Outils logiques et algorithmiques – TP 1 – Arbres binaires

Ces exercices sont intégralement à réaliser en caml. Les premiers exercices reprennent plusieurs programmes du cours : essayez dans un premier temps de les reproduire sans regarder le poly !

Exercise 1 (Échauffement : exponentiation rapide) On veut écrire une fonction Caml `puiss: int -> int -> int` telle que `puiss` k n calcule k^n. Voici deux manières de caractériser le nombre k^n

$$\begin{array}{rcl} k^0 & = & 1 \\ k^{n+1} & = & k \times k^n \end{array} \qquad\qquad \begin{array}{rcll} k^0 & = & 1 & \\ k^n & = & \begin{cases} (k \times k)^{n/2} & n > 0 \wedge n \text{ pair} \\ k \times (k \times k)^{n/2} & n > 0 \wedge n \text{ impair} \end{cases} \end{array}$$

Écrire deux versions de la fonction `puiss`, suivant chacune de ces deux stratégies. □

Exercise 2 (Tri de listes chaînées)

Pour commencer.

1. Écrire une fonction `est_triee: int list -> bool` testant si la liste donnée en paramètre est triée.

Définition récursive du tri par insertion.

— la liste vide est triée

— pour trier une liste $e :: \ell$, on trie la liste ℓ puis on insère e à la bonne position dans la liste obtenue

Réalisation du tri par insertion.

2. Écrire une fonction `insere: int -> int list -> int list` qui prend en paramètres un entier x et une liste ℓ supposée triée, et qui renvoie la liste triée obtenue en insérant x à une position adaptée de ℓ.
3. Écrire une fonction `tri_insertion: int list -> int list` qui prend en paramètre une liste ℓ et renvoie une liste triée comportant les mêmes éléments que ℓ.
4. Comment modifier ces fonctions pour que le tri supprime à la volée les doublons ?

Définition récursive du tri par fusion.

— la liste vide est triée

— une liste singleton est triée

— pour trier une liste avec au moins deux éléments, on la sépare en deux parties qu'on trie indépendamment, puis on fusionne les résultats

Réalisation du tri par fusion.

5. Écrire une fonction `separe: int list -> int list * int list` qui prend en paramètre une liste ℓ et qui renvoie une paire de listes ℓ_1, ℓ_2 qui se partagent les éléments de ℓ. Faire en sorte que ℓ_1 et ℓ_2 aient des tailles aussi proches que possible.
6. Écrire une fonction `fusion: int list -> int list -> int list` qui prend en paramètres deux listes ℓ_1 et ℓ_2 supposées triées, et qui renvoie une liste triée contenant les éléments de ℓ_1 et de ℓ_2.
7. Écrire une fonction `tri_fusion: int list -> int list` qui prend en paramètre une liste ℓ et renvoie une liste triée comportant les mêmes éléments que ℓ.

Définition récursive du tri par sélection.

— la liste vide est triée

— pour trier une liste non vide ℓ, on cherche le plus petit élément m de ℓ, on trie la liste obtenue en retirant m de ℓ, puis on place m en tête du résultat

Réalisation du tri par sélection.

8. Écrire une fonction `min_list: int list -> int` qui prend en paramètre une liste ℓ supposée non vide, et qui renvoie son plus petit élément.
9. Écrire une fonction `remove: int -> int list -> int list` qui prend en paramètres un entier x et une liste ℓ supposée contenir x, et qui renvoie une liste contenant les mêmes éléments que ℓ, moins une occurrence de x.
10. Écrire une fonction `tri_selection: int list -> int list` qui prend en paramètre une liste ℓ et renvoie une liste triée comportant les mêmes éléments que ℓ.

Bonus.

11. Toutes les fonctions précédentes sauf une peuvent être écrites avec exclusivement des fonctions récursives terminales.

□

Exercise 3 (Arbres binaires) On se donne la définition Caml suivante pour le type des arbres binaires.

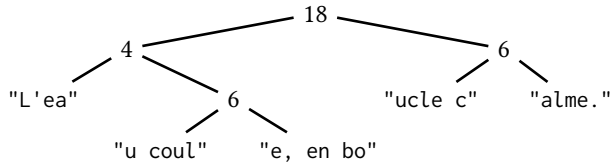
```
type arbre =
  | V
  | N of arbre * arbre
```

Dans cette définition, les nœuds ne portent aucune information. On ne s'intéresse donc ici qu'à la forme des arbres.

1. Écrire une fonction `taille: arbre -> int` renvoyant la taille de l'arbre donné en paramètre.
   Taille : nombre de nœuds.
2. Écrire une fonction `hauteur: arbre -> int` renvoyant la hauteur de l'arbre donné en paramètre.
   Hauteur : profondeur maximale d'un nœud +1*, ou* 0 *pour l'arbre vide.*
3. Écrire une fonction `equilibre: arbre -> bool` testant si l'arbre donné en paramètre est équilibré.
   Équilibré : en chaque nœud, les hauteurs du sous-arbre gauche et du sous-arbre droit diffèrent au plus de 1*.*
4. Écrire une fonction `equilibre': arbre -> int option` renvoyant `Some h` si l'arbre donné en paramètre est équilibré et a une hauteur `h`, ou `None` s'il n'est pas équilibré. *Sans utiliser les fonctions précédentes.*
5. Écrire une fonction `complet: arbre -> bool` testant si l'arbre donné en paramètre est localement complet.
   Localement complet : chaque nœud a zéro ou deux fils.
6. Écrire une fonction `parfait: arbre -> bool` testant si l'arbre donné en paramètre est parfait.
   Parfait : complet, et toutes les feuilles sont à la même profondeur.
7. Écrire une fonction `genere_parfait: int -> arbre` prenant en paramètre un entier h et renvoyant un arbre parfait de hauteur h.

□

Exercise 4 (Cordes) La structure de *corde* (en anglais, *rope*) permet des manipulations fluides des grandes chaînes de caractères. Une corde a la forme d'un arbre binaire non vide, où les feuilles portent des fragments de la chaîne représentée, et où les nœuds internes relient les fragments entre eux. La chaîne représentée est obtenue en concaténant les chaînes portées par chaque feuille, de gauche à droite. Ainsi, l'arbre



représente la chaîne `"L'eau␣coule,␣en␣boucle␣calme."`. Pour permettre un accès efficace aux différentes positions de la chaîne, chaque nœud d'une corde contient en outre un entier, indiquant le nombre total de caractères de toutes les chaînes présentes aux feuilles de son sous-arbre gauche. On se donne la définition suivante d'un type Caml pour les cordes.

```
type corde =
  | Feuille of string
  | Noeud   of int * corde * corde
```

1. Donner une valeur Caml représentant la corde donnée en exemple ci-dessus.
2. Écrire une fonction `to_string: corde -> string` renvoyant la chaîne de caractères représentée par la corde donnée en paramètre.
3. Écrire une fonction `longueur: corde -> int` prenant en paramètre une corde c et renvoyant la longueur de la chaîne représentée par c.
4. Écrire une fonction `char: int -> corde -> char` prenant en paramètres un entier i et une corde c, et renvoyant le caractère à l'indice i de la chaîne représentée par c. On considérera comme d'habitude que le premier caractère a l'indice 0. La fonction doit échouer si l'indice i est invalide.
5. Écrire une fonction `concat: corde -> corde -> corde` prenant en paramètres deux cordes c_1 et c_2 et renvoyant une corde représentant la concaténation des chaînes représentées par c_1 et c_2. *Il suffit presque de renvoyer un nouveau nœud ayant pour fils* c_1 *et* c_2*.*
6. Écrire une fonction `coupe: int -> corde -> corde * corde` prenant en paramètres un entier i et une corde c, et renvoyant deux cordes c_1 et c_2 telles que c_1 contient les i premiers caractères de c, et c_2 les autres. La fonction doit échouer si l'indice i est invalide.
7. Écrire une fonction `insere: string -> int -> corde -> corde` prenant en paramètres une chaîne de caractères s, un entier i et une corde c, et renvoyant le corde obtenue en insérant la chaîne s à partir de l'indice i dans c (les caractères suivants sont décalés). *Il suffit de combiner* `coupe` *et* `concat` *de la bonne manière.*
8. Écrire une fonction `supprime: int -> int -> corde -> corde` prenant en paramètres deux entiers i et l et une corde c, et renvoyant la corde obtenue en supprimant l caractères à partir de l'indice i dans c. *Il suffit de combiner* `coupe` *et* `concat` *de la bonne manière.*

□