Internet sera représenté par la liste `("fr" "jussieu" "ccr")`. Un nœud est un nom de domaine si sa forêt n'est pas vide (par exemple `com` ou `jussieu.fr`), un nœud est une machine si sa forêt est vide (par exemple, `www.ccr.jussieu.fr` ou `yahoo.com`).

On pourra, sans le redéfinir, utiliser le prédicat `ag-feuille?` vu en TD pour toutes les questions qui suivent.

Question 3.1 – Écrire la signature et une définition de la fonction, nommée `nombre-de-fils-sous-racine`, qui prend un arbre Internet et calcule le nombre de sous-arbres issus de la racine. Ainsi

```
(nombre-de-fils-sous-racine (ai))  → 3
```

Réponse [3/60]

```
;;; nombre-de-fils-sous-racine: ArbreGeneral[string] -> nat
;;; (nombre-de-fils-sous-racine ag) calcule le nombre de fils issues
;;; de la racine de l'arbre ag.
(define (nombre-de-fils-sous-racine ag)
  (length (ag-foret ag)) )
```

Question 3.2 – Écrire la signature et une définition de la fonction, nommée `nombre-total-de-fils`, qui prend un arbre Internet et calcule le nombre de descendants de la racine présents dans cet arbre (c'est-à-dire le nombre total de nœuds total de l'arbre diminué de 1) Ainsi

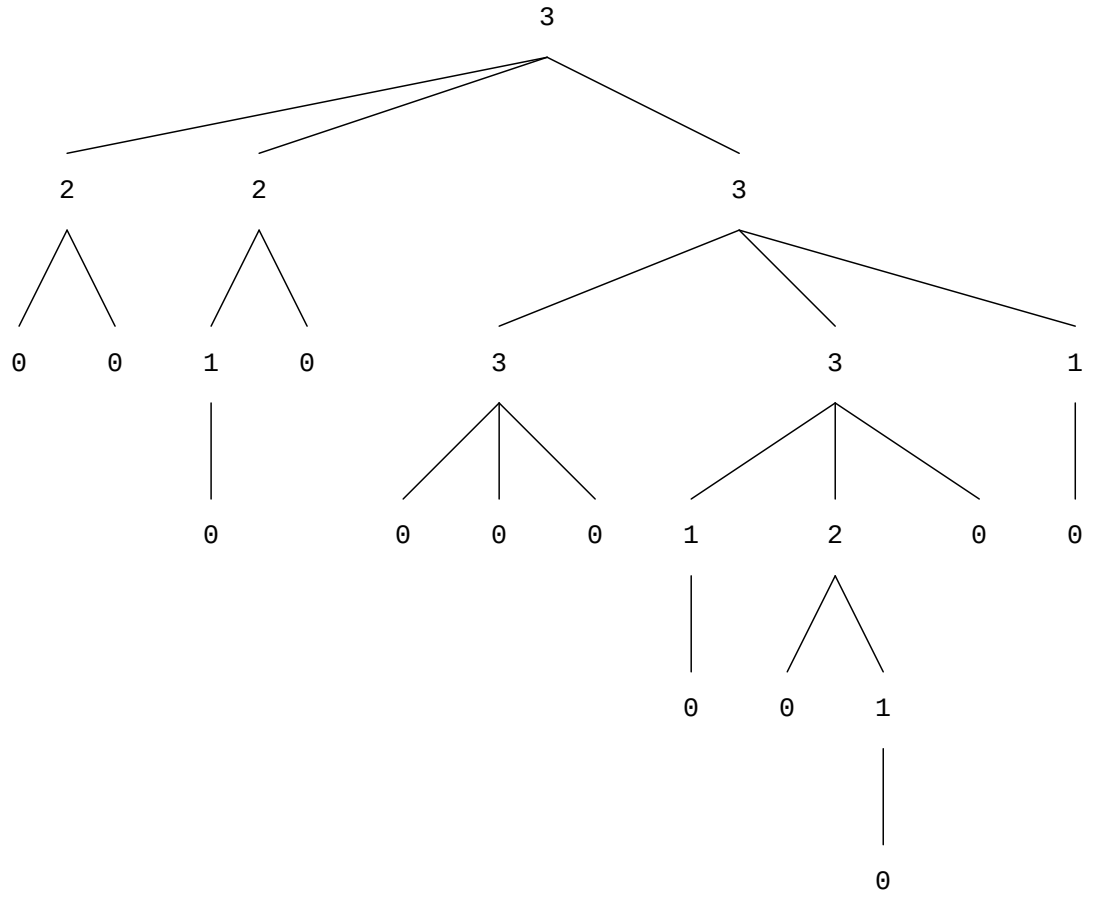
```
(nombre-total-de-fils (ai))  → 22
```

Section Numéro d'anonymat

Réponse [3/60]

```
;;; nombre-total-de-fils: ArbreGeneral[string] -> nat
;;; (nombre-total-de-fils ag) calcule le nombre de branches présentes
;;; dans l'arbre ag.
(define (nombre-total-de-fils ag)
  (reduce +
          (nombre-de-fils-sous-racine ag)
          (map nombre-total-de-fils (ag-foret ag)) ) )
```

Question 3.3 – Écrire la signature et une définition de la fonction, nommée `arbre-des-nombres-de-fils`, transformant tout nœud de l'arbre Internet reçu en argument en un nouveau nœud dont l'étiquette est le nombre de fils qu'il possède. Ainsi l'application de cette fonction sur l'arbre produit par l'expression `(ai)` donne l'arbre suivant :

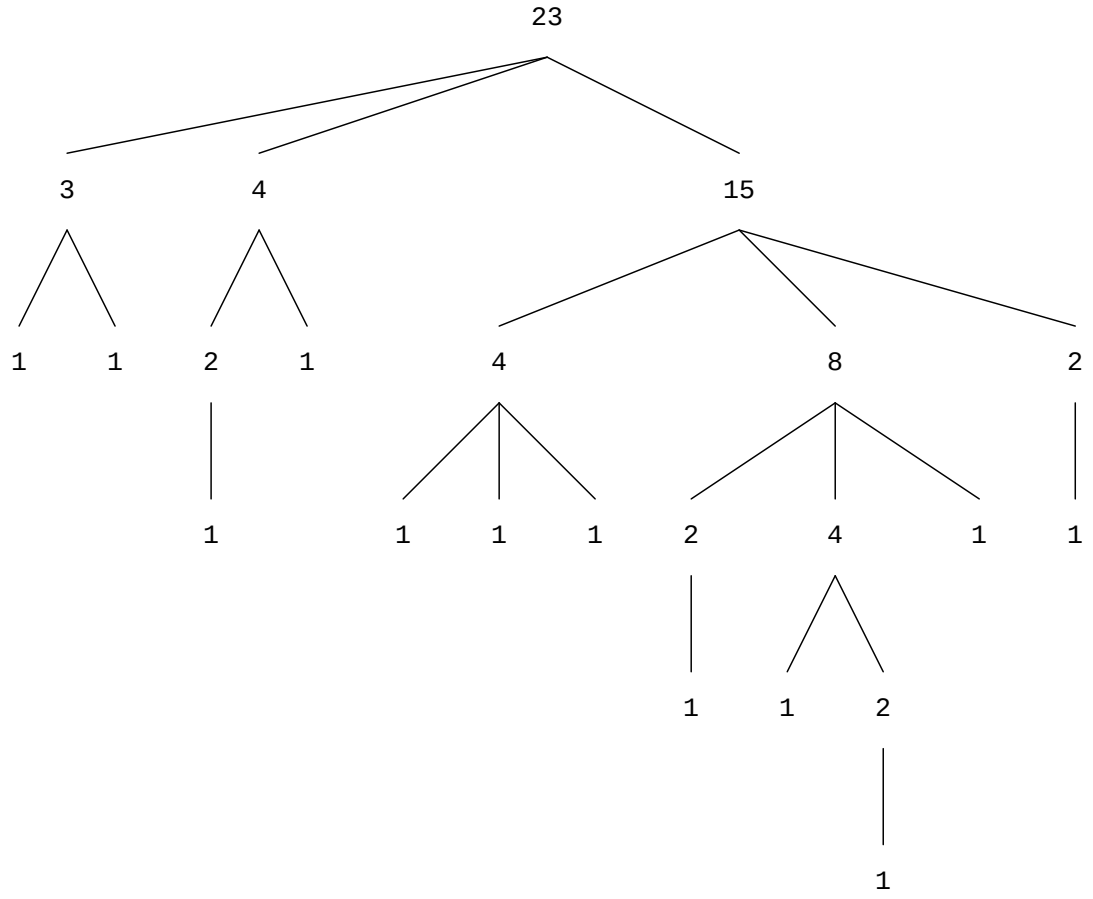


Réponse [4/60]

```
;;; arbre-des-nombres-de-fils: ArbreGeneral[string] -> ArbreGeneral[nat]
;;; (arbre-des-nombres-de-fils ag) crée un arbre de même structure qu'ag dont
;;; chaque nœud a pour étiquette son nombre de fils.
(define (arbre-des-nombres-de-fils ag)
  (ag-noeud (length (ag-foret ag))
            (map arbre-des-nombres-de-fils (ag-foret ag)) ) )
```

Question 3.4 – Écrire une fonction, nommée `arbre-des-nombres-de-descendants`, transformant tout nœud de l'arbre reçu en argument en un nouveau nœud dont l'étiquette est le nombre de descendants (y compris lui-même) qu'il possède. Ainsi l'application de cette fonction sur l'arbre produit par l'expression `(ai)` donne l'arbre suivant :

Section | Numéro d'anonymat



Réponse [4/60]

```
;;; arbre-des-nombres-de-descendants: ArbreGeneral[string] -> ArbreGeneral[nat]
;;; (arbre-des-nombres-de-descendants ag) crée un arbre de même structure qu'ag
;;; dont chaque nœud a pour étiquette son nombre total de descendants.
(define (arbre-des-nombres-de-descendants ag)
  (let ((foret (map arbre-des-nombres-de-descendants (ag-foret ag))))
    (ag-noeud (reduce + 1 (map ag-etiquette foret))
              foret ) ) )
```

Question 3.5 – Écrire la signature et une définition d'un prédicat, nommé `machine-existe?`, prenant un arbre Internet et une liste de chaînes de caractères représentant le nom d'une machine et vérifiant si cette machine est l'une des machines de cet arbre Internet. Le nom d'une machine, disons `www.upmc.fr`, sera représenté par une liste `("fr" "upmc" "www")`. Ainsi

```
(machine-existe? (ai) '("fr" "upmc" "www"))  → #T
(machine-existe? (ai) '("fr" "www"))  → #F
(machine-existe? (ai) '("upmc" "www"))  → #F
```

Section Numéro d'anonymat

Réponse [4/60]

```
;;; machine-existe?: ArbreInternet * LISTE[string] -> bool
;;; (machine-existe? ai ls) vérifie si ls est un chemin menant à une feuille dans
;;; l'arbre ai.
(define (machine-existe? ai ls)
  ;; existe-foret?: string * LISTE[ArbreInternet] -> ArbreInternet
  ;; (existe-foret? nom foret) rend l'arbre de la foret dont l'étiquette est
  ;; égale à nom. Rend #f s'il n'y a pas de tel arbre.
  (define (existe-foret? nom foret)
    (if (pair? foret)
        (if (equal? nom (ag-etiquette (car foret)))
            (car foret)
            (existe-foret? nom (cdr foret)) )
        #f ) )
  (if (pair? ls)
      (let ((branche (existe-foret? (car ls) (ag-foret ai))))
        (if branche
            (machine-existe? branche (cdr ls))
            #f ) )
      (not (pair? (ag-foret ai))) ) )
```

Voici une autre solution sous la forme de deux fonctions indépendantes :

```
;;; Autre solution sous forme de deux fonctions indépendantes:
(define (machine-existe-2? arbre nom)
  (if (pair? nom)
      (recherche? (ag-foret arbre) nom)
      (not (pair? (ag-foret arbre)))))
(define (recherche? foret nom)
  (if (pair? foret)
      (if (equal? (car nom) (ag-etiquette (car foret)))
          (machine-existe-2? (car foret) (cdr nom))
          (recherche? (cdr foret) nom))
      #f))
```

Question 3.6 – Écrire la signature et une définition d'une fonction, nommée `machines`, prenant un arbre Internet et calculant la liste des noms de toutes les machines présentes dans cet arbre.

```
(car (machines (ai)))  → "yahoo.com"
(cadr (machines (ai)))  → "amazon.com"
(caddr (machines (ai)))  → "www.mit.edu"
```

Réponse [4/60]

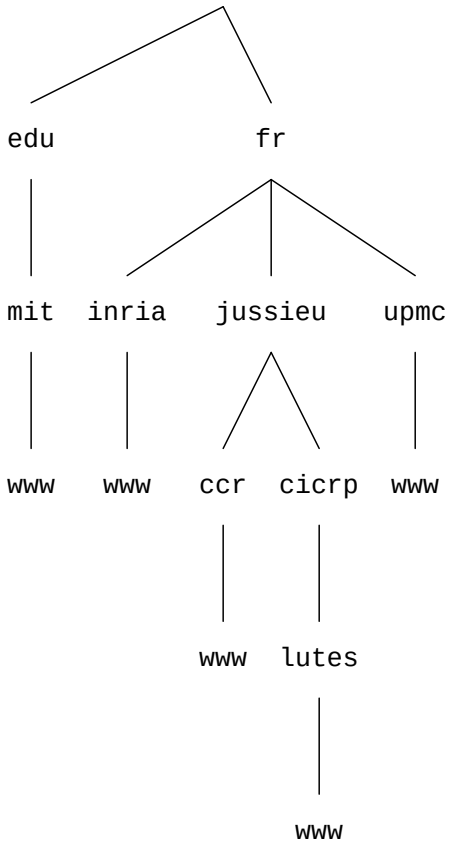
```
;;; machines: ArbreInternet -> LISTE[string]
;;; (machines ai) calcule la liste des noms de machines présentes dans l'arbre ai.
(define (machines ai)
  ;; sous-machines: ArbreInternet -> LISTE[string]
  ;; (sous-machines sai) rend la liste des noms de machines présentes dans le
  ;; sous-arbre sai.
  (define (sous-machines sai)
    (define (suffixe nom)
      (if (equal? (ag-etiquette ai) "")
          nom
          (string-append nom "." (ag-etiquette ai)) ) )
    (map suffixe (machines sai)) )
  (if (pair? (ag-foret ai))
      (reduce append (list) (map sous-machines (ag-foret ai)))
      (list (ag-etiquette ai)) ) )
```

Question 3.7 – Les serveurs Web sont des machines dont, la plupart du temps, le nom débute par `www`. On souhaite déterminer les serveurs Web existant dans un arbre Internet. On souhaite donc élaguer l'arbre Internet de toute branche qui ne mène pas vers une feuille étiquetée `www`. Écrire la fonction, nommée `www-seules`, qui prend un arbre internet

et calcule un tel nouvel arbre internet (on retournera Faux si l'arbre internet ne contient aucun serveur Web). Voici sa spécification :

```
;;; www-seules: ArbreInternet -> ArbreInternet + #f
;;; (www-seules ai) calcule un nouvel arbre Internet ne comportant que des nœuds
;;; d'étiquette "www" comme feuilles.
```

Ainsi l'application de cette fonction sur l'arbre produit par l'expression `(ai)` donne l'arbre suivant



Section | Numéro d'anonymat

Réponse [5/60]

```
;;; www-seules: ArbreInternet -> ArbreInternet + #f
;;; (www-seules ai) calcule un nouvel arbre Internet ne comportant que des nœuds
;;; d'étiquette "www" comme feuilles.
(define (www-seules ai)
  (define (elaguer-foret foret)
    (if (pair? foret)
        (let ((nouveau (www-seules (car foret))))
          (if nouveau
              (cons nouveau (elaguer-foret (cdr foret)))
              (elaguer-foret (cdr foret)) ) )
        (list) ) )
  (if (pair? (ag-foret ai))
      (let ((foret (elaguer-foret (ag-foret ai))))
        (if (pair? foret)
            (ag-noeud (ag-etiquette ai) foret)
            #f ) )
      (if (equal? (ag-etiquette ai) "www")
          ai
          #f ) ) )
```

