Feuille d'exercices n°5

Pour représenter les arbres généraux, on utilise le type suivant:

```
type 'a gtree =
    Empty
  | Node of ('a * ('a gtree) list)
```

Exercice I :

Q1 – Définir la fonction `size : 'a gtree -> int` qui calcule la taille (nombre de nœuds) de son argument.

Utilisez d'abord une double récurrence, puis l'itérateur `List.fold_left`.

Q2 – Définir la fonction `height : 'a gtree -> int` qui calcule la hauteur de son argument.

Q3 – Définir une fonction `to_list : 'a gtree -> 'a list` qui donne la liste des étiquette de son argument. Peu importe l'ordre dans lequel les étiquettes sont rangées dans la liste; ce qui importe est qu'elles y soient toutes.

Exercice II : Les feuilles d'un arbre général

Sur les arbres binaires, on n'avait qu'une seule notion de *feuille*: les arbres de la forme `Node(x,Empty,Empty)`. Avec les arbres généraux, la chose est plus complexe: les arbres de la forme `Node(x,[]`, ou `Node(x,[Empty])`, ou `Node(x,[Empty;Empty])`, etc. peuvent être vus comme des *feuilles* car il n'a y aucune autre étiquette que `x` dans de tels arbres.

On appelle *forêt vide* les listes d'arbres de la forme `[]`, `[Empty]`, `[Empty; Empty]`, etc. On appelle donc `feuille` un arbre non vide, de la forme `Node(x,gts)` où `gts` est une *forêt vide*.

Q1 – Définir la fonction `empty_forest : ('a gtree) list -> bool` qui donne `true` si et seulement si son argument est une forêt vide.

Vous *pouvez* utiliser l'itérateur `List.for_all`, ou non.

Q2 – Définir la fonction `nb_leaves : 'a gtree -> int` qui calcule le nombre de feuilles de son argument.

Q3 – Définir la fonction `leaves : 'a gtree -> 'a list` qui donne la liste de toutes les feuilles de son argument.

Exercice III : Normalisation des arbres généraux non vides

Q1 – Définir la fonction `remove_Empty : 'a gtree -> 'a gtree` qui donne l'arbre obtenu en supprimant les (sous)arbres vides. On suppose que l'arbre donné «au départ» est non vide et `(remove_Empty Empty)` déclenche l'exception `(Invalid_argument "remove_Empty")`.

Indication: c'est dans les forêts que cela se passe.

Q2 – Pour tester la fonction précédente, définissez la fonction `not_exists_Empty: 'a gtree -> bool` qui vaut `true` si et seulement si son argument ne contient aucun constructeur `Empty`.

Vous *pouvez* utiliser l'itérateur `List.for_all`.

Exercice IV : Autre arbre général

On définit un type pour les arbres généraux non vides de la manière suivante:

```
type 'a gtree2 =
    Leaf of 'a
  | Branch of 'a * ('a gtree2) list
```

Q1 – Définissez la fonction `to_2 : 'a gtree -> 'a gtree2` qui transforme un arbre de type `'a gtree` que l'on suppose non vide en un arbre de type `'a gtree2`. On aura que `(to_2 Empty)` déclenche l'exception `(Invalid_argument "to_2")`.

Si `gt` est une *feuille* (au sens de l'exercice 2) alors `(to_2 gt)` vaut `(Leaf x)` où `x` est l'étiquette de `gt`. Par exemple: `(to_2 (Node(x,[Empty;Empty])))` vaut `(Leaf x)`.

Exercice V : Recherche dans un arbre général

Q1 – Définir la fonction `find : 'a -> 'a gtree -> 'a gtree` telle que `(find x0 gt)` donne un sous-arbre de `gt` dont l'étiquette est égale à `x0`. L'application `(find x0 gt)` déclenche l'exception `Not_found` si `x0` n'apparaît pas dans `gt`. Si plusieurs sous-arbres de `gt` portent l'étiquette `x0`, on prend la première qui se présente.

Vous utiliserez la construction `try-with` pour rechercher dans une forêt.

Travaux sur machine

Nous allons utiliser les arbres généraux pour étudier le jeu du *tictactoe* (ou jeu du *morpion*). Si vous ne connaissez pas le jeu, la requête `règles tic tac toe` sur un moteur de recherche vous donnera toutes les informations utiles.

Vous trouverez dans le fichier `/Infos/lmd/2019/licence/ue/LU2IN019-2019oct/tictactoe.ml` une implantation des fonctions de gestion du jeu. Les deux joueurs sont représentés par les valeurs `X` et `O` du type `player`. Le type `game` représente un état de la grille de jeu.

Ne lisez pas la totalité des définitions du fichier `tictactoe.ml`. Comprenez et, éventuellement, testez les fonctions qui vous seront utiles pour réaliser ce TME. À savoir:

- `new_game : int -> game` donne l'état initial d'un jeu. L'entier en argument est la taille de la grille qui est toujours carrée (nombre de lignes = nombre de colonnes).

  Remarques: le jeu standard se joue avec une grille de 3×3. Dans ce qui suit, nous vous déconseillons d'utiliser des grilles de taille supérieure à 3.

- `write : (int*int) -> player -> game -> game` tel que `(write (i,j) p m)` donne l'état du jeu après que le joueur `p` a posé sa marque en `(i,j)` sur la grille donnée par l'état `m`. En termes plus simples: `p` joue en `(i,j)`.

- `next_player : player -> player` permet de passer d'un joueur à l'autre: `(nex_player X)` vaut `O` et `(next_player O)` vaut `X`.

- `free_cells : game -> (int*int) list` donne la liste des cellules (positions ou cases) libres du jeu `m`. Ce sont les positions où l'on peut jouer.

- `winning_game : game -> bool` telle que `(winning_game m)` vaut `true` si (et seulement si) l'état du jeu `m` est gagnant pour l'un ou l'autre des joueur.

- `winning : game -> player -> bool` telle que `(winning m p)` vaut `true` si et seulement si l'état du jeu `m` est gagnant pour `p`.

- `null_game : game -> bool` qui vaut `true` si est seulement si l'état de jeu `m` est une grille pleine et aucun des joueur n'est gagnant: la partie est nulle.

On sait qu'à ce jeu, le premier joueur ne peut pas perdre: soit il gagne (si son adversaire est inattentif), soit la partie est nulle. Cela ne signifie pas qu'il n'existe pas de partie où le second joeur n'a pas la possibilité de gagner (si le premier est très inattentif). Cela signifie qu'à chaque tour le premier joueur a la possibilité de jouer un coup qui empêchera le second de gagner. Le but de cette séance est de définir une fonction qui vérifie cela.

> **Conseil:** les algorithmes et fonctions de ce TME ne sont pas toujours faciles. Si vous ne vous sentez pas d'attaque, implantez et testez soigneusement les fonctions vues en TD; sinon, tentez l'aventure.

Exercice VI : Unique

Q1 – Définir la fonction `all_games_tree : game -> player -> game tree` telle que `(all_games_tree m p)` calcule l'arbre de toutes les parties possibles en partant de l'état de jeu `m` lorsque c'est au joueur `p` de jouer.

Il faut penser à faire alterner les joueurs et que l'on joue sur les cases libres. Ici, chaque joueur jouera sur toutes les cases libres pour construire l'arbre des toutes les parties. La construction s'arrête lorsqu'il n'y a pas de case libre (voir fonction `free_cells`).

L'application `(all_games_tree (new_game n) X)` donne l'arbre de toutes les parties de tictactoe lorsque `X` commence avec une grille de taille $n \times n$.

Attention: c'est un peu long à calculer pour une grille de taille 3 en partant de l'état initial. Nous n'avons pas obtenu de réponse pour une grille de taille 4...

Tests: il n'est pas facile de tester la correstion d'une telle fonction. Vous pouvez tester votre fonction en partant d'un état non vide du jeu. Par exemple:

```
let m = (write (2,2) X
           (write (2,1) O
              (write (0,1) X
                 (write (0,0) O
                    (write (1,1) X
                       (new_game 3))))))
```

qui donne la grille

O	X	
	X	
	O	X

Pour `(all_games_tree (new_game n) X)` vous devrez vérifier que

```
let at = all_games_tree (new_game 3) X
let _ = assert (size at = 986410)
let _ = assert (nb_leaves at = 362880)
let _ = assert (not_exists_Empty at)
```

(les fonctions `size`, `nb_leaves` et `not_exists_Empty` sont celles vue en TD)

Vous aurez noté que l'arbre construit ne contient pas d'arbre vide. Les feuilles de l'arbre construit ont toutes la forme `Node(m,[])`.

Q2 – L'arbre construit à la question précédente est inutilement trop gros. On peut arrêter la construction dès que l'on obtient une partie gagnante (voir fonction `winning_game`).

Définir la fonction `winning_games_tree : game -> player -> game tree` telle que `(winning_games_tree m p)` construit l'arbre des parties possibles en partant de l'état de jeu `m`, lorsque c'est à `p` de jouer. Les feuilles de l'arbre obtenu sont de la forme `Node(m,[])` où `m` est gagnant (pour `X` ou `O`) ou nul.

Vous devrez vérifier que:

```
let wt = winning_games_tree (new_game 3) X
let _ = assert (size wt = 549946)
let _ = assert (nb_leaves wt = 255168)
let _ = assert (not_exists_Empty wt)
```

Q3 – À partir d'un arbre construit avec `winning_games_tree` on construit un arbre (général) non vide dont les constructeurs donnent de l'information sur l'état du jeu. C'est un type d'arbre spécifique au jeu du tictactoe. Voici le type proposé:

```
type ttt_tree =
    Null of game
  | Xwin of game
  | Owin of game
  | XNode of game * ttt_tree list
  | ONode of game * ttt_tree list
```

Dans un tel arbre:

- `(Null m)` est une feuille pour un jeu nul.
- `(Xwin m)` est une feuille pour un jeu gagnant pour `X`.
- `(Owin m)` est une feuille pour un jeu gagnant pour `O`.
- `(XNode(m,gts))` est un nœud joué par `X`.
- `(ONode(m,gts))` est un nœud joué par `O`.

Définir la fonction `ttt_of_gtree : player -> game gtree -> ttt_tree` qui transforme un arbre de partie `game gtree` en un arbre de type `ttt_tree`.

Pour que la transformation soit correcte, il faut être cohérent entre l'argument de type `player` passé à la fonction de transformation et celui passé à la construction de l'arbre général `game gtree`: si vous avez construit un arbre `gt:game gtree` avec `(winning_games_tree m X)`, il faut le transformer avec `(tt_of_gtree X gt)`.

Q4 – Définir la fonction `stat : ttt_tree -> (int*int*int)` telle que `(stat t)` donne un triplet composé (dans cet ordre) du nombre de feuilles gagnantes pour `X`, du nombre de feuilles gagnantes pour `O` et du nombre de feuilles à jeu nul.

Vous devrez vérifier que

```
let tt = ttt_of_gtree X wt
let _ = assert (stat tt = (131184, 77904, 46080))
let _ = assert (131184+77904+46080 = nb_leaves wt)
```

Stratégie non perdante pour `X` Pour s'assurer qu'il existe une startégie «non perdante» pour `X`, il faut s'assurer que pour chaque coup jouable par `O`, il existe une réponse «non perdante» de `X`.

Q5 – Définir la fonction `not_loosing_X : ttt_tree -> bool` qui vaut `true` si son argument est un arbre de parties «non perdantes» pour `X`, en supposant que `X` est le premier joueur.

Algorithme pour `(not_loosing_X t)`:

- si `t` est une feuille `(Owin m)`: la réponse est `false`.
- si `t` est une feuille `(Xwin m)` ou `(Null m)`: la réponse est `true`.
- si `t` est de la forme `(XNode(m,ts))`: il faut vérifier qu'il existe un sous-arbre dans `ts` «non perdant» pour `X`.
- si `t` est de la forme `(ONode(m,ts))`: il faut vérifier que pour tous les arbres de `ts` sont «non perdants» pour `X`.

On peut utiliser les itérateurs `List.exists` et `List.for_all`.

Vous vérifierez que

```
let xt = remove_Owin tt
let _ = assert (not_loosing_X xt)
```

On a ainsi obtenu dans un arbre de parties où `X` commence, que quelque soit le coup joué par `O`, il existe une réponse de `X` qui contient des feuilles non perdantes (gagnante ou nulle) pour `X`.