TP12: Arbres

MP2I Lycée Pierre de Fermat

Vous ***devez*** **documenter** vos fonctions, c'est à dire décrire leurs sorties en fonction de leurs entrées (ce qui **doit** faire apparaître chaque nom de paramètre), les **typer** et les **tester**: ce n'est pas facultatif !

Il est vital de de le faire **avant** d'écrire le code de la fonction et pas après. En effet, pour des fonctions complexes, il sera très utile d'avoir devant les yeux la spécification précise ainsi que le type de chaque paramètre, pour pouvoir réfléchir récursivement, et le simple fait d'écrire des tests va vous aider à mieux concevoir ce que la fonction doit faire, et donc comment l'implémenter.

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 *)

(* Exemple *)
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) = 9)
```

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

Question 2. Écrire une fonction `est_feuille` qui détermine si un arbre binaire est une feuille, puis une fonction `feuilles` comptant le nombre de feuilles d'un arbres.

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.

Question 3. Écrire une fonction `etiquette: 'a ab -> bool list -> 'a` qui renvoie l'étiquette du noeud correspondant à un chemin dans un arbre. Si le chemin n'est pas valide, une erreur sera levée avec `failwith`.

OCaml possède un type somme pré-existant (comme les listes) appelé le type **option**:

```
type 'a option =
  | None
  | Some of 'a
```

Ce type permet de rajouter librement à chaque type une valeur ayant le sens "pas de valeur" (un peu comme None en Python). Il est très courant d'utiliser ce type lorsque l'on écrit une fonction qui peut ne pas renvoyer de résultat dans certains cas. Par exemple, si l'on veut calculer le max d'une liste:

```
(* max_opt l renvoie l'élément maximum de l.
   Renvoie None si l est vide *)
let rec max_opt (l: 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: q ->
    begin match max_opt q with
      | None -> Some x
      | Some y -> Some (max x y)
    end
```

De même, plusieurs fonctions de la librairie standard d'OCaml sur les listes ont une version qui renvoie une option. Par exemple, on peut lire dans la documentation:

```
val find_opt : ('a -> bool) -> 'a list -> 'a option
  find_opt f l returns the first element of the list l that satisfies the predicate f.
  Returns None if there is no value that satisfies f in the list l.
```

Question 4. Écrire une fonction `etiquette_opt` qui renvoie l'étiquette associée à un chemin binaire dans un arbre, et renvoie `None` si le chemin est invalide.

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.

Question 5. Écrire une fonction `print_prefixe: int ab -> unit` qui affiche les étiquettes d'un arbre binaire d'entiers, dans l'ordre préfixe.

Question 6. Écrire deux fonctions `liste_prefixe: 'a ab -> 'a list` et `liste_infixe`, calculant la liste des éléments d'un arbre binaire, dans l'ordre préfixe et infixe. On rappelle l'existence de l'opérateur binaire `@` permettant de concaténer deux listes, et on s'autorise à l'utiliser pour cette question.

Question 7. Écrire une fonction `liste_postfixe`. Il pourra être judicieux d'écrire d'abord une fonction auxiliaire qui calcule la liste postfixe à l'envers, afin d'éviter d'utiliser la concaténation de manière abusive.

Question 8. Réécrire `liste_prefixe` sans utiliser l'opérateur de concaténation `@` (ou bien une fonction de concaténation). On pourra passer par une fonction auxiliaire, prenant en paramètre additionnel une liste d'étiquettes.

Question 9. (**Optionnel**) Faire de même avec `liste_infixe` et `liste_postfixe`

Question 10. Écrire une fonction `tree_map: ('a -> 'b)-> 'a ab -> 'b ab` qui applique une fonction f sur chaque noeud d'un arbre a.

Question 11. Écrire une fonction `tree_sum: int ab -> int` calculant la somme des entiers contenus dans un arbre binaire.

Question 12. Écrire une fonction `appartient : 'a -> 'a ab -> bool` déterminant si un élément se trouve dans un arbre binaire.

Question 13. (**Optionnel**) Déterminer un schéma commun dans les dernières fonctions, et proposer une fonction `tree_fold` permettant de les généraliser, tout comme `fold` permet de généraliser de nombreuses fonctions sur les listes.

Pour les deux dernières questions, on considère des arbres dont les étiquettes sont des couples (k, v). On peut alors voir ces arbres comme une implémentation concrète des dictionnaires.

Question 14. Écrire une fonction `assoc_opt : 'k -> ('k * 'v)ab -> 'v option` prenant en entrée un élément k et un arbre A de couples clé-valeur, et renvoyant **une** valeur v telle que (k, v) est une étiquette de A. Cette fonction renverra `None` si aucune étiquette n'a k comme clé.

Question 15. Écrire une fonction `assoc_all : 'k -> ('k * 'v)ab -> 'v list` prenant en entrée un élément k et un arbre A de couples clé-valeur, et renvoyant **la liste** de toutes les valeurs v telle que (k, v) est une étiquette de A.

Exercice 2: Arbres quelconques

On souhaite étudier maintenant des arbres d'arité quelconque. Un noeud ne stockera plus un tuple d'enfants mais une liste. On ne représente plus l'arbre vide.

```
type 'a tree =
  Node of 'a * ('a tree list) (* Noeud: étiquette, liste des enfants *)
```

Dans cette partie, nous allons implémenter plusieurs opérations sur les arbres. Certaines questions demandent de coder des fonctions auxiliaires, ou d'utiliser des fonctions sur les listes déjà implémentées dans OCaml: map, fold, filter, etc... On rappelle que la documentation des différentes fonctions standards sur les listes se trouve ici: v2.ocaml.org/api/List.html.

Il y a deux manières standards d'écrire d'écrire des fonctions sur le type `'a tree`. La première consiste à écrire **deux** fonctions mutuellement récursives: une qui agit sur les arbres, une autre qui agit sur les listes d'arbres. Par exemple, pour calculer la taille d'un arbre:

```
(* Calcule la taille de l'arbre Node(x, l) *)
let rec taille (Node(x, l) : 'a tree) : int =
  1 + taille_liste l

(* Calcule la somme des tailles des arbres
   dans l *)
and taille_liste (l: 'a tree list) : int =
  match l with
  | [] -> 0
  | d :: q -> taille d + taille_liste q
```

Le mot clé **and** permet de lier les deux fonctions afin de pouvoir les définir récursivement l'une par rapport à l'autre.

La deuxième version consiste à utiliser de manière judicieuse les fonctions pré-existantes sur les listes. Par exemple, pour calculer la taille d'un arbre:

```
(* Calcule la taille de l'arbre Node(b, l) *)
let rec taille2 (Node(x, l) : 'a tree) : int =
  1 + List.fold_left (+) 0 (List.map taille2 l)
```

Dans la suite du TP, vous pouvez utiliser la méthode que vous voulez. Il est cependant conseillé de faire quelques questions avec chacune des deux, pour vous entraîner.

Question 16. Implémenter une fonction `hauteur` qui calcule la hauteur d'un arbre

Question 17. Implémenter une fonction `etiquette` qui prend en entrée un arbre et une liste d'entiers $[a_1; \ldots; a_n]$ représentant un chemin, et qui renvoie l'étiquette du noeud correspondant. Si le chemin n'est pas valide dans l'arbre, la fonction lèvera une exception.

Question 18. Écrire les fonctions `liste_prefixe` et `liste_postfixe` de parcours préfixe et postfixe.

On introduit un nouveau type de parcours: les ***parcours en largeur***. Un tel parcours énumère les noeuds d'un arbre par ordre croissant de profondeur. Il est plus facile de décrire ces parcours de manière itérative, avec une boucle, qu'avec de la récursivité. On utilise une file pour effectuer des parcours en profondeur:

Algorithme 1 : Parcours en profondeur

Entrée(s) : A un arbre

$r \leftarrow$ la racine de A ;
$F \leftarrow$ une file vide ;
F.**enfiler**(r);
tant que $F \neq \emptyset$ **faire**
    $u \leftarrow F$.**defiler**();
    Traiter u;
    pour v ***enfant de*** u **faire**
        F.**enfiler**(v);

Remarque 1. Si l'on remplace la file par une pile, on retrouve les parcours en profondeur.

Voyons comment implémenter récursivement ces parcours. Nous allons utiliser deux listes, qui en OCaml se comportent comme des piles, pour simuler une file. Plus précisément, on utilisera une contion auxiliaire comme suit:

```
let liste_largeur (a: 'a tree) =
  let rec liste_largeur_niveaux (l_actuel: 'a tree list) (l_suivant: 'a tree list) : 'a list
      =
    ...
  in liste_largeur_niveaux [a] []
```

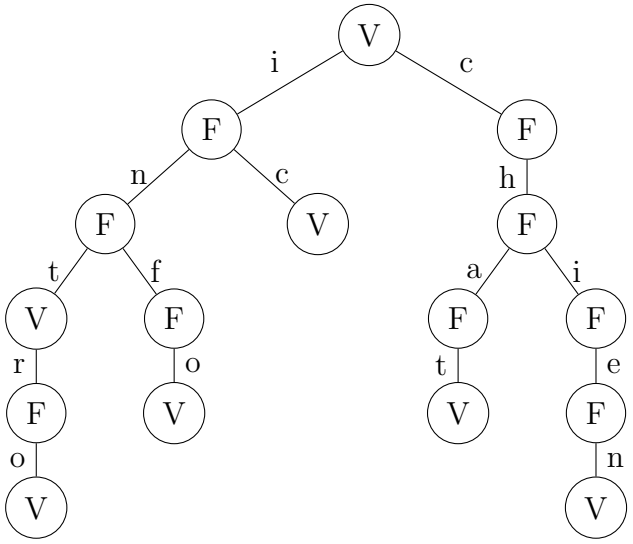
La spécification de la fonction auxiliaire est qu'elle calcule la liste des éléments dans les arbres des listes `l_actuel` et `l_suivant`. La liste `l_actuel` correspondra aux arbres du niveau actuel regardé, et la liste `l_suivant` correspondra aux arbres du niveau suivant.

Ainsi, la fonction auxiliaire devra vérifier l'invariant suivant: `l_suivant` contient des noeuds de l'arbre initial d'un profondeur d'exactement 1 de plus que les noeuds de `l_actuel`. En particulier, chacune des deux listes contiendra des noeuds d'une seule profondeur à la fois.

Question 19. Implémentez le parcours en largeur par une fonction `liste_largeur`.

Exercice 3 (Optionnel): Arbres préfixes

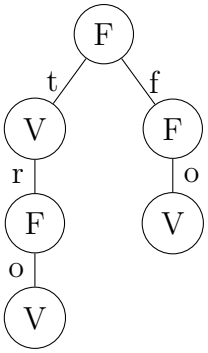
Les arbres préfixes servent à stocker des ensembles de mots. L'idée est de stocker sur chaque arête entre un noeud et son parent une lettre. Un noeud stockera un booléen, et représentera le mot formé de toutes les lettres entre la racine et lui. Par exemple:



Les noeuds contenant des V (comme vrai) correspondent à des mots de l'ensemble. Dans l'exemple au dessus, l'arbre représente l'ensemble $\{\varepsilon$, int, intro, info, ic, chat, chien$\}$. On propose le type suivant pour représenter ces arbres:

```
1 type pre_tree = Node of (bool * (char * pre_tree) list)
```

Par exemple, voici un arbre préfixe et sa représentation sous ce type en OCaml:




```
1  let t =
2  Node(false, [
3    ('t', Node(true, [
4      ('r', Node(false, [
5        ('o', Node(true, []))
6      ]))
7    ]));
8    ('f', Node(false, [
9      ('o', Node(true, []))
10   ]))
11 ])
```

On peut transformer une liste de caractères en un string comme suit en OCaml:

```
1 let string_of_list l = String.of_seq (List.to_seq l)
```

Question 20. Écrire une fonction `taille` qui calcule le nombre de mots contenus dans un arbre préfixe.

Question 21. Écrire une fonction `plus_long` qui calcule la longueur du plus long mot contenu dans un arbre préfixe.

Question 22. Écrire une fonction `rechercher: string -> pre_tree -> bool` qui détermine si un mot est dans un arbre préfixe.

Question 23. Écrire une fonction `ajouter: string -> pre_tree -> pre_tree` qui ajoute un mot à un arbre préfixe.

Question 24. Écrire une fonction `enumerer` qui renvoie la liste des mots contenus dans un arbre préfixe, par ordre alphabétique.

Question 25. Écrire une fonction `enumerer_prefixe: string -> pre_tree -> string list` telle que pour s un mot et t un arbre préfixe, `enumerer_prefixe s t` renvoie la liste des mots de t qui commencent par s.

Question 26. Rendre la fonction `enumerer` linéaire en la taille de l'arbre (donc interdit d'utiliser une fonction de tri ou de concaténer des listes à tout va). Il pourra être utile de modifier les fonctions précédentes pour obliger les enfants d'un noeud à être stockés par ordre alphabétique.

Question 27. En vous renseignant sur la fonction `read_line` ainsi que sur le module `String` d'OCaml (documentation: v2.ocaml.org/api/String.html), implémentez une fonction `construire` telle que pour a un arbre préfixe, `construire a` lit dans le terminal une ligne de mots séparés par des espaces, et renvoie l'arbre a dans lequel tous les mots ont été rajoutés.

Question 28. Écrire une fonction `fusion` permettant de faire l'union de deux arbres préfixes. Attention à ne pas créer de doublons.