Exercices sur les arbres

Généralités

Sauf exception, les exercices de cette feuille utilisent le type :

```
type 'a arbre = Nil | Node of 'a arbre * 'a * 'a arbre;;
```

Exercice 1. 1. Ecrire la fonction **branchedroite** de type **'a arbre -> 'a list** qui renvoie la liste des étiquettes de la branche droite de l'arbre en paramètre.

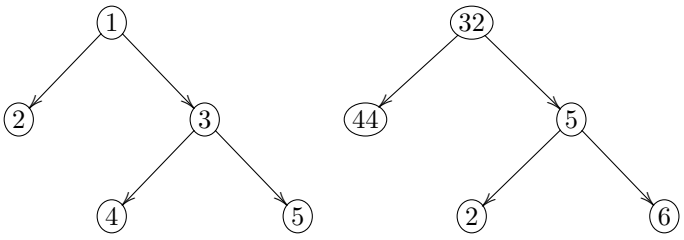
```
# a;;
- : int arbre =
  Node (Node (Node (Nil, 2, Nil), 1, Node (Nil, 3, Nil)), 4,
  Node (Node (Nil, 5, Nil), 6, Node (Nil, 7, Nil)))

# branchedroite a;;
- : int list = [4; 6; 7]
```

2. Ecrire la fonction `plusAdroite : 'a arbre -> 'a list` qui retourne le chemin d ela racine à la feuille la plus à droite.

```
# let feuille x = Node(Nil, x, Nil);;
val feuille : 'a -> 'a arbre = <fun>
# let a = let deux = feuille 2 and quatre = feuille 4
  and sept = feuille 7 and huit = feuille 8 in
  let six = Node(sept,6,huit) in let cinq = Node(six,5,Nil)
  in let trois = Node(quatre,3,cinq) in Node(deux,1,trois)
in plusAdroite a;;
        - : int list = [1; 3; 5; 6; 8]
```

Exercice 2. 1. Ecrire la fonction `val test_squelette : 'a arbre -> 'b arbre -> bool` qui teste l'égalité des *squelettes* de deux arbres arbres binaires.
Par exemple, les arbres suivants ont même squelette :



2. Écrire la fonction `inclus : 'a arbre -> 'b arbre -> bool` qui teste si le squelette du premier arbre est inclus dans celui du second.

```
let deux = feuille 2 and quatre = feuille 4
and sept = feuille 7 and huit = feuille 8 in
let six = Node(sept,6,huit) in let cinq = Node(six,5,Nil)
in let trois = Node(quatre,3,cinq) in let un = Node(deux,1,trois) in inclus
    cinq un, inclus (Node(deux,3,six)) un;;
```

Exercice 3. La numérotation *Sosa* d'un arbre binaire est donnée par :

— le numéro de la racine est 1 ;

— Si k est le numéro d'un nœud, alors son fils gauche (resp. droit) porte le numéro $2k$ (resp. $2k+1$)

— On ne numérote pas l'arbre vide.

Ecrire la fonction `sosa : 'a arbre -> (int * 'a) arbre` telle que `sosa a` reconstruit l'arbre en argument en ajoutant le numéro Sosa à l'étiquette de chaque nœud.

```
# a;;
- : int arbre =
Node (Node (Node (Nil, 2, Nil), 1, Node (Nil, 3, Nil)), 4,
Node (Node (Nil, 5, Nil), 6, Node (Nil, 7, Nil)))

# sosa a;;
- : (int * int) arbre =
Node (Node (Node (Nil, (4, 2), Nil), (2, 1), Node (Nil, (5, 3), Nil)),
 (1, 4), Node (Node (Nil, (6, 5), Nil), (3, 6), Node (Nil, (7, 7), Nil)))
```

Exercice 4. Montrer que dans un arbre binaire entier, le nombre de feuilles est égal au nombre de nœuds internes plus 1.

Exercice 5. Ecrire la fonction `completg : 'a arbre -> bool` telle que **completg a** renvoie vrai si l'arbre est complet gauche.

Injectivité du parcours en profondeur suffixe

Exercice 6. Dans cet exercice et les suivants, on considère le type :

```
type 'a btree =
  (*modélise les arbres binaires entiers*)
  | F of 'a
  | N of 'a * 'a btree * 'a btree;;
```

Les étiquettes ont le même type pour les nœuds internes et les feuilles. Les arbres ainsi implémentés sont entiers et l'arbre vide n'est pas une possibilité.

Implanter la fonction **suffixe** qui retourne la liste des nœuds d'un arbre dans l'ordre du parcours en profondeur suffixe. Un élément de la liste renvoyée est du type **label** suivant

```
type 'f label = L of 'f| V of 'f
```

Par exemple `L("5")` désigne une feuille (*leaf*) d'étiquette **5** et `V("+")` désigne un nœud interne (*vertice*) d'étiquette +.

```
let bt = N("+", F "5" , N("x", F "6", F"7"));;
suffixe bt;;
```

Le retour est **[L "5" ; L "6" ; L "7" ; V "x" ; V "+"]**

Exercice 7. Fait référence à l'exercice 6.

On veut montrer que la fonction précédente est injective sur les arbres binaires entiers, c'est à dire que si `suffixe a` est égal à `suffixe b`, alors **a** est égal à **b**.

Nous notons de façon plus concise $s(a)$ pour **suffixe(a)** : c'est la liste suffixe des étiquettes de a dans laquelle on a distingué les feuilles des nœuds internes. On note **@** l'opérateur de concaténation des listes.

Soit p une liste d'étiquettes distinguées. On note D la différence entre le nombre de L (feuilles) et de V (nœuds internes) dans P. Par exemple, si p est **[L "5" ; L "6" ; L "7" ; V "x" ; V "+"]**, alors $D(p) = 1$.

1. (a) Si A est un arbre binaire entier montrer que $D(s(A)) = 1$.
   (b) (Lemme) Si $s(A) = p' @ p$" et $p' \neq \emptyset$ alors $D(p') > 0$.
2. En déduire l'injectivité de l'application.
3. Montrer que l'application s n'est pas une surjection de l'ensemble des arbres entiers non vides étiquetés par E vers l'ensemble des listes d'étiquettes distinguées.

Exercice 8. Fait référence à l'exercice 6.

Écrire le code de la fonction **build_tree** qui prend en paramètres une liste de **'a label** et retourne l'unique **a' btree** dont elle est issue.

Pa exemple :

```
let bt = N("a",
           N("b", F "c", F "d"),
           N("e", F "f", N("g",F "h", F "i")));;
let flat =  suffixe bt;;
build_tree flat;;
```

Et le retour dela dernière instruction est
N ("a", N ("b", F "c", F "d"), N ("e", F "f", N ("g", F "h", F "i")))

Arbres quelconques

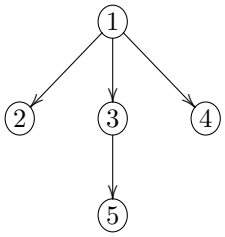
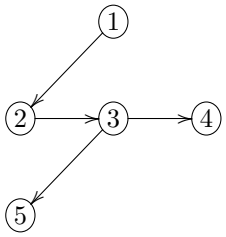
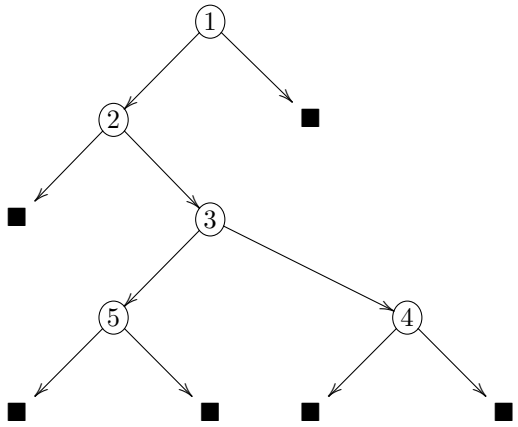
Exercice 9. On considère le type d'arbre suivant :

```
type 'a arbreqc = N of 'a * 'a arbreqc list;;
```

Il modélise le type des arbres d'arité quelconque. L'arbre vide n'est pas représenté et **N(x,[])** représente une feuille. L'expression **N(x,[a1 ;a2 ;a3 ;a4])** représente un arbre à 4 fils. Par convention on dit que a_1 est le *fils aîné*, que a_1, a_2, a_3, a_4 sont *frères* et que a_{k+1} est le *frère suivant* de a_k. Les arbres quelconques sont naturellement associés à une structure hiérarchique appelée « left-child ; next-sibling » (voir plus bas).

On considère l'application qui a un arbre de type **arbreqc** associe un arbre **btree** dans lequel chaque nœud admet pour fils gauche l'arbre associé à son fils aîné, et pour fils droit, l'arbre associé à son frère suivant.

Ainsi l'arbre quelconque à gauche est transformé en l'arbre binaire de droite :

Repésentation conventionnelle	left-child ; next-sibling	Résultat (arbre binaire)
		

1. Dessiner l'arbre quelconque puis l'arbre binaire associés à

```
let a = N(1,[N(2,[N(5,[]);N(6,[])]);N(3,[N(7,[])]); N(4,[])]);;
```

2. Écrire la fonction `convert : 'a arbreqc -> 'a btree` qui convertit un arbre quelconque en arbre binaire.
3. Écrire une fonction `inverse : 'a arbre -> 'a arbreqc` qui réalise l'inverse de l'opération précédente (si c'est possible).

```
        # inverse @@ convert aqc = aqc;;
        - : bool = true
```

Études de complexité

Exercice 10. On donne la fonction **subtrees a p** qui prend en arguments un entier p et un arbre binaire a et qui retourne la liste des sous-arbres non vides dont la racine est à la profondeur p dans a.

```
let rec subtrees a p = match a with
  | Nil -> []
  | b when p=0-> [b]
  | Noeud(g,x,d) -> (subtrees g (p-1))@(subtrees d (p-1));;
```

1. Un arbre binaire A est *parfait* lorsqu'il est vide ou lorsque ses fils gauche et droit sont parfaits et de même hauteur.
   On veut calculer le nombre d'ajout en tête de liste (AJT) pour un arbre donné.
   (a) Lorsque a est un arbre binaire parfait de hauteur p, combien d'insertions u_p en tête de liste la l'appel **subtrees a p** effectue-t-il ? Donner cette valeur en fonction de p.
   Indication On commence par donner une relation de récurrence vérifiée par u_p et on étudie la suite auxiliaire $v_p = \dfrac{u_p}{2^p}$.
   (b) Exprimer cette complexité en fonction de n, le nombre de nœuds d'un arbre parfait.

2. Ecrire une meilleure version de la fonction (avec un accumulateur).
3. Lorsque a est un arbre binaire parfait de hauteur p, combien d'insertions en tête de liste votre fonction effectue-t-elle ?

Exercice 11. Un arbre binaire A est *parfait* lorsque ses fils gauche et droit sont parfaits et de même hauteur. Pour déterminer si un arbre binaire est parfait, on propose la fonction suivante :

```
let rec est_parfait = function
  | Nil -> true
  | Node (fg, _, fd) -> est_parfait fg && est_parfait fd
                        && hauteur fg = hauteur fd ;;
```

1. Dans le cas d'un arbre parfait, évaluer en fonction de $n = |A|$ le coût temporel de cette fonction.
2. Lorsque l'arbre n'est pas parfait, l'évaluation paresseuse rend difficile l'étude de complexité. On peut faire une majoration qui n'en tient pas compte.

   Notation : dans toute la suite, on note c_n la complexité au pire du calcul de **est_parfait a** pour un arbre de taille n.

   (a) On note c_n la complexité au pire pour un arbre de taille n.

   Soit A est un arbre binaire de taille n. Si $k \in [\![0, n-1]\!]$ est la taille de son fils gauche, la complexité de l'étude du fils gauche est inférieure à c_k et celle pour l'étude du fils droit est inférieure à c_{n-1-k}. Le calcul des hauteurs des fils explore $n-1$ nœuds, la complexité de ces calculs est donc en $O(n)$.

   Alors la complexité pour l'étude d'un arbre dont le fils gauche a k nœuds est majorée par $c_k + c_{n-k-1} + n$. Ainsi

$$c_n = \max_{k=0,\dots,n-1} (c_k + c_{n-k-1}) + \underbrace{n}_{\text{calcul de hauteur}}$$

   Etablir une majoration quadratique en fonction de n pour c_n.
3. Donner une meilleure version pour vérifier qu'un arbre est parfait.