Université Paris Diderot — Programmation Fonctionnelle
Licence — Année 2010-2011

○

Arbres binaires

1 Listes (révisions)

Rappelons que la fonction `List.map` permet d'appliquer une fonction à chacun des éléments d'une liste. Par exemple `List.map (fun x -> x + 1) [3; 2; 7]` s'évalue en `[4; 3; 8]`.

1. Ecrire une fonctions `enum` telle que `enum n` renvoie la liste `[0 ;1 ;... ;n]`.
2. A partir de cette fonction et de `List.map`, écrire une fonction `enum_droite` telle que `enum_droite i n` renvoie la liste `[(i,0) ;(i,1) ;... ;(i,n)]`.
3. En déduire le code de `enum_paires : int -> int -> int list` telle que `enum_paires n p` renvoie la liste des éléments de $\{0, \ldots, \mathtt{n}\} \times \{0, \ldots, \mathtt{p}\}$ dans l'ordre lexicographique. Exemple :

   ```
   enum_paires 1 2 = [(0,0);(0,1);(0,2);(1,0);(1,1);(1,2)]
   ```

4. En utilisant cette fonction, écrire une fonction `table_mult n` calculant la table de multiplication de 0 à `n`, sous la forme d'une liste de triplets $(i, j, i \times j)$. Exemple :

   ```
   table_mult 2 = [(0,0,0);(0,1,0);(0,2,0);
                   (1,0,0);(1,1,1);(1,2,2);
                   (2,0,0);(2,1,2);(2,2,4)]
   ```

2 Arbres binaires

On peut effectuer en Caml des *déclarations de types*, c'est-à-dire créer de nouveaux types de valeurs. Les arbres binaires étiqueés par des entiers peuvent par exemple être représentés en Caml à l'aide de la déclaration de type suivante :

```
type tree = Nil | Node of int * tree * tree;;
```

`Nil` et `Node` seront appelés les *constructeurs* du type `tree`. Une fois ce type déclaré, on peut construire des valeurs de type `tree` :

```
Nil;;
Node (42, Nil, Nil);;
Node (42, Node(10, Nil, Nil), Nil);;
```

Comme pour les listes, une fonction peut être définie par cas sur la forme d'un arbre :

```
let rec f a = match a with
  Nil       -> ...
| Node(n,g,d) -> ...
;;
```

Exercice 1

Ecrire les fonction suivantes :

- `taille : tree -> int` renvoyant la taille d'un arbre, c'est-à-dire son nombre de noeuds internes.
- `hauteur : tree -> int` renvoyant la hauteur d'un arbre, c'est-à-dire la longueur de sa plus longue branche.
- `detect : tree -> int -> bool`, telle que `detect a n` renvoie `true` si et seulement si l'un des nœuds de l'arbre est étiqueté par `n`.
- `complet : tree -> bool` déterminant si un arbre est complet, c'est-à-dire si toutes ses feuilles sont à la même profondeur,

Exercice 2

On rappelle qu'un *arbre binaire de recherche* est un arbre dans lequel pour *chaque* noeud, on a la propriété suivante :

- soit `n` l'étiquette de ce nœud :
  - toutes les étiquettes du fils gauche de ce nœud sont $<$ `n`.
  - toutes les étiquettes du fils droit de ce nœud sont $>$ `n`.

Ecrire les fonctions suivantes :

- `detect_abr : int -> tree -> bool` tel que si `a` est un ABR et `n` un entier, `detect_abr n a` renvoie `true` si `n` apparaît dans l'arbre, et `false` sinon.
  L'exploration de `a` devra être minimale, en tenant compte du fait qu'il s'agit d'un ABR.
- `ajout_abr : int -> tree -> tree` tel que si `a` est un ABR et `n` est un entier, `ajout_abr` renvoie l'arbre `a` dans lequel on a ajouté `n` s'il n'y est pas déjà. Le résultat doit être encore un ABR. Si `n` est déjà dans l'arbre `a`, cette fonction doit renvoyer simplement `a`.
- `est_abr : tree -> bool` prenant en argument un arbre quelconque, et déterminant s'il s'agit d'un ABR. A noter que cette fonction nécessite de déclarer plusieurs fonctions auxiliaires, qui pourront être déclarées en fonctions locales.