Feuille de travaux pratiques
Arbre binaire de recherche

Ce sujet est inspiré du cours de Jean-François Monin à l'Université Grenoble-Alpes.

Le but de l'exercice est traiter des arbres binaires de recherche (ABR). Pour le typage, ce sont simplement des arbres binaires.

```
type 'a abr = Leaf | Node of 'a abr * 'a * 'a abr;;
```

Mais il doivent en plus vérifier la condition suivante : pour chaque noeud, tous les éléments contenus dans le sous-arbre gauche doivent être inférieurs ou égaux aux éléments contenus dans le sous-arbre droit. Dans toute la suite, on supposera connue une fonction de comparaison, de type `'a -> 'a -> bool` sur le type considéré, qui renvoie vrai si le premier argument est inférieur au second et faux sinon. Elle sera systématiquement passée aux différentes fonctions (on ne le précisera pas à chaque fois).

1. Ecrire trois arbres de type `abr` : un vide, un d'au moins 3 noeuds et qui est bien un ABR, et un d'au moins 3 noeuds et qui n'est pas un ABR.
2. Ecrire une fonction `recherche` de type `('a -> 'a -> bool) -> 'a -> 'a abr -> bool` qui renvoie vrai si son deuxième argument est présent dans son troisième, supposé un ABR, et faux sinon.
3. Ecrire une fonction `insere` de type `('a -> 'a -> bool) -> 'a -> 'a abr -> 'a abr` qui insère son deuxième argument dans son troisième, supposé un ABR.
4. Ecrire une fonction `supprime` de type `('a -> 'a -> bool) -> 'a -> 'a abr -> 'a abr` qui supprime son deuxième argument de son troisième, supposé un ABR (s'il y est présent).

Première vérification ABR Dans la suite, on veut vérifier qu'un arbre de type `abr` est bien un ABR, de deux façons différentes. Dans un premier temps, on collecte dans une liste tous les éléments de l'arbre, dans l'ordre (gauche < droite), et on vérifie que cette liste est triée.

5. Ecrire un prédicat `liste_triee` de type `('a -> 'a -> bool) -> a list -> bool` qui prend en argument un ordre (par exemple `fun x y -> x < y`) et une liste, et renvoie vrai si la liste est triée pour l'ordre considéré, faux sinon (indication : pour qu'une liste ne soit pas triée, il suffit qu'il y ait deux éléments qui se suivent et qui ne respectent pas l'ordre).
6. Ecrire une fonction `collecte : 'a abr -> 'a list` qui collecte les éléments contenus dans un arbre supposé ABR, dans l'ordre de leur apparition.
7. Ecrire une fonction `est_abr1 : ('a -> 'a -> bool) -> 'a abr -> bool` qui renvoie vrai si un arbre de type `abr` est bien un ABR, faut sinon.

Deuxième vérification ABR Dans un deuxième temps, on passe par un itérateur sur les `abr`, qui compare tous les éléments dans le sous-arbre gauche avec ceux du sous-arbre droit.

8. Ecrire une fonction `check` de type `('a -> bool) -> 'a abr -> bool` qui prend en argument un prédicat et un arbre, et qui vérifie que tous les éléments de l'arbre vérifient le prédicat.
9. En utilisant `check`, écrire une fonction `est_abr2` qui vérifie qu'un arbre est bien un ABR.

Efficacité On veut maintenant observer l'efficacité de ces deux fonctions.

10. On peut tracer les appels à une fonction avec l'instruction `#trace <nom_fonction>`. Tracer les deux fonctions `est_abr1` et `est_abr2` sur des arbres pas trop gros. Qu'en déduisez-vous ?
11. On peut observer le temps de calcul avec la fonction `Sys.time : unit -> float`[1]. Ecrire une fonction `observe : ('a -> 'b) -> 'a -> float` qui renvoie le temps de calcul d'un appel de fonction.
12. Observer le temps de calcul de `est_abr1` et `est_abr2` sur des arbres assez gros (voir le fichier associé au TP sur madoc). Comment l'expliquez-vous ?

1. Voir `https://ocaml.org/api/Sys.html`