Licence STS, semestre 3 — 2007–2008
Algorithmique et approche fonctionnelle (Info 211) — Durée : 2h
2 pages

Examen du 19 Décembre 2007

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.

Important : Dans toutes les questions qui suivent, nous vous invitons à utiliser des itérateurs sur les listes chaque fois que cela est possible.

1 Graphisme (6 points)

On se propose dans cette section de définir des types et fonctions graphiques de base afin de manipuler des scènes graphiques. Les objets de ces scènes seront des polygones. Chaque polygone est représenté par la liste de ses sommets (le premier et le dernier élément de la liste étant le même sommet). Un sommet est un point à coordonnées entières dans le plan. Concrètement, ces objets seront représentés par les types OCaml suivants :

```
type point = int * int
type polygone = point list
type scene = polygone list
```

Pour l'affichage des scènes nous utiliserons deux primitives graphiques `moveto : point -> unit` et `lineto : point -> unit`. Un appel `moveto p` positionne le point courant du dispositif graphique sur `p`. Un appel `lineto p` trace une ligne depuis le point courant vers le point `p`, ce dernier devenant le point courant après cet appel.

1. En utilisant les deux primitives graphiques, définir une fonction `afficher_polygone` de type `polygone -> unit` permettant d'afficher un polygone.
2. Définir une fonction `afficher_scene` de type `scene -> unit` pour afficher une scène.
3. Définir une fonction `transformation` de type `(point -> point) -> scene -> scene` telle que `transformation f p` applique la fonction `f` à tous les points des polygones d'une scène.
4. En déduire une fonction `translation` de type `(int * int) -> scene -> scene` qui applique une translation à une scène.
5. En supposant donnée une fonction `dist : point -> point -> int` calculant la distance entre deux points, définir la fonction `perimetre_polygone : polygone -> int` calculant le périmètre d'un polygone (la somme des longueurs de ses cotés).
6. Enfin, définir la fonction `perimetre : scene -> int` calculant la somme des périmètres des polygones d'une scène.

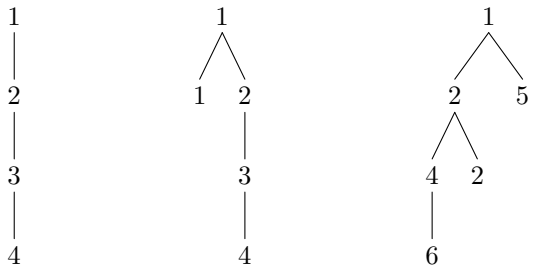

FIG. 1 – exemples de tas

2 Tri pas Tas (14 points)

Nous nous intéressons dans cette section à l'implémentation d'un algorithme de tri efficace pour des listes d'entiers. Cet algorithme reposera sur une structure de données appelée *tas*. Les tas représentent des ensembles (éventuellement avec des répétitions) où l'accès au plus petit élément est efficace.

Un tas peut être représenté par un *arbre binaire* possédant la propriété que l'entier stocké à la racine de l'arbre est *inférieur ou égal* à tous les éléments de ses sous-arbres gauche et droit, ces sous-arbres étant également des tas. Les arbres binaires représentés dans la figure 1 sont des exemples de tas.

Nous utiliserons le type OCaml suivant pour représenter les tas.

```
type tas = Vide | Noeud of int * tas * tas
```

On commence par définir les fonctions d'ajout et de suppression sur les tas, en supposant donnée une fonction `fusion : tas -> tas -> tas` pour faire l'union de deux tas.

1. Définir une fonction `enleve_racine` de type `tas -> tas` qui supprime le plus petit élément d'un tas.
2. Définir une fonction `ajouter` de type `tas -> int -> tas` pour ajouter un élément dans un tas.

À l'aide des fonctions `enleve_racine` et `ajouter`, il est facile d'implémenter le tri par tas.

3. Définir une fonction `tas_of_list` de type `int list -> tas` qui construit un tas à partir d'une liste d'entiers.
4. Définir une fonction `list_of_tas` de type `tas -> int list` qui renvoie la liste, par ordre croissant, des éléments d'un tas.
5. En utilisant les fonctions précédentes, définir une fonction `trier` de type `int list -> int list` qui trie par ordre croissant une liste d'entiers.

La fusion de deux tas est l'opération la plus complexe. Supposons donnés deux tas `a` et `b` non-vides (l'opération de fusion est immédiate quand l'un des deux tas est vide). Nous avons donc `a = Noeud(m,g,d)` et `b = Noeud(n,l,r)`. Supposons $\texttt{m} \leq \texttt{n}$ (le cas $\texttt{n} \leq \texttt{m}$ est symétrique). La fusion consiste à construire un nouveau tas dont la racine est `m`, le sous-arbre gauche est `d` et le sous-arbre droit est le résultat de la fusion de `g` et `b`.

6. Définir la fonction `fusion` de type `tas -> tas -> tas` en suivant l'algorithme donné ci-dessus.