TP n°10 - Arbres généraux et ABR en Ocaml

Notions abordées

- Définition d'un type pour les arbres généraux (*i.e.* d'arité quelconque)
- Fonctions sur les arbres généraux avec deux fonctions mutuellement inductives
- Fonctions sur les arbres généraux avec `List.fold_left`
- Parcours par niveau d'un arbre général
- Transformation d'un arbre général en arbre binaire
- Insertion et suppression dans un ABR
- Exceptions : définition, avec ou sans paramètres, levée, rattrapage

Toutes les fonctions doivent être commentées et **testées**. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

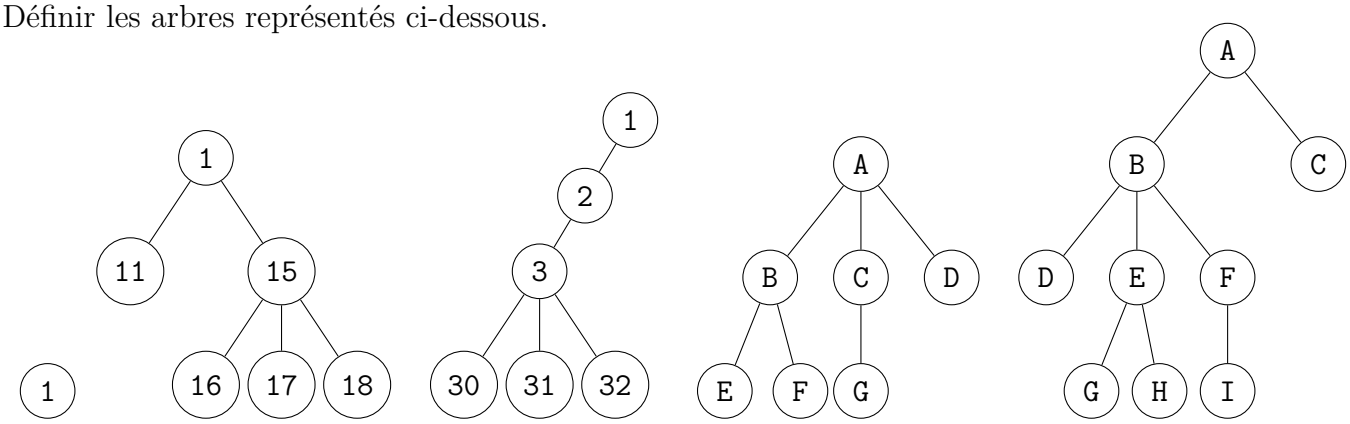
Arbres généraux

Exercice 1 S'approprier ce nouveau type d'arbre

Dans cet exercice on s'intéresse à des arbres généraux étiquetés, c'est-à-dire des arbres vides ou composés de nœuds d'arités quelconques. Pour un ensemble d'étiquettes $S \neq \emptyset$, on peut décrire formellement cet ensemble comme construit par induction à partir des règles de construction $V|^0_{\{\_\}}$ et $N|^n_S$ pour tout $n \in \mathbb{N}^*$.
Puisqu'il n'est pas possible d'implanter une infinité de constructeurs dans un type somme en OCaml, on utilisera le type suivant pour des arbres généraux étiquetés par des valeurs de type `'a`.

```
type 'a arbre_g =
  | Vide
  | Nd of 'a * ('a arbre_g) list
```

Question 1
Définir les arbres représentés ci-dessous.



Question 2 💻
Définir une fonction `est_vide` qui teste si un arbre général est vide.

Question 3 💻
Définir une fonction `est_feuille` qui teste si un arbre général est réduit à une feuille, c'est-à-dire un nœud ayant aucun enfant, ou uniquement l'arbre vide comme enfant. *On pourra d'abord créer une fonction* `est_plein_de_vide` qui teste si tous les éléments d'une liste d'arbres généraux sont des arbres vides.

Exercice 2 Fonctions récursives sur les arbres généraux

Question 1 ✍
Définir par induction sur l'ensemble des arbres généraux la fonction mathématique h qui donne la hauteur. *On pourra s'inspirer de la définition donnée en cours pour les arbres binaires.*

Question 2 💻
Définir une fonction `hauteur` qui calcule la hauteur d'un arbre général. *On n'attend pas une fonction récursive terminale. Il est conseillé de faire ici deux fonctions mutuellement récursives : l'une qui gère les arbres, et l'autre qui gère les listes d'arbre. La première appelle la seconde pour traiter les enfants d'un nœud, tandis que la seconde appelle la première pour chacun de ses éléments.*

Question 3 ✍
Définir par induction sur l'ensemble des arbres généraux la fonction mathématique s qui donne la taille, c'est-à-dire le nombre de nœuds.

Question 4 💻
Définir une fonction `nb_noeuds` qui calcule la taille d'un arbre général.

Question 5 ✍
Définir par induction sur l'ensemble des arbres généraux la fonction f donnant le nombre de feuilles.

Question 6 💻
Définir une fonction `nb_feuilles` qui calcule le nombre de feuilles d'un arbre général.
Attention, une feuille peut a priori avoir des enfants, seulement ils sont tous vides.

On rappelle que le module `List` du langage OCaml propose une fonction nommée `fold_left` de signature `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` qui calcule $f(\dots f(f(a, L_1), L_2), \dots L_n)$ à partir de la fonction f passée en premier argument, de l'élément a passé en deuxième argument et de la liste $L_1, L_2, \dots L_n$ passée en troisième argument.

Question 7 💻
À l'aide du schéma `List.fold_left`, proposer des fonctions `hauteur_bis`, `nb_noeuds_bis` et `nb_feuilles_bis` alternatives des fonctions précédemment proposées.

Exercice 3 Plus de fonctions récursives sur les arbres généraux

Cet exercice est un prolongement du précédent. Il n'introduit pas de nouvelles notions mais constitue un exercice d'entraînement. Il est donc conseiller de le passer dans un premier temps, notamment lors de la séance en classe, et d'y revenir plus tard, "pour repasser une couche". De plus, pour un entraînement complet, les fonctions demandées peuvent être implémentées par des fonctions mutuellement récursives d'une part, et grâce au schéma `List.fold_left` *d'autre part.*

Question 1
Définir une fonction `est_present` qui teste si un élément est présent dans un arbre général, plus précisément qui teste si une valeur d'étiquette est l'étiquette d'un des nœuds de l'arbre général.

Question 2
Définir une fonction `est_en_feuille` qui teste si un élément est présent en feuille dans un arbre général, plus précisément qui teste si une valeur d'étiquette est l'étiquette d'une des feuilles de l'arbre général.

Exercice 4 Parcours par niveau d'un arbre général

Cet exercice est lui aussi un exercice à passer en classe. Il permet de "repasser une couche" sur le parcours par niveau déjà vu sur les arbres binaires.

Question 1
Définir une fonction récursive terminale `affiche_par_niveaux` qui prend en argument un arbre général, ainsi qu'une fonction transformant une étiquette en chaîne de caractères, et qui affiche les étiquettes de tous les nœuds de l'arbre, niveau par niveau, et de gauche à droite pour chaque niveau.

Exercice 5 Transformer un arbre général en arbre binaire

Dans cet exercice on utilise à la fois des arbres généraux et des arbres binaires, le but étant de transformer un arbre général en un arbre binaire équivalent dans un sens à préciser.
Afin de ne pas avoir deux types différents avec les mêmes constructeurs `Vide` et `Nd`, on redéfinit le type des arbres binaires comme suit.

```
type 'a ab =
  | VideB
  | NdB of 'a* ('a ab) * ('a ab)
```

Définition 1

Soit S un ensemble d'étiquettes. Soit $a \in \mathcal{A}_G(S)$. Soit $b \in \mathcal{A}_B(S)$.
On dit que b est équivalent à a s'il existe une bijection des nœuds de a vers ceux de b telle que la racine de a est envoyée sur celle de b, et pour tout nœud n de a envoyé sur n' dans b, les descendants (resp. les frères) de n dans a sont envoyés sur les descendants droits (resp. gauches) de n' dans b.

Question 1 ✍
Dessiner les arbres binaires équivalents aux 5 arbres généraux représentés en début de TP (*cf.* p 1).

Question 2 💻
En vue de tests futurs définir les 5 arbres binaires précédents. *Au moins les trois premiers si vous n'avez pas le courage.*

Question 3 💻
Proposer une fonction `transfo` qui calcule l'arbre binaire équivalent à un arbre général. *On n'attend pas une fonction récursive terminale ici. C'est faisable mais assez lourd. Me demander des explication si ça vous intéresse.*

Complément sur les arbres binaires

Exercice 6 Arbres binaires de recherche - partie 2

On complète dans cet exercice les fonctions permettant de manipuler les arbres binaires de recherche (ABR). Commencer par recopier le code des fonctions de l'exercice sur les ABR du dernier TP.

Question 1
Définir une fonction récursive de signature `insere_f (a:'a abr) (elem :'a) : 'a abr` qui réalise l'insertion de l'élément **elem** en feuille dans l'ABR `a`. L'arbre calculé doit lui même être un ABR.

Question 2
Définir une fonction récursive de signature `insere_f (a:'a abr) (elem :'a) : 'a abr` qui réalise l'insertion de l'élément **elem** en feuille dans l'ABR `a`. L'arbre calculé doit lui même être un ABR.

Exercice 7 Chercher un élément satisfaisant

On a déjà vu qu'on pouvait déclencher une erreur qui renvoie un message grâce à la fonction `failwith`. En réalité, cette fonction relève d'un mécanisme plus général appelé levée d'exception. On définit une exception en précisant son nom, qui porte obligatoirement une majuscule, et le type de la valeur qu'elle encapsule, en suivant la syntaxe suivante : `exception MonException of mon_type`.
En fait c'est un peu comme si on ajoutait un constructeur au type `exn` : après une telle déclaration on peut définir une valeur de type `exn` à partir d'une expression `expr` de type `mon_type` par `MonException expr`.

Question 1
Définir une exception nommée `MaFailure` qui encapsule une valeur de type `string`. Interpréter ensuite des expressions comme `MaFailure "coucou"`.

L'intérêt du type exception réside dans la fonction `raise`, qui prend en argument une exception, et qui, lorsqu'elle est appelée stoppe l'évaluation et renvoie cette exception. En français, on dit qu'on lève une exception.

Question 2
Interpréter plusieurs appels à la fonction `raise` sur des exceptions construites avec `MaFailure`.

Question 3
Définir une fonction `mon_failwith` qui prend en argument une chaîne de caractères et qui lève l'exception construite avec `MaFailure` qui encapsule cette chaîne. Comparer cette fonction avec la fonction prédéfinie `failwith` (type, valeur, comportement pour différents appels...).

De même qu'une type somme peut faire apparaître des constructeurs qui n'ont aucun argument, on peut définir des exceptions qui n'encapsulent rien.

Question 4
Définir une exception sans paramètre nommée `PasTrouve`.

Question 5 🖳
Définir une fonction `cherche_elem` qui prend en argument une liste et un élément de type compatible, et qui cherche cet élément dans la liste. Plus précisément, cette fonction retourne un indice où l'élément est présent dans la liste s'il en existe, et qui lève l'exception `PasTrouve` sinon. *Préciser en commentaire dans le jeu de tests des cas où l'exception est levée.*

Question 6 🖳
Définir une fonction `elem_suivant` qui prend en argument une liste `l` et un élément `e` de type compatible, et qui cherche `e` dans `l` pour retourner l'élément suivant. Plus précisément, cette fonction retourne l'élément suivant la première occurrence de `e` dans `l` si c'est possible, et lève une exception pertinente dans les autres cas, c'est-à-dire dans le cas où `e` n'apparaît pas dans `l` et dans le cas où il apparaît en dernière position. *Préciser en commentaire dans le jeu de tests des cas où une exception est levée et laquelle.*

Les exceptions levées avec `raise` peuvent être rattrapées, c'est-à-dire qu'on indique une valeur de remplacement pour ne pas être bloqué avec une exception. Cette valeur de remplacement peut dépendre de l'exception, de son constructeur et, le cas échéant, de la valeur qu'elle encapsule. En cela le mécanisme de rattrapage ressemble au filtrage. Par exemple, l'expression

```
try elem_suivant [1;2] 5 with
| PasTrouve -> 18
| EstDernier -> 56
```

est évaluée à 18, car l'appel `elem_suivant [1;2]` lève l'exception `PasTrouve`, qui est ensuite rattrapée dans le premier cas du `try`, tandis que l'expression

```
try elem_suivant [1;2;5] 5 with
| PasTrouve -> 18
| EstDernier -> 56
```

est évaluée à 56, car l'appel `elem_suivant [1;2]` lève l'exception `EstDernier`, qui est ensuite rattrapée dans le second cas du `try`. Enfin, l'expression

```
try elem_suivant [1;2;5;6] 5 with
| PasTrouve -> 18
| EstDernier -> 56
```

est évaluée à 6, car l'appel `elem_suivant [1;2]` vaut 6, aucune exception n'est levée, le `try` est simplement ignoré.

Question 7 🖳
Définir une fonction récursive de signature `cherche_feuille_pte (a:'a ab) (pte:'a->bool): 'a` qui calcule l'étiquette d'une feuille de `a` satisfaisant `pte` (*i.e.* dont la valeur selon `pte` est `true`) s'il existe une telle feuille, et qui lève l'exception PasTrouve sinon. *On pourra s'engager dans une recherche dans le sous-arbre gauche, et ne s'attaquer au sous-arbre gauche que si cette recherche est infructueuse. L'idée est de faire ça grâce à un rattrapage d'exception.*