TRAVAUX PRATIQUES XV

Introduction aux arbres

Dans ce TP, les questions marquées *(Bonus)* sont à réserver pour la fin du TP ou si vous êtes en avance. Sinon, priorisez de passer à la suite pour avoir le temps d'explorer un peu chaque partie.

A Généralités

Dans ce TP, on représente les arbres binaires (non stricts) avec le type OCaml suivant :

```
type 'a tree = Nil | Node of 'a tree * 'a * 'a tree
```

A.1 Génération d'arbres

Dans `arbres.ml` , un premier arbre vous est fourni :

```
let arbre_exemple = Node( Node(Node(Nil, 5, Nil),
                               3,
                               Node(7, Nil, Nil) ),
                          42,
                          Node(Nil, 24, Nil) )
```

Il correspond à l'arbre suivant :

Terminal

```
arbre_exemple :
       42
      /  \
     3   24
    / \
   5   7
```

Bien entendu, il ne suffira pas. Vous devez faire **vos propres tests** entre chaque fonction, comme d'habitude ! Pour vous y aider, codons quelques fonctions de génération d'arbre :

0. Écrire une fonction `peigne_gauche : int -> int tree` tel que `peigne_gauche h x` est le peigne gauche de hauteur `h` dont les noeuds internes contiennent h..0 de la racine jusqu'à la feuille.

1. *(Bonus)* Faire de même sauf que les étiquettes valent 0..h .

2. Écrire une fonction `parfait : int -> int tree` tel que `parfait h` soit un arbre parfait de hauteur `h` où chaque noeud est étiquetté par la hauteur du sous-arbre qu'il enracine.

A.2 Manipulations élémentaires

3. Écrire une fonction `taille : 'a tree -> int` qui calcule le nombre total de noeuds d'un arbre.

4. Écrire une fonction `hauteur : 'a tree -> int` qui calcule la hauteur d'un arbre.

5. Écrire une fonction `somme : int tree -> int` qui calcule la somme des étiquettes des noeuds d'un arbre étiquetté par des entiers. Comment qualifieriez-vous cet algorithme ?

A.3 DFS

6. Écrire une fonction `ordre_prefixe : 'a tree -> 'a list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe. On veut qu'en tête de la liste soit le premier noeud traité dans l'ordre préfixe, jusqu'au fond de la liste qui doit contenir le dernier. On pourra utiliser `@` pour concaténer des listes.
7. *(Bonus)* `@` est couteux (quel est son coût ?) Réécrire cette fonction sans `@` .
8. Écrire une fonction `ordre_infixe : 'a tree -> 'a list` qui fait de même mais dans l'ordre infixe. De même pour *(Bonus)*.
9. Écrire une fonction `ordre_postfixe : 'a tree -> 'a list` qui fait de même mais dans l'ordre postfixe. De même pour *(Bonus)*.

A.4 Arbres particuliers

10. Écrire une fonction `est_strict : 'a tree -> bool` qui teste si un arbre binaire est strict.
11. Écrire une fonction `est_parfait : 'a tree -> bool` qui teste si un arbre binaire est strict.

B Arbres d'arité quelconque

Dans cette partie, on étudie des arbres dont les noeuds peuvent avoir un nombre arbitraire d'enfants :

```
type 'a arbre = Nil_k | Node_k of 'a * 'a arbre list
```

Si l'on omet le « _k », OCaml ne saurait pas comment typer `Nil`. Plus précisément, il utiliserait le dernier `Nil` défini. Il y a plusieurs façons d'éviter ce problème ; ici on fait au plus simple : au lieu de créer un nouveau `Nil` on crée `Nil_k`.

12. Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre d'arité quelconque. *Indication :* `List.map` ainsi qu'un maximum de liste (à coder) seront utiles.
13. Écrire une fonction `transfo_LCRS : 'a arbre -> 'a tree` qui applique la transformation LCRS à un arbre d'arité quelconque
    (Indication) : Vous pourrez faire une fonction auxiliaire récursive qui prend en argument le noeud à traiter et ses adelphes.
14. Écrire une fonction `inverse_LCRS :'a tree -> 'a arbre` qui « détransforme » un arbre binaire en arbre d'arité k en appliquant la transformation LCRS inverse. On pourra lever l'exception `Invalid_argument "inverse_LCRS"` (cf cours Bonnes Pratiques) si jamais le `'a tree` n'a pas pu être obtenu via LCRS.