Université Pierre et Marie Curie
DEUG MIAS première année
Informatique 1

Année 2001-2002
Examen septembre
2 septembre 2002
2 heures

Remarques :

- *Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.*
- *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.*
- *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 si elle est donnée dans la question).*
- *Le barême (total sur 45) n'est donné qu'à titre indicatif.*

Listes triées

Dans cet exercice, on considère des listes triées (croissantes) de nombres. Ces listes peuvent contenir plusieurs occurrences d'un même nombre.

Question 1 (3 points) :

Écrire une définition de la fonction `recherche`, qui a comme spécification :
`;;;` *recherche : Nombre * LISTE[Nombre] -> bool*
`;;;` *Hypothèse : la liste est triée croissante*
`;;;` *(recherche x L) renvoie Vrai ssi le nombre x apparait dans la liste triée L*

Question 2 (4 points) :

Écrire la spécification et une définition de la fonction `ajout`, qui, étant donnés un nombre `x` et une liste triée de nombres `L` , renvoie la liste triée résultant de l'ajout du nombre `x` dans la liste `L` (si `x` est déjà présent dans `L`, on l'ajoutera aussi).

Question 3 (4 points) :

Écrire la spécification et une définition de la fonction `suppressionUn`, qui, étant donnés un nombre `x` et une liste triée de nombres `L` , renvoie la liste triée résultant de la suppression d'une occurrence du nombre `x` dans la liste `L` (si `x` n'est pas dans `L` , renvoie la liste `L`). Par exemple
`(suppressionUn 3 '(1 2 3 3 5 10))` renvoie la liste `(1 2 3 5 10)`,
`(suppressionUn 4 '(1 2 3 3 5 10))` renvoie la liste `(1 2 3 3 5 10)`.

Question 4 (5 points) :

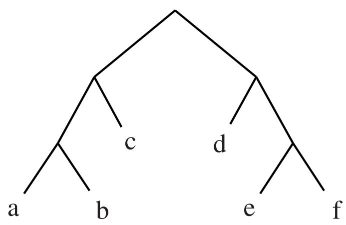
Écrire la spécification et une définition de la fonction `suppressionTous`, qui, étant donnés un nombre `x` et une liste triée de nombres `L`, renvoie la liste résultant de la suppression de toutes les occurrences du nombre `x` dans la liste `L` (si `x` n'est pas dans `L` , renvoie la liste `L`). Par exemple
`(suppressionTous 3 '(1 2 3 3 5 10))` renvoie la liste `(1 2 5 10)`.

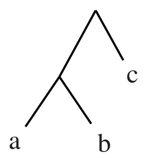
Question 5 (6 points) :

Écrire la spécification et une définition de la fonction `suppressionEntre`, qui, étant donnés deux nombres `x1`, `x2` et une liste triée de nombres `L`, renvoie la liste résultant de la suppression de toutes les occurrences de tous les nombres compris entre `x1` et `x2` (inclus) dans la liste `L`. Par exemple
`(suppressionEntre 3 5 '(1 2 3 3 4 5 10))` renvoie la liste `(1 2 10)`.

Arbres binaires étiquetés aux feuilles

Dans ce problème, nous nous intéressons aux arbres étiquetés aux feuilles. Voici deux dessins de tels arbres :





Dans ces arbres, tout nœud est une feuille ou a exactement deux sous-arbres immédiats. C'est uniquement aux feuilles que sont attachées des étiquettes.

Pour manipuler ces arbres, dont le type est `AB[`α`]`, vous devez utiliser la barrière d'abstraction suivante :

;;;; *Reconnaisseur*
;;; *AB-feuille? : AB[α] -> bool*
;;; *(AB-feuille? A) rend #t ssi A est réduit à une feuille*

;;;; *Accesseurs*
;;; *AB-etiquette : AB[α] -> α*
;;; *HYPOTHESE : (AB-feuille? A) est vrai*
;;; *(AB-etiquette A) rend l'étiquette attachée à la feuille A*

;;; *AB-gauche : AB[α] -> α*
;;; *HYPOTHESE : (AB-feuille? A) est faux*
;;; *(AB-gauche A) rend le sous-arbre gauche de A*

;;; *AB-droit : AB[α] -> α*
;;; *HYPOTHESE : (AB-feuille? A) est faux*
;;; *(AB-droit A) rend le sous-arbre droit de A*

;;;; *Constructeurs*
;;; *AB-feuille : α -> AB[α]*
;;; *(AB-feuille x) rend l'arbre réduit à une feuille étiquetée par x*

;;; *AB-arbre : AB[α] * AB[α] -> AB[α]*
;;; *(AB-arbre A1 A2) rend l'arbre ayant A1 comme sous-arbre gauche et A2 comme sous-arbre droit*

Question 6 (2 points) :

Voici deux fonctions, utilisant la barrière d'abstraction précédente, qui renvoient des arbres étiquetés aux feuilles.

;;; *Exemple1 : -> AB[α]*
;;; *(Exemple1) rend l'arbre EX1*

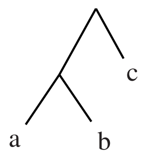
```
(define (Exemple1)
  (let ((F1 (AB-feuille 'd))
        (F2 (AB-feuille 'e))
        (F3 (AB-feuille 'f)))
    (AB-arbre  F3 (AB-arbre F1 F2))))
```

;;; *Exemple2 : -> AB[α]*
;;; *(Exemple2) rend l'arbre EX2*

```
(define (Exemple2)
  (let* ((F1 (AB-feuille 'p))
         (F2 (AB-feuille 'q))
         (F3 (AB-feuille 'r))
         (AA (AB-arbre  F1 (AB-arbre F2 F3))))
    (AB-arbre  (Exemple1) AA)))
```

Dessiner graphiquement les arbres EX1 et EX2 construits par ces deux fonctions.

Question 7 (3 points) :
Donner une définition de la fonction `Exemple3` qui renvoie l'arbre EX3 suivant :



Question 8 (5 points) :
Rappel : on définit récursivement la profondeur d'un arbre comme suit :
- si l'arbre est réduit à une feuille, sa profondeur est 1,
- sinon sa profondeur est égale au maximum des profondeurs de ses sous-arbres immédiats, augmenté de 1.

Par exemple, la profondeur de l'arbre construit par `Exemple3` est 3.

Écrire la spécification et une définition de la fonction `profondeur` qui, étant donné un arbre, renvoie sa profondeur.

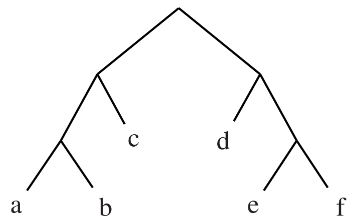
Question 9 (6 points) :

Écrire la spécification et une définition de la fonction `liste-etiquettes`, qui, étant donné un arbre binaire étiqueté aux feuilles `A`, renvoie la liste, de gauche à droite, des étiquettes des feuilles de `A`. Par exemple `(liste-etiquettes(Exemple3))` rend la liste `(a b c)`.

Étant donné un arbre binaire étiqueté aux feuilles et une liste de booléens, on peut parcourir l'arbre en considérant que la liste de booléens représente le chemin à suivre dans l'arbre à partir de la racine, en allant dans le sous-arbre droit (resp. gauche) lorsque le premier élément de la liste est égal à `#t` (resp. `#f`), en continuant dans le sous-arbre droit (resp. gauche) lorsque le deuxième élément de la liste est égal à `#t` (resp. `#f`)... et en s'arrêtant lorsque la liste est vide. On dit alors que la liste désigne le nœud ainsi atteint.

Ce parcours n'est pas défini lorsque l'on arrive sur une feuille et que la liste n'est pas vide, et il n'est pas non plus défini lorsque la liste est vide et que l'on n'est pas sur une feuille.

Par exemple, sur l'arbre dessiné ci-dessous :



- la liste `'(#t #f)` désigne la feuille étiquetée par `d`,
- la liste `'(#f #f #f)` désigne la feuille étiquetée par `a`,
- la liste `'(#t #f #f)` ne désigne rien,

– la liste `'(#f #f)` ne désigne rien.

Question 10 (2 points) :

Donner les listes qui désignent les feuilles étiquetées par `b`, `c` et `e` dans l'arbre ci-dessus.

Question 11 (5 points) :

Donner la spécification et une définition de la fonction `element` qui, étant donnés un arbre binaire étiqueté aux feuilles `A` et une liste de booléens `L`, rend la valeur faux si la liste ne désigne pas une feuille de l'arbre, et sinon rend l'etiquette de la feuille de `A` désignée par `L`.