Arbres binaires en OCaml
Seconde salve d'exercices

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]
```

Solution. Voici

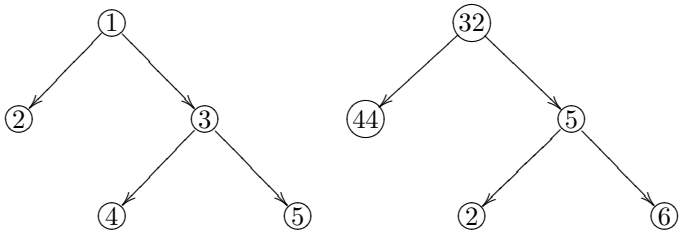
```
let rec branchedroite a = match a with
  | Nil -> []
  | Node (_,x,d) -> x::(branchedroite d);;

let plusAdroite a =
  let rec pad a acc = match a with
    | Nil -> List.rev acc
    | Node(g,x,Nil) -> pad g (x::acc)
    | Node(_,x,d) -> pad d (x::acc)
  in pad a [];;
```

□

Exercice 2. 1. Ecrire la fonction `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;;
```

Solution. Voici

```
let rec test_squelette a1 a2 = match a1,a2 with
  | Nil, Nil -> true
  | Nil, _ -> false
  | _,Nil -> false
  | Node (g1,_,d1), Node(g2,_,d2)-> test_squelette g1 g2
                                    && test_squelette d1 d2;;
```

□

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)))
```

Solution. Voici

```
let  sosa a =
  let rec aux a k =
    match a with
    | Nil -> Nil
```

```
    | Node(g,x,d) -> Node (aux g (2*k), (k,x), aux d (2*k+1))
  in aux a 1;;
```

□

Exercice 4. Dans un arbre binaire entier, exprimer le nombre de feuilles en fonction du nombre de nœuds internes. Justifier.

Solution. Ce résultat est vrai pour les feuilles. On note $i(A)$ le nombre d enœuds internes d'un arbre A et $f(A)$ son nombre de feuilles.

Soit $A = N(x, g, d)$ un arbre binaire entier non réduit à une feuille. On suppose que $i(g) + 1 = f(g)$ et $i(d) + 1 = f(d)$ (hypothèse d'induction).

Alors

$$i(A) + 1 = i(g) + i(d) + 1 + 1 = f(g) + f(d) = f(A).$$

CQFD

□

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

Voici :

Solution. Voici

```
let rec sons f =
  (*f : forêt des nodes à une profondeur donnée -pas la dernièr-*)
  (*Avant le dernier niveau,    aucun node ne peut être Nil*)
  let rec aux f acc = match f with
  | [] -> List.rev acc
  | Nil::q ->
     failwith "pas complet gauche"
  | Node(g,_,d)::q-> aux q (d::(g::acc))
 in aux f [];;

let rec dernier f =
  (*à utiliser si f est le dernier
   niveau*)
  match f with
  | [] -> true
  | Nil::Node(_,_,_)::q ->
     false
  | _ ::q-> dernier q;;

```

```
let completg0 a =
  let h = hauteur a in
  let rec aux f h = match h with
    | 0 -> dernier f
    | _ -> try
           aux (sons f) (h-1)
         with Failure _ -> false
  in aux [a] h;;
```

□

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 "+"]`

Solution. Un premier code

```
    let rec suffixe a = match a with
      | F(x) -> [L x]
      | N(x,g,d)->suffixe g@suffixe d@[V x];;
```

Cependant avec

```
      a
   b     c
 d   e f   g
```

la fonction ajoute `L d` une première fois à gauche de `[L e; V b]`. Et ce résultat `[L d;L e;V b]` est ensuite ajouté à gauche de `[L f;L g;V c;V a]`. Donc `L d` est ajouté deux fois : ce n'est pas bon !

On va plutôt gérer un accumulateur :

```
let suffixe2 t =
  let rec aux acc t = match t with
    | F(x) -> (L x)::acc
    | N(x,g,d) -> let acc' = aux ((V x)::acc) d in
      aux acc' g in
  aux [] t;;
```

□

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.

Solution. L'application n'est pas une surjection car `[L 1, L 2]` n'est pas la liste suffixe d'un arbre (il faut qu'un arbre à deux feuilles ait une racine, laquelle est un nœud interne).

Si `p` est la liste suffixe d'un arbre binaire entier `a`,

— le nombre d'éléments de la forme `V _` dans `p` est égal au nombre de nœuds internes de `a`.

— Et le nombre d'éléments de la forme `L _` est le nombre de feuilles.

— En conséquence la différence $D(p)$ entre le nombre de feuilles et de nœuds internes est 1 puisque `a` est binaire entier (Ce résultats se montrent par récurrence sur la hauteur de l'arbre ou par induction).

Lemme **Si $s(a) = p' @ p''$ et le préfixe p' est non vide alors $D(p') \geq 1$.**

Preuve du lemme On le montre par récurrence FORTE sur la longueur de `p'`.

1. Si la longueur est 1, alors p' est de la forme `[L e]` (le premier élément de p est une feuille), et on a bien $D(p') \geq 1$.
2. Si l'HR est vérifiée lorsque le membre gauche de la concaténation est de longueur $i \leq k$ tel que $1 \leq k \leq |s(a)| - 1$, alors considérons p' de longueur $k + 1$.

— Si le dernier élément de p' est de la forme $L\,e$ (feuille), alors on peut écrire

$$s(a) = \underbrace{q \,@\, [L\,e]}_{=p'} \,@\, p"$$

où q est non vide de longueur k.

Comme q est de longueur k on a $D(q) \geq 1$ par HR.

Et si on ajoute $[L\,e]$

$$D(p') = D(q) + D([L\,e]) = D(q) + 1 > 1$$

— Supposons que le dernier élément de p' soit de la forme $V\,e$ (nœud interne).

Par construction, tous les descendants du sous-arbre b de a dont le nœud racine a donné $V\,e$ sont à gauche de $V\,e$.

On peut donc écrire $p' = q \,@\, s(b)$ où $s(b)$ est la liste suffixe du sous-arbre dont la racine est représentée par $V\,e$ (q peut être vide). Ce sous-arbre b étant binaire entier, il vient que $D(s(b)) = 1$.

Si q est vide alors $D(p') = 1$.

Et on a aussi $D(q) \geq 1$ par HR si q non vide (puisque $s(a) = q \,@\, (s(b) \,@\, p")$ et $|q| < |p|'$). Alors $D(p') = D(q) + D(s(b)) \geq 1$. CQFD

Autre preuve du lemme On peut aussi raisonner par induction : on montre que la propriété « tous les préfixes non vides de $s(a)$ ont un Delta strictement positif » est vraie.

C'est immédiat pour l'arbre-feuille.

Considérons un arbre dont les fils gauche g et droit d vérifient HI. On a $s(a) = s(g)@s(d)@[V\,x]$ si x est l'étiquette de la racine.

Considérons ensuite un préfixe p non vide. Si $p = s(a)$, alors $D(p) = 1$ puisque a est entier.

Si p est un préfixe strict :

— Soit p est un préfixe de g. Alors $D(p) > 0$ par HI sur g.

— Soit p s'écrit $s(g)@p'$ avec $|p'| > 0$. Alors p' est un préfixe de $s(d)$, et, par HI, $D(p') > 0$.

On a donc

$$D(p) = \underbrace{D(s(g))}_{=1} + \underbrace{D(p')}_{>0 \text{ par HI}} > 0$$

Hérédité OK

Retour au problème On considère deux arbres binaires entiers `a=N(x,g,d)` et `b=N(x',g',d')` de même image `p`. Ainsi les deux arbres ont le même nombre de feuilles et de nœuds internes du fait de la première remarque.

On raisonne par récurrence forte sur le nombre de nœuds internes de deux arbres a, b tels que $s(a) = s(b)$. HR(n) = « si le nombre de nœuds internes est n pour a, b, alors $s(a) = s(b)$, entraîne que $a = b$ ».

Cas de base Le cas où $n = 0$ (il n'y a qu'une feuille) est évident.

Hérédité Considérons des arbres à $n + 1$ nœuds internes.

Le dernier élément de $s(a)$ est de la forme $V\,x$, celui de $s(b)$ est $V\,x'$. On a donc $x = x'$.

On note g, d, g', d' les fils gauche et droits de a et b.

Par égalité de $s(a) = s(b)$ et l'unicité du préfixe d'une longueur donnée, on a

$$s(g)@s(d) = s(g')@s(d')$$

Égalité des longueurs des listes suffixes de gauche Si $|s(g)| = |s(g')|$ alors $s(g) = s(g')$ par unicité du préfixe d'une longueur donnée.

Et comme g et g' ont moins de n nœuds internes, $s(g) = s(g')$ entraîne par HR que $g = g'$

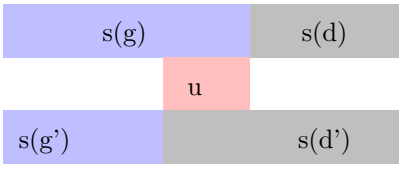
Par régularité à gauche, de $s(g)@s(d) = s(g')@s(d')$ et $s(g) = s(g')$, on tire $s(d) = s(d')$. Donc $d = d'$ par HR.

Et donc les arbres binaires entiers a et b ayant les mêmes sous-arbres gauches et droits et la même racine sont égaux.

Une liste suffixe est plus longue que l'autre Si $|s(g)| > |s(g')|$, alors il existe u telle que :

$$s(g') \,@\, u = s(g),\ |u| > 0 \text{ et } u \,@\, s(d) = s(d')$$

(propriété de Levi : intuitive mais admise -cf. cours sur les mots-).



— $D(g) = 1$ car $s(g)$ est la liste suffixe d'un arbre binaire entier.

— De même $D(g') = 1$.

— $D(s(g') \,@\, u) = D(s(g)) = 1$. Donc $D(s(g')) + D(u) = 1$. Et par suite $D(u) = 0$

— Mais u est un préfixe non vide de $s(d')$ donc $D(u) > 0$ par le lemme. Absurde.

Donc u est vide. Et par suite $|s(g)| = |s(g')|$. Ce qui entraîne que $a = b$ comme on l'a vu.

Le cas $|s(g)| < |s(g')|$ se traite identiquement.

□

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")))`

Solution. Voici

```
let build_tree l =
  let rec construire lb ls = (*ls : liste suffixe; lb : liste de btree*)
    match lb, ls with
    | _ , (L e)::q -> construire (F(e)::lb) q
    | d::g::qb, (V e)::qs ->  construire (N(e,g,d)::qb) qs (*noter l'inversion g et d*)
    | _ , (V e)::qb -> failwith "pas assez de fils pour un NI"
    | [t], [] -> t
    | _, _ -> failwith "pas le sufixe d'un arbre entier" in
  construire [] l;;
```

La fonction interne gère deux listes utilisées comme des piles : `l` qui contient au départ la liste « plate » des `'a node`, l'autre, `lb`, vide au départ est remplie avec des `'a btree`.

Si la liste `l` :

— commence par une feuille on l'empile dans `lb`.

— commence par un nœud interne `V e`, on vérifie qu'il a au moins deux prédecesseurs dans `lb`. On dépile les trois éléments de tête, on forme un arbre avec, et on empile cet arbre. Attention : l'ordre dans la pile inverse les fils droits et gauches.

Il faut bien faire attention à ce qu'un nœud interne ait toujours deux fils car on veut obtenir un arbre binaire entier. Donc un nœud interne dans `l`, auquel on ne pourrait pas fournir ses deux fils en les prenant dans `lb` doit soulever une exception.

— est vide et la liste `lb` ne contient qu'un seul arbre, c'est cet arbre qui est renvoyé.

— Les autres cas correspondent à des listes sans antécédent par la fonction `suffixe`.

□

Arbres quelconques

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

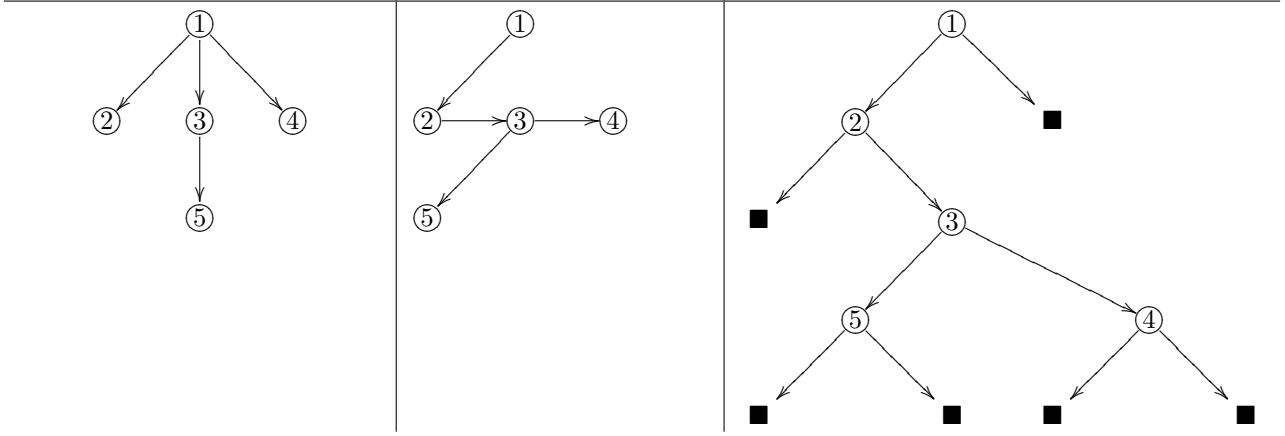
```
type 'a arbreqc = N of 'a * 'a arbreqc list;;
```

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 `arbre` 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 :

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



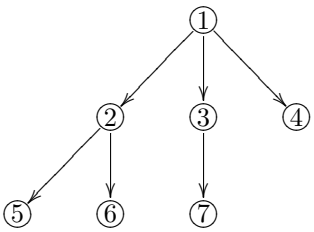
1. Dessiner l'arbre quelconque puis l'arbre binaire associés à

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

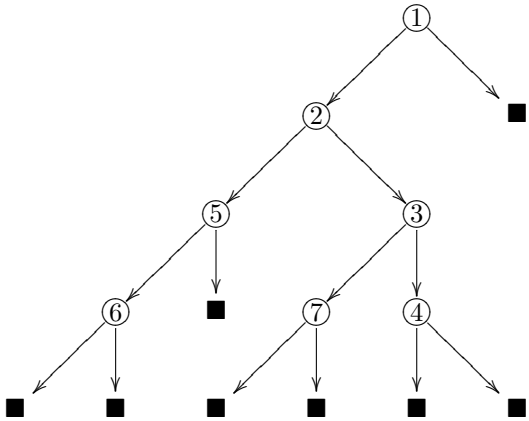
2. Écrire la fonction `convert : 'a arbreqc -> 'a arbre` qui convertit un arbre quelconque en arbre binaire.
3. Écrire une fonction `inverse : 'a arbre -> 'a arbreqc` qui réalise l'inverse de l'opération précédente (si c'est possible).

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

Solution. Voici

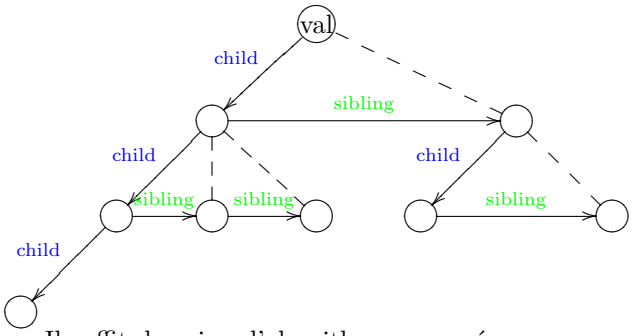


On obtient



Sur la figure **??** on compare la façon usuelle de représenter les arbres d'arité quelconque (en rouge) avec l'implémentation imposée par notre type `arbreqc` (appelée représentation « fils-gauche, frère droit »).

FIGURE 1 – Un arbre quelconque. En rouge les arcs de la représentation usuelle, en bleu et vert les arcs de la représentation « fils-gauche, frère droit »



Il suffit de suivre l'algorithme proposé

```
let convert a =
  let rec aux a acc = match a, acc with
    | N(x,[]), [] -> Node(Nil,x,Nil)
    | N(x,a::q), [] -> Node(aux a q,x,Nil)
    | N(x,[]), a::q -> Node (Nil, x, aux a q)
    | N(x,a::q), a'::q' -> Node(aux a q,x, aux a' q')
  in aux a [];;
```

Pour la réciproque, on remarque qu'on ne peut pas faire un arbre quelconque à partir d'un arbre binaire si ce dernier :

— est vide ;

— ou bien possède un fils droit non vide.

On définit une fonctin auxiliaire qui tansforme un arbre binaire en liste d'arbres quelconques.

Quand je suis sur un nœud `Node(g,x,d)` de l'arbre binaire, je forme un arbre quelconque en mettant

`x` comme étiquette de la racine, et la conversion du fils gauche comme premier fils, la liste des autres fils s'obtenant en appliqant la fonction de conversion au fils droit.

```
let inverse a =
let rec forest_of_bintree a  = match a  with
  | Nil -> []
  | Node(g,x,d) -> N(x, forest_of_bintree g) :: (forest_of_bintree d)
in
match forest_of_bintree a with
| [t] -> t
| _ -> failwith "pas possible"
;;
```

□