TP11: Arbres

MP2I 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.

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) = 9)
```

Q1. Écrire une fonction `hauteur` permettant de calculer la hauteur d'un arbre binaire. On rappelle que la hauteur d'un arbre vide est -1.

Q2. 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 arbre.

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.

Q3. 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`.

En plus des listes, OCaml possède un autre type somme pré-existant: les **options**:

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

Ce type permet de rajouter librement à chaque type une valeur ayant le sens “pas de valeur”. Il permet donc d'écrire des fonctions qui peuvent ne rien renvoyer. Par exemple, pour calculer le maximum d'une liste, jusqu'ici, la liste vide levait une erreur qui arrêtait le programme. Avec le type option, on peut procéder comme suit:

```
(* max_opt l renvoie Some m avec m 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[1]:

```
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.
```

Q4. 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.

[1]Traduction: `find opt l` renvoie le premier élément de l qui vérifie le prédicat f, et renvoie None si aucune valeur de l ne satisfait l.

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.

Q5. Écrire une fonction `print_prefixe: int ab -> unit` qui affiche les étiquettes d'un arbre binaire d'entiers, dans l'ordre préfixe.

Q6. Écrire de même deux fonctions `print_postfixe` et `print_infixe`.

On se propose maintenant d'écrire une fonction renvoyant la **liste** des éléments d'un arbre dans l'ordre d'un parcours postfixe. En utilisant l'opérateur de concaténation de liste `@`, on peut écrire:

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

Cependant, la concaténation est coûteuse: elle est linéaire en la taille de la liste de gauche. Ainsi, la fonction `liste_postfixe` vérifie l'équation de complexité

$$C(n) = C(n_1) + C(n_2) + \Theta(n)$$

Avec n_1 et n_2 les tailles des deux enfants. On rappelle que dans le cas d'un arbre peigne gauche, $n_1 = n - 1$, et l'équation précédente se résout alors en $C(n) = \Theta(n^2)$: la fonction `liste_postfixe` est donc en $\Omega(n^2)$. De plus, même pour des arbres parfaitement équilibrés (par exemple des arbres complets), on retrouverait l'équation caractéristique du tri fusion, et donc du $\Theta(n \log n)$.
Nous allons implémenter la fonction `liste_postfixe` en $\mathcal{O}(n)$. Pour cela, il faut donc procéder **sans concaténation**. Nous allons utiliser une fonction auxiliaire avec accumulateur. Pour mieux comprendre la méthode à implémenter, on commence par proposer une manière alternative de coder la fonction `taille`.

Q7. 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+1) in
          taille_add d ng
```

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

Q8. En s'inspirant de ce schéma, écrire une fonction `liste_postfixe` sans utiliser l'opérateur de concaténation `@`. On pourra passer par une fonction auxiliaire:

```
(* Renvoie la liste des éléments de a dans l'ordre postfixe *)
let liste_postfixe (a: 'a ab) : 'a list =
  (* Renvoie (liste_postfixe a) @ l *)
  let rec postfixe_concat (a: 'a ab) (l: 'a list) : 'a list =
    ...
```

Q9. (**Optionnel**) Faire de même avec `liste_infixe` et `liste_prefixe`.

Q10. (**Optionnel**) Implémenter une fonction permettant de reconstruire un arbre à partir de ses parcours infixes et préfixes. On pourra implémenter la fonction auxiliaire suivante:

```
(* Si pre et inf sont deux listes avec
   - inf le parcours infixe d'un arbre a
   - pre une liste commenÃ§ant par le parcours préfixe de a
   alors reconstruire pre inf renvoie le couple (a, l) avec
   l la liste des valeurs de pre non utilisées dans a *)
let reconstruire (pre: 'a list) (inf: 'a list) : ('a ab * 'a list) = ...
```

Q11. Écrire une fonction `tree_map: ('a -> 'b) -> 'a ab -> 'b ab` qui applique une fonction f sur chaque noeud d'un arbre a.

Q12. Écrire une fonction `tree_sum: int ab -> int` calculant la somme des entiers contenus dans un arbre binaire.

Q13. Écrire une fonction `appartient : 'a -> 'a ab -> bool` déterminant si un élément se trouve dans un arbre binaire.

Q14. (**Optionnel**) Déterminer un schéma commun dans les trois dernières fonctions, et proposer une fonction `tree_fold` permettant de les généraliser, tout comme `fold_left` 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:

```
type ('k, 'v) dico = ('k * 'v) ab
```

Q15. (**Optionnel**) Écrire une fonction `assoc_opt : 'k -> ('k, 'v) dico -> 'v option` prenant en entrée un élément k et un dictionnaire d 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.

Q16. (**Optionnel**) Écrire une fonction `assoc_all : 'k -> ('k, 'v) dico -> '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 (optionnel): Arbres stricts

On considère, pour A, B deux ensembles, les arbres binaires stricts dont les noeuds internes sont étiquetés par A, et les feuilles par B:

```
type ('a, 'b) abs = F of 'a | N of 'b * ('a, 'b) abs * ('a, 'b) abs
```

Par exemple, on peut considérer les expressions arithmétiques:

```
type operateur = Plus | Fois | Moins | Div
type arith = (operateur, int) abs

(* (1+2) * (3-4) *)
let e = N(Fois,
  N(Plus, F 1, F 2),
  N(Moins, F 3, F 4)
)
```

Nous avons vu que pour les arbres binaires quelconques, le parcours postfixe seul ne permet pas de déterminer exactement l'arbre. Pour les arbres stricts dont les feuilles et les noeuds vivent dans des ensembles disjoints, l'opération de parcours postfixe est injective, et donc réversible. On souhaite écrire une fonction qui, étant donné une liste d'étiquettes, construit un arbre dont le parcours postfixe est la liste en entrée. Afin de pouvoir considérer des listes pouvant contenir deux types distincts, on crée un nouveau type union:

```
type ('a, 'b) union = A of 'a | B of 'b

let (l: (int, string) union list) =
  [A 5; B "bla"; A 4; A 3; B "toto"]
```

L'objectif de cet exercice est donc d'écrire une fonction ayant la spécification suivante:

```
(* construire_post l renvoie Some a avec a un arbre dont le parcours
   postfixe donne l. Renvoie None si l n'est pas un parcours postfixe
   valide. *)
construire_post: ('a, 'b) union list -> ('a, 'b) abs option
```

Q1. Implémenter la fonction auxilaire suivante:

```
(* construire_post_arbres l_etiq pile construit des arbres à partir de la liste
   l_etiq d'étiquettes. La liste `pile` est la pile des arbres déjà construits:
   la lecture d'une étiquette interne permet de fusionner les deux arbres au
   sommet de la pile, tandis que la lecture d'une étiquette feuille crée un
   nouvel arbre sur la pile. La liste renvoyée est la pile des éléments
   restants après toutes les opérations *)
construire_post_arbres: ('a, 'b) union list -> ('a, 'b) abs list -> ('a, 'b) abs list}
```

Q2. En déduire la fonction `construire_post`, et la tester en l'utilisant pour lire quelques expressions arithmétiques en notation postfixe.

Exercice 3: Arbres généraux

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: `List.map`, `List.fold_left`, `List.filter`, etc... Sur la machine virtuelle, le logiciel **Zeal** vous donne accès à la documentation de ces fonctions (et de toute la librairie standard OCaml). Si vous n'êtes pas sur la machine virtuelle, cette documentation est trouvable en ligne ici: v2.ocaml.org/api/List.html.

Il y a deux manières simples 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(x, 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.

Q1. Implémenter une fonction `hauteur` qui calcule la hauteur d'un arbre

Q2. 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.

Q3. Écrire les fonctions `liste_prefixe` et `liste_postfixe` de parcours préfixe et postfixe.

On rappelle que le **parcours en largeur** énumère les noeuds d'un arbre par ordre croissant de profondeur. Il est plus naturel de décrire ces parcours de manière itérative, avec une boucle et une file:

Algorithme 1 : Parcours en largeur

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);

Nous allons transformer la boucle tant que de l'algorithme en une fonction auxiliaire récursive. Commençons par implémenter une structure de file en OCaml. Nous avons remarqué que les listes OCaml se comportent comme des piles (dernier arrivé, premier sorti). On peut implémenter une file avec deux piles, comme suit:

- La file est constituée d'une pile tête et d'une pile queue;
- Pour enfiler un élément, on l'empile sur la pile queue;
- Pour défiler un élément, on le défile depuis la pile tête. Si celle-ci est vide, on commence par **renverser** la pile queue dans la pile tête.

Q4. Sur feuille, représenter l'état d'une file F, initialement vide, après chacune des opérations suivantes. On représentera d'une part F sous forme de file abstraite, et d'autre part avec les deux piles décrites ci-dessus:

```
enfiler 1; enfiler 2; defiler; enfiler 3; enfiler 4;
defiler; enfiler 5; defiler; defiler
```

Q5. On pose `type 'a file = 'a list * 'a list`. Implémenter les 4 opérations de file:

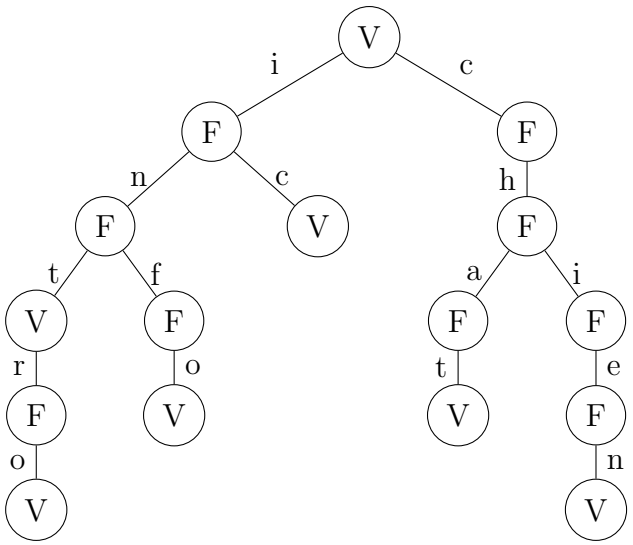
```
file_vide: unit -> 'a file

enfiler: 'a file -> 'a -> 'a file

defiler: 'a file -> 'a * 'a file (* couple (élément défilé, file restante) *)

est_vide: 'a file -> bool
```

Q6. Implémenter le parcours en largeur, utilisant la fonction auxiliaire suggérée ci-dessous:

```
(* Renvoie la liste des étiquettes de a dans l'ordre du parcours en largeur *)
let liste_largeur (a: 'a tree) =
  (* Applique la boucle du parcours en largeur de a, à partir
     de la file de noeuds f. Renvoie la liste des éléments visités. *)
  let rec liste_largeur_file (f: 'a tree file) : 'a list =
    ...
  in
  let file_depart = enfiler (file_vide ()) a in
  liste_largeur_file file_depart
```

Exercice 4 (optionnel): Arbres préfixes

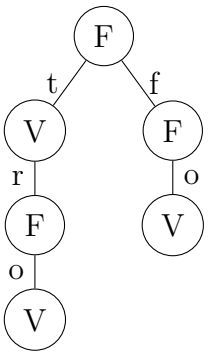
Les arbres préfixes servent à stocker des ensembles de mots. L'idée est de stocker sur chaque arête entre un noeud et son parent une lettre. Un noeud stockera un booléen, et représentera le mot formé de toutes les lettres entre la racine et lui. Par exemple:



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:

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

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




```
let t =
Node(false, [
  ('t', Node(true, [
    ('r', Node(false, [
      ('o', Node(true, []))
    ]))
  ]));
  ('f', Node(false, [
    ('o', Node(true, []))
  ]))
])
```

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

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

Q1. Écrire une fonction `taille` qui calcule le nombre de mots contenus dans un arbre préfixe.

Q2. Écrire une fonction `plus_long` qui calcule la longueur du plus long mot contenu dans un arbre préfixe.

Q3. Écrire une fonction `rechercher: string -> pre_tree -> bool` qui détermine si un mot est dans un arbre préfixe.

Q4. Écrire une fonction `ajouter: string -> pre_tree -> pre_tree` qui ajoute un mot à un arbre préfixe.

Q5. Écrire une fonction `enumerer` qui renvoie la liste des mots contenus dans un arbre préfixe, par ordre alphabétique.

Q6. 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.

Q7. 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.

Q8. 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.

Q9. Écrire une fonction `fusion` permettant de faire l'union de deux arbres préfixes. Attention à ne pas créer de doublons.