TP4: Arbres

MPSI Lycée Pierre de Fermat

Dans ce TP, certaines fonctions sont assez complexes. Il est fortement conseillé d'écrire en entier le commentaire de documentation **avant** même de réfléchir au code: cela vous permettra de mieux raisonner et vous aidera à implémenter ces fonctions.

1 Exercice 1: Arbres binaires

On propose le type suivant pour les arbres binaires:

```
type 'a ab =
  | V (* Vide *)
  | N of 'a * 'a ab * 'a ab (* Noeud: elem, gauche, droite *)

let t =
N(3,
  N(5,
    V,
    N(8, V, V)
  ),
  N(7, V, V)
)

(* taille a renvoie le nombre total de noeuds de a *)
let rec taille (a: 'a ab) : int =
  match a with
  | V -> 0
  | N(x, g, d) -> 1 + taille g + taille d

let test_taille () =
  assert (taille V = 0);
  assert (taille t = 4);
  assert (taille (N(2, t, t) = 1)
```

Q1. Dessiner sur une feuille un arbre d'au moins 12 nœuds, puis le recréer dans votre code. *Conseil: ne le codez pas en un seul coup, utilisez des variables intermédiaires pour créer les différents nœuds.*

Q2. Écrire une fonction `hauteur` permettant de calculer la hauteur d'un arbre binaire. On rappelle que la hauteur d'un arbre vide est -1.

Q3. Écrire une fonction permettant de calculer le maximum d'un arbre. On pourra commencer par implémenter la fonction auxiliaire suivante:

```
(* Renvoie l'élément maximal parmi m et les éléments de a. *)
let rec maximum_compare (a: 'a arbre) (m: 'a) : 'a = ...
```

Q4. Écrire une fonction `tree_sum: int ab -> int` calculant la somme des entiers contenus dans un arbre binaire.

Q5. Écrire une fonction `tree_map: ('a -> 'b)-> 'a ab -> 'b ab` qui applique une fonction f sur chaque nœud d'un arbre a.

Q6. Écrire une fonction `appartient : 'a -> 'a ab -> bool` déterminant si un élément se trouve dans un arbre binaire.

On représente un chemin dans un arbre binaire par une liste de booléens: true indique que l'on part à droite, et false que l'on part à gauche.

Q7. Écrire une fonction `etiquette: 'a ab -> bool list -> 'a` qui renvoie l'étiquette du nœud correspondant à un chemin dans un arbre. Si le chemin n'est pas valide, une erreur sera levée avec `failwith`.

Parcours

On rappelle qu'un parcours **préfixe** d'un arbre est une opération qui traite d'abord l'étiquette à la racine d'un arbre avant de traiter les sous-arbres. De même, un parcours **postfixe** traite l'étiquette à la racine après avoir traité les sous-arbres. Enfin, pour les arbres binaires, il existe également le parcours **infixe**, dans lequel on traite l'étiquette à la racine après le sous-arbre gauche, mais avant le sous-arbre droit.

Q8. Écrire une fonction `print_prefixe: int ab -> unit` qui affiche les étiquettes d'un arbre binaire d'entiers, dans l'ordre préfixe. On affichera chaque étiquette sur une ligne.

On se propose maintenant d'écrire une fonction renvoyant la **liste** des éléments d'un arbre dans l'ordre d'un parcours postfixe. En OCaml, la fonction `List.append: 'a list -> 'a list -> 'a list` permet de concaténer deux listes. Par exemple:

```
utop # List.append [1;2;3] [4;5;6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
```

En OCaml, on peut simplement utiliser l'opérateur `@`, qui est simplement un raccourci pour `List.append`. On peut donc écrire:

```
utop # [1;2;3] @ [4;5;6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
```

Attention, le calcul de `l1 @ l2` est **linéaire en la taille de** l_1, i.e. de complexité $\mathcal{O}(|l_1|)$ ($|.|$ dénotant la taille).
On peut considérer la fonction suivante qui calcule un parcours postfixe:

```
let rec postfixe (a: 'a ab) : 'a list =
  match a with
  | V -> []
  | N(x, g, d) -> postfixe g @ (postfixe d @ [x])
```

Cette fonction est correcte, mais coûteuse. En effet, elle vérifie l'équation de complexité suivante (n_1 et n_2 étant les tailles respectives de g et d):

$$C(n) = C(n_1) + C(n_2) + \mathcal{O}(n_2) + \mathcal{O}(n_1) \tag{1}$$

Pour simplifier, nous allons étudier l'équation suivante:

$$C(n) = C(n_1) + C(n_2) + n_2 + n_1 \tag{2}$$

Les termes $+n_2$ et $+n_1$ représentent les coûts des deux concaténations. Tentons de minorer le pire cas pouvant survenir. Pour $n \in \mathbb{N}$, on définit le peigne droit de taille n, noté P_n, par:

- $P_0 = V$
- Pour $n > 0$, $P_n = N(n, V, P_{n-1})$

Q9. Dessiner $P_0, P_1, \ldots, P_5$, puis représenter P_n de manière générale.

Q10. On note $D(n)$ le coût d'exécution de la fonction `postfixe` sur l'arbre P_n. Proposer une relation de récurrence sur P_n en réécrivant l'équation (2) ci-dessus.

Q11. En déduire la valeur de $D(n)$.

Ainsi, dans le pire cas, la fonction `postfixe` s'exécute au moins en temps quadratique. Le but des questions suivantes est de trouver une implémentation en $\mathcal{O}(n)$. Pour cela, il faut donc procéder **sans concaténation**. Nous allons utiliser une fonction auxiliaire prenant en paramètre additionnel un accumulateur, une liste dans laquelle on viendra ajouter un à un les éléments croisés. Pour mieux comprendre le principe, on commence par étudier une manière alternative de coder la fonction `taille`.

Q12. On définit la fonction suivante:

```
let rec taille_add (a: 'a ab) (n: int) : int =
  match a with
  | V -> n
  | N(x, g, d) ->
    let ng = taille_add g n in (* ajouter àn la taille de g *)
    let nd = taille_add g ng in (* ajouter àng la taille de d *)
    nd + 1 (* ajouterànd l'élément du noeud actuel *)
```

Montrer par induction structurelle sur les arbres que `taille_add a n = taille a + n` pour tout arbre `a` et pour tout entier `n`. En déduire une nouvelle fonction `taille2: 'a ab -> int` calculant la taille d'un arbre.

Q13. En s'inspirant de ce schéma, écrire une fonction `postfixe2` sans utiliser l'opérateur de concaténation `@`. On pourra passer par une fonction auxiliaire:

```
(* Renvoie la liste des éléments de a dans l'ordre postfixe.
   Autrement dit, postfixe2 a = postfixe a *)
let postfixe2 (a: 'a ab) : 'a list =
  (* Renvoie (postfixe a) @ l *)
  let rec post_concat (a: 'a ab) (l: 'a list) : 'a list =
    ...
  in post_concat a []
```

Q14. Montrer formellement la correction de votre fonction. On commencera par montrer la correction de la fonction auxiliaire.

Q15. (**Optionnel**) Déterminer un schéma commun dans les fonctions des questions 2 à 6, et proposer une fonction `tree_fold` permettant de les généraliser, tout comme `fold_left` permet de généraliser de nombreuses fonctions sur les listes (cf TP sur les listes).

On souhaite écrire une fonction qui numérote les étiquettes d'un arbre dans l'ordre préfixe. Autrement dit, on souhaite remplacer chaque nœud $N(x, g, d)$ par un nœud $N((x, i), g, d)$, avec i le numéro du nœud dans l'ordre préfixe.

Q16. Écrire une fonction `num_liste: 'a list -> ('a * int)list` numérotant une liste à partir de 0. Par exemple:

```
utop # num_liste ['A'; 'B'; 'C']
- : (char * int) list = [('A', 0); ('B', 1); ('C', 2)]
```

On pourra passer par une fonction auxiliaire prenant en argument un indice à partir duquel numéroter.

Q17. En s'inspirant de la question précédente, écrire une fonction:

```
num_pref: 'a arbre -> ('a * int) arbre
```

numérotant un arbre dans l'ordre préfixe.