Licence STS, semestre 3 — 2007–2008
Algorithmique et approche fonctionnelle (Info 211) — Durée : 3h
3 pages

Examen du 16 Juin 2008

Les notes de cours, de TD et de TP manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse. Le barème est indicatif.

1 Typage (4 points)

Les fonctions suivantes sont-elles bien typées ? Si oui, donner leur type, sinon préciser pourquoi.

```
let f1 x y z = (y x) + z

let f2 a b = if a=b then 1 else 2

let rec f3 x y = if f3 y x then y else x

let rec f4 x = f4 x , f4 x

let f5 x y = if x() then y else x y

let rec f6 x = f6 (x || f6 (x && f6 x))
```

2 Évaluation (4 points)

1. Étant donnée la fonction `mystere: int -> int -> int -> int` suivante :

   ```
   let rec mystere a b c =
     if b = 0 then a - c
     else mystere c (b-1) a
   ```

   Donner le résultat des évaluations de `mystere 0 45377 10` et de `mystere 0 124322 10`.

2. Étant donnée la fonction `mystere: unit list -> unit list -> unit list` suivante :

   ```
   let rec mystere x = function
       [] -> ()::x
     | y::l -> mystere (y::x) l
   ```

   Donner le résultat de l'évaluation de `mystere [()] [();()]`

3. Étant donnée la fonction `mystere: 'a list * 'a list -> 'a list -> 'a list * 'a list` suivante :

   ```
   let rec mystere (g,d) = function
       [] -> (g,d)
     | x::l -> mystere (d,x::g) l
   ```

   Donner le résultat de l'évaluation de `mystere ([],[]) [1;2;3;4;5;6]`

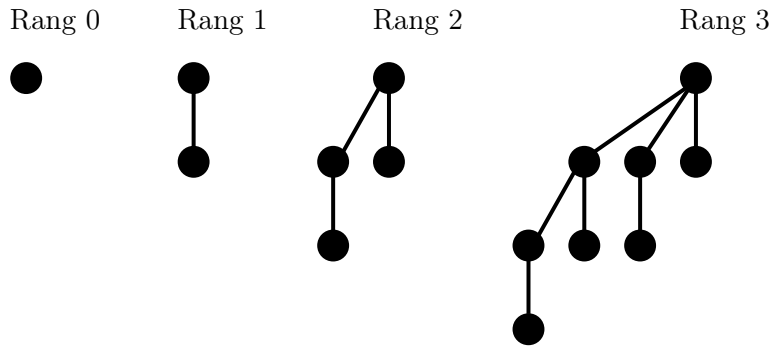
3 Programmation : Files binomiales à priorités (12 points)

Nous nous intéressons dans cette section à l'implémentation d'une structure de données appelée *file binomiale à priorités*. Une file à priorités est une structure de données sur laquelle on peut effectuer les trois opérations suivantes : *ajouter* un élément, *extraire* et *supprimer* l'élément *minimum* de la file.

Les files binomiales à priorités sont implémentées comme des listes d'*arbres binomiaux*. Ces arbres sont définis récursivement de la manière suivante :

- un arbre binomial de *rang* 0 est un arbre composé d'un unique nœud ;
- un arbre binomial de rang $n+1$ est formé en *reliant* deux arbres binomiaux de rang n, l'un des deux arbres devenant le fils le plus à gauche de l'autre arbre.

Une autre manière de définir un arbre binomial est de dire qu'un arbre binomial de rang n est un arbre ayant n fils $t_1, \ldots, t_n$, où chaque t_i est un arbre binomial de rang $n-i$. Selon ces définitions, il est immédiat de voir qu'un arbre binomial de rang n possède *exactement* 2^n nœuds. La figure suivante montre la forme des arbres binomiaux de rangs 0, 1, 2 et 3.



En plus de cette forme particulière, les arbres binomiaux utilisés pour construire des files à priorités auront la propriété que l'élément stocké à la racine de l'arbre est *inférieur ou égal* à tous les éléments de ses sous-arbres, ces sous-arbres ayant également cette propriété.

Nous utiliserons le type OCaml suivant pour représenter les arbres binomiaux.

```
type 'a arbre = { rang : int ; elem : 'a ; fils: 'a arbre list}
```

Les étiquettes `rang`, `elem` et `fils` contiennent respectivement le rang de l'arbre, l'élément associé à la racine de l'arbre, et enfin la liste des fils de la racine qui sera maintenue selon l'ordre *décroissant* des rangs de ces sous-arbres.

Les files binomiales à priorités seront représentées par des listes d'arbres binomiaux ayant tous des rangs différents deux à deux. Ces listes d'arbres seront ordonnées selon l'ordre *croissant* des rangs.

```
type 'a file =  'a arbre list
```

Les questions suivantes permettent de définir progressivement les fonctions nécessaires à l'implémentation des trois opérations (ajout, extraction et suppression) des files binomiales à priorités. Nous supposerons par la suite que les éléments de type `'a` stockés dans les files à priorités peuvent être comparés avec les opérateurs de comparaison polymorphes de OCaml (tels que `<`, `<=` ou `=`).

1. Définir une fonction `relier : 'a arbre -> 'a arbre -> 'a arbre` permettant de relier deux arbres binomiaux de *même* rang n afin de former un arbre binomial de rang $n+1$ (en respectant bien la propriété sur l'ordre des éléments).

2. Définir une fonction `inserer_arbre : 'a arbre -> 'a file -> 'a file` pour insérer un arbre `a` de rang n dans un file dont l'arbre `b` de plus petit rang est supérieur ou égal à n. Il suffira pour cela d'ajouter `a` en tête de liste si n est *strictement* inférieur au rang de `b`, ou d'ajouter récursivement l'arbre obtenu en reliant `a` et `b` dans le reste de la file.
3. Définir la fonction `inserer : 'a -> 'a file -> 'a file` pour insérer un élément x dans une file à priorités en insérant, à l'aide de la fonction précédente, un arbre de rang 0 contenant x dans la file à priorités.
4. Définir une fonction `supprimer_arbre_min : 'a file -> 'a arbre * 'a file` qui supprime l'arbre avec la plus petite racine dans une file à priorités. Cette fonction retourne cet arbre ainsi que la file résultant de cette suppression.
5. À l'aide de la fonction précédente, définir la fonction `minimum : 'a file -> 'a` qui retourne le plus petit élément d'une file à priorités.
6. Définir la fonction `fusion : 'a file -> 'a file -> 'a file` qui permet de fusionner deux files à priorités. Cette fonction (récursive) parcourt les deux files (dans l'ordre croissant des rangs), en reliant les arbres de même rang à l'aide de la fonction `relier` et en insérant (à l'aide de la fonction `inserer_arbre`) l'arbre ainsi obtenu dans le résultat – récursif – de la fusion.
7. En utilisant la fonction précédente ainsi que la fonction `List.rev` (pour renverser un liste), définir la fonction `supprimer_minimum : 'a file -> 'a file` qui supprime le plus petit élément d'un file à priorités.