Correction TP11: Arbres

1 Arbres binaires

Concernant les tests : Il n'est pas pratique de créer des arbres en code d'un seul coup : on se perd facilement dans les virgules, les parenthèses, les indentations... Rien n'empêche de construire les arbres de test morceau par morceau. Par exemple :

```
let make_feuille (x: 'a) : 'a ab =
  N(x, V, V)


let tests () =
  let f1 = make_feuille 1 in
  let f2 = make_feuille 2 in
  let f3 = make_feuille 3 in
  let f4 = make_feuille 4 in
  let f5 = make_feuille 5 in
  let f6 = make_feuille 6 in

  let t7 = N(7, f5, f6) in
  let t8 = N(8, f3, f4) in
  let t9 = N(9, t7, t8) in
  let t10 = N(10, f1, f2) in

  let t = N(11, t9, t10) in
  (*
  11-+-9-+-7-+-5
     |   |   |
     |   |   +-6
     |   |
     |   +-8-+-3
     |       |
     |       +-4
     |
     +-10-+-1
          |
          +-2
   *)
  assert (hauteur t = 3);
  assert (hauteur V = -1);
  assert (hauteur t8 = 1);
  assert (hauteur f2 = 0);
```

Q1. On rappelle l'existence de la fonction `max: 'a -> 'a -> 'a` qui donne le maximum de deux éléments :

```
(* longueur maximale d'un chemin dans a. -1 pour l'arbre vide *)
let rec hauteur (a: 'a ab) : int =
```

```
  match a with
  | V -> -1
  | N (x, g, d) -> 1 + max (hauteur g) (hauteur d)
```

Q2. Par définition des feuilles :

```
(* Renvoie true si a est un arbre réduit à une feuille,
   false sinon. *)
let rec est_feuille (a: 'a ab) =
  match a with
  | N(x, V, V) -> true
  | _ -> false

(* Renvoie le nombre de feuilles de a *)
let rec feuilles (a: 'a ab) =
  match a with
  | V -> 0
  | N(x, V, V) -> 1
  | N(x, g, d) -> feuilles g + feuilles d
```

Q3. Une manière simple d'écrire cette fonction est de faire le match-with sur le couple (arbre, chemin) :

```
(* Renvoie l'étiquette associée à chemin dans a,
   où true <-> droite et false <-> gauche. *)
let rec etiquette (a: 'a ab) (chemin: bool list) : 'a =
  match (a, chemin) with
  | V, _ -> failwith "chemin invalide" (* pas d'étiquettes dans l'arbre vide *)
  | N(x, g, d), [] -> x
  | N(x, g, d), true :: q -> etiquette d q
  | N(x, g, d), false :: q -> etiquette g q
```

Q4. On reprend le même schéma, mais au lieu de lever une erreur avec failwith, on renvoie None :

```
(* Renvoie l'étiquette associée à chemin dans a,
   où true <-> droite et false <-> gauche.
   Renvoie None si le chemin est invalide. *)
let rec etiquette_opt (a: 'a ab) (chemin: bool list) : 'a option =
  match (a, chemin) with
  | V, _ -> None
  | N(x, g, d), [] -> Some x
  | N(x, g, d), true :: q -> etiquette_opt d q
  | N(x, g, d), false :: q -> etiquette_opt g q
```

Q5. On rappelle que préfixe signifie "racine, puis enfant gauche, puis enfant droit", ce qui se traduit directement en code :

```
(* Affiche les éléments de a dans l'ordre
   préfixe *)
let rec print_prefixe (a: int ab) : unit =
  match a with
  | V -> ()
  | N(x, g, d) ->
    print_int x;
    print_newline ();
    print_prefixe g;
    print_prefixe d
```

Q6.

```
(* Affiche les éléments de a dans l'ordre
   postfixe *)
let rec print_postfixe (a: int ab) : unit =
  match a with
  | V -> ()
  | N(x, g, d) ->
    print_prefixe g;
    print_prefixe d;
    print_int x;
    print_newline ()

(* Affiche les éléments de a dans l'ordre
   infixe *)
let rec print_infixe (a: int ab) : unit =
  match a with
  | V -> ()
  | N(x, g, d) ->
    print_prefixe g;
    print_int x;
    print_newline ();
    print_prefixe d
```

Q7. Preuve par induction structurelle sur les arbres binaires que pour tout arbre a, pour tout $n \in \mathbb{Z}$, $\textbf{taille\_add}(a, n) = \textbf{taille}(a) + n$

- $a = \mathbf{V}$: $\textbf{taille\_add}(\mathbf{V}, n) = n = \textbf{taille}(V) + n$.
- $a = \mathbf{N}(x, g, d)$ avec g, d des arbres.
  Notons $n_g = \textbf{taille\_add}(g, n+1)$.
  Alors, par HI, $n_g = \textbf{taille}(g) + n + 1$.
  De plus :
  $\textbf{taille\_add}(a, n) = \textbf{taille\_add}(d, ng)$.
  Par HI, cela vaut :
  $\textbf{taille}(d) + n_g = \textbf{taille}(d) + \textbf{taille}(g) + n + 1$
  $= \textbf{taille}(a) + n$

On en déduit une nouvelle version de la fonction taille :

```
(* Renvoie la taille de a *)
let taille2 (a: 'a ab) = taille_add a 0
```

Q8. Sur le même principe, si l'on veut écrire une fonction `postfixe_concat a l` qui renvoie la liste des éléments de a en postfixe, concaténé à l, on voit que si $a = \mathbf{N}(x, g, d)$, alors :

$$\begin{aligned}
\textbf{postfixe_concat}(a, l) &= \textbf{postfixe}(a)@l \\
&= \textbf{postfixe}(g)@\textbf{postfixe}(d)@[x]@l \\
&= \textbf{postfixe}(g)@\textbf{postfixe}(d)@(x :: l) \\
&= \textbf{postfixe}(g)@(\textbf{postfixe_concat}(d, x :: l)) \\
&= \textbf{postfixe_concat}(g, \textbf{postfixe_concat}(d, x :: l))
\end{aligned}$$

Ce qui nous donne le code suivant :

```
(* postfixe_concat a l = postfixe a @ l *)
let rec postfixe_concat (a: 'a ab) (l: 'a list) : 'a list =
  match a with
  | V -> l
  | N(x, g, d) ->
    let ld = postfixe_concat d (x::l) in
```

```
    postfixe_concat g ld

(* Renvoie la liste des éléments de a dans l'ordre postfixe *)
let liste_postfixe2 (a: 'a ab) : 'a list =
  postfixe_concat a []
```

Q9.

Q10.

Q11.

```
(* Applique f sur chaque étiquette de a *)
let rec tree_map (f: 'a -> 'b) (a: 'a ab) : 'b ab =
  match a with
  | V -> V
  | N (x, g, d) -> N (f x, tree_map f g, tree_map f d)
```

Q12.

```
(* Renvoie la somme des éléments de a *)
let rec tree_sum (a: int ab) : int =
  match a with
  | V -> 0
  | N (x, g, d) -> x + tree_sum g + tree_sum d
```

Q13.

```
(* Renvoie true si x apparaît dans a, false sinon *)
let rec appartient (x: 'a) (a: 'a ab) : bool =
  match a with
  | V -> false
  | N (y, g, d) -> x = y || appartient x g || appartient x d
```

2 Reconstruction d'arbres stricts

3 Arbres généraux

L'archive de la correction contient deux versions du code : une avec les fonctions mutuellement récursives et une utilisant les fonctions du module List. Pour la première question, les deux sont montrées dans ce document, mais pour celles d'après, une des deux est choisie, et vous pourrez voir l'autre directement dans le code si besoin.

Q1. Avec les fonctions mutuellement récursives :

```
(* hauteur a renvoie la profondeur maximale d'une feuille de a *)
let rec hauteur (Node(x, l): 'a tree) : int =
  1 + hauteur_liste l
(* hauteur_liste l renvoie la hauteur maximale d'un arbre de l,
   -1 si l est vide *)
and hauteur_liste (l: 'a tree list) =
  match l with
  | [] -> -1
  | x :: q -> max (hauteur x) (hauteur_liste q)
```

En utilisant la fonction `List.fold_left` :

```
1
2  (* hauteur a renvoie la profondeur maximale d'une feuille de a,
3     -1 pour une liste vide *)
4  let rec hauteur (Node(x, l): 'a tree) : int =
5    1 + List.fold_left max (-1) (List.map hauteur l)
6
7
8  (* etiquette a c renvoie l'étiquette du noeud de a pointé par le chemin c.
9     Échoue avec failwith si c n'est pas un chemin valide dans a. *)
```

Q2. La fonction `List.nth i l` renvoie le i-ème élément d'une liste : c'est exactement ce que l'on cherche à faire ici :

```
1  (* etiquette a c renvoie l'étiquette du noeud de a pointé par le chemin c.
2     Échoue avec failwith si c n'est pas un chemin valide dans a. *)
3  let rec etiquette (Node(x, l): 'a tree) (c: int list) : 'a =
4    match c with
5    | [] -> x
6    | i :: c' ->
7      let a = List.nth l i in
8      etiquette a c'
```

Q3. Commençons par une version simple non-optimisée, utilisant des concaténations :

```
1  (* liste_prefixe a renvoie la liste des étiquettes de a
2     dans l'ordre préfixe *)
3  let rec liste_prefixe (Node(x, l): 'a tree) : 'a list =
4    x :: liste_prefixe_liste l
5
6  (* liste_prefixe_liste l renvoie la concaténation des listes
7     préfixes de tous les arbres de l *)
8  and liste_prefixe_liste (l: 'a tree list) : 'a list =
9    match l with
10   | [] -> []
11   | a :: q -> liste_prefixe a @ liste_prefixe_liste q
```

Comme pour les arbres binaires, la complexité est en $\Omega(n^2)$ à cause des concaténations. On peut reprendre le schéma vu à l'exercice 1 et faire une fonction auxiliaire utilisant un accumulateur :

```
1  (* renvoie liste_postfixe a @ accu *)
2  let rec postfixe_concat (Node(x, l): 'a tree) (accu: 'a list) =
3    List.fold_right postfixe_concat l (x::accu)
```

La fonction `List.fold_right` fonctionne comme `List.fold_left`, mais opère de droite à gauche, ce qui permet de faire les concaténations dans le bon ordre ici. La ligne va donc partir de $x :: accu$ puis y ajouter les éléments du dernier arbre de l puis de l'avant dernier, et ainsi de suite. En effet :

$$\begin{aligned} & \textbf{List.fold\_right}(\textbf{postfixe\_concat}, [a1; a2; ...an], (x :: accu)) \\ = \ & \textbf{postfixe\_concat}(a1, \textbf{postfixe\_concat}(a2, ...\textbf{postfixe\_concat}(an, x :: accu)\dots) \\ = \ & \textbf{postfixe}(a1)@(\textbf{postfixe}(a2)@(...\textbf{postfixe}(an)@(x :: accu))\dots) \\ = \ & \textbf{postfixe}(a)@accu \end{aligned}$$

Q4. ...

Q5.

```
let file_vide = ([], [])

(* enfile x dans f *)
let enfiler (x: 'a) (f: 'a file) =
  let t, q = f in
  (t, x::q)

(* défile la tete de f et la renvoie *)
let rec defiler (f: 'a file) : 'a * 'a file =
  match f with
  | [], [] -> failwith "file vide"
  | x::t, q -> (x, (t, q))
  | [], q -> let t = List.rev q in defiler (t, [])

let rec defiler ((t, q): 'a file) : 'a * 'a file =
  match t with
  | [] ->
    begin match List.rev q with
    | [] -> failwith "File vide"
    | x :: r -> x, (r, [])
    end
  | x :: r -> (x, (r, q))

(* true si f est vide, false sinon *)
let est_vide (f: 'a file) =
  f = ([], [])
```

Q6. Il pouvait être utile de faire une fonction auxiliaire permettant d'enfiler tous les éléments d'une liste dans une file. En effet, dans la boucle du parcours en largeur, on doit enfiler tous les enfants du noeud défilé :

```
(* Fonction auxiliaire pratique pour le parcours en largeur: faire une fonction
qui enfile toute une liste d'éléments dans une file *)
let rec enfiler_liste (l: 'a list) (f: 'a file) : 'a file =
  match l with
  | [] -> f
  | x :: q ->
    let f' = enfiler x f in
    enfiler_liste q f'
```

L'indication du sujet permet alors de simuler la boucle du parcours à l'aide d'une fonction récursive :

```
(* liste des étiquettes de a par ordre de
   profondeurs croissantes *)
let liste_largeur (a: 'a tree) =
  (* liste des étiquettes des éléments de f.
     Simule la boucle de l'algorithme itératif. *)
  let rec liste_largeur_file (f: 'a tree file) =
    if est_vide f then [] else
    let (Node(x, l), f) = defiler f in
    (* Enfiler tous les éléments de l dans la file *)
    let prochaine_file = enfiler_liste l f in
    (* ajouter x au résultat, et continuer le parcours (notons que ce n'est pas
       récursif terminal...) *)
    x :: liste_largeur_file prochaine_file
  in
  liste_largeur_file (enfiler a file_vide)
```

Bonus : Nous avons vu qu'il n'est pas pratique de rendre des fonctions sur les arbres récursives terminales, car il y a systématiquement un appel à gauche puis un appel à droite, ou l'inverse. Cependant, en passant par les boucles, on peut le faire assez facilement. Par exemple, reprenons la fonction précédente, en récursif terminal :

```
let liste_largeur_rec_term (a: 'a tree) =
  let rec liste_largeur_file (f: 'a tree file) (res: 'a list) =
    if est_vide f then res else
    let (Node(x, l), f) = defiler f in
    let prochaine_file = enfiler_liste l f in
    liste_largeur_file prochaine_file (x :: res)
  in
  (* penser à renverser le résultat *)
  let f_init = enfiler a file_vide in
```

De plus, nous avons vu que les parcours en profondeur peuvent aussi s'écrire avec une boucle, en passant par le même algorithme que le parcours en largeur et en remplaçant la file par une pile. Ceci permet d'écrire en récursif terminal certaines des fonctions vues aux exercices 1 et 3 ! Par exemple, voici une fonction calculant la somme des éléments d'un arbre binaire, en récursif terminal :

```
(* renvoie la somme des éléments de a *)
let somme (a: int ab) : int =
  (* Fait un parcours en profondeur à partir
     de la pile d'arbres p, et calcule la
     somme de toutes les étiquettes, en utilisant
     res comme accumulateur *)
  let rec boucle (p: int ab list) (res: int) : int =
    match p with
    | [] -> res
    | V :: q -> boucle q res
    | N(x, g, d) :: q ->
      (* empiler g et d, mettre à jour la somme,
         puis relancer la boucle *)
      let p' = g :: d :: q in
      let res' = x + res in
      boucle p' res'
  in
  boucle [a] 0
```

4 Arbres préfixes