Licence STS mention Informatique, semestre 5 2018–2019

Programmation fonctionnelle (Info 311)

TP3

1 Le jeu du morpion

On s'intéresse dans cette partie à la programmation du jeu du Morpion. Ce jeu se joue à deux joueurs. Chaque joueur choisit un symbole différent, habituellement une croix ou un cercle. Puis, chacun leur tour, ils écrivent leur symbole dans une case vide d'une grille de taille 3×3. Le joueur qui gagne est celui qui a réussi à aligner trois de ses symboles (horizontalement, verticalement on en diagonale). Voici un exemple d'une partie de Morpion où le joueur jouant les croix gagne :

X		O
X	O	
X		O

On représente la grille du morpion avec les marques à l'aide des types OCaml suivants :

```
type marque = { c : bool; p: int * int}
type grille = marque list
```

Les marques sont représentées par des enregistrements de type `marque` contenant deux champs. Le champ `c` de type `bool` représente la marque : la valeur `true` signifie qu'il s'agit d'une croix, et `false` un cercle. Le champ `p` contient la position de la marque. Ces positions sont des paires d'entiers $(colonne, ligne)$ avec une numérotation représentée ci-dessous :

	0	1	2
0			
1			
2			

Question 1.1 Donner une valeur OCaml de type `grille` pour représenter les grilles ci-dessous.

X		O
X	O	
X		O

	X	O
X	O	
O	X	

O	X	X
X	O	O
O	X	O

On commence par déterminer si une valeur de type `grille` représente bien une grille possible pour le jeu du Morpion 3×3. Cela signifie, en particulier, que dans une liste des marques la position de chaque marque doit respecter les bornes du jeu $\{(0,0),(0,1),...(2,2),...,(3,2),(3,3)\}$, et cette position doit être différente de celle de toutes les autres.

Question 1.2 En utilisant l'itérateur `List.for_all` dont le type est rappelé à la fin de ce sujet, écrire une fonction `dans_les_bornes: marque list -> bool` qui renvoie vrai si dans la liste passée en argument toutes les marques respectent les bornes du jeu.

Question 1.3 En utilisant l'itérateur `List.exists`, dont le type est rappelé à la fin de ce sujet, écrire une fonction `existe_symbole`, de type `grille -> int -> int -> bool`, telle que `existe_symbole g i j` détermine s'il existe une marque (peu importe le symbole) à la position `(i,j)` dans la grille `g`.

Question 1.4 En utilisant la fonction précédente, écrire une fonction récursive `sans_doublons: marque list -> bool` qui renvoie vrai si dans la liste passée en argument toutes les marques ont des positions distinctes.

Enfin, on suppose que dans toutes les parties on débute en écrivant une croix sur la grille. On s'intéressera donc pour répondre à la question suivante aux nombres de croix et de cercles dans la grille.

Question 1.5 En utilisant un itérateur, écrire une fonction `compter` de type `grille -> int * int` qui renvoie une paire d'entiers contenant le nombre de croix et le nombre de cercles dans une grille.

Question 1.6 En utilisant les fonctions précédentes, écrire une fonction `bonne_grille`, de type `grille -> bool`, qui détermine si une valeur de type `grille` représente bien une grille possible pour le jeu du Morpion.

Question 1.7 En utilisant la fonction précédente, écrire une fonction `bonne_grille_exn`, de type `grille -> unit`, qui renvoie `()` si pour une liste des marques `l` `bonne_grille l` renvoie `true`, et qui sinon lève une exception `Grille_invalide` qu'on suppose avoir défini précédemment.

Maintenant, on va s'intéresser à déterminer si les croix ou les cercles ont gagné dans une partie donnée du jeu de Morpion.

Question 1.8 On suppose ici qu'une grille ne contienne que des marques d'un même symbole. En utilisant la fonction précédente `existe_symbole`, écrire une fonction `gagne : grille -> bool`, qui détermine si une grille contient trois symboles alignés (horizontalement, verticalement, ou en diagonale).

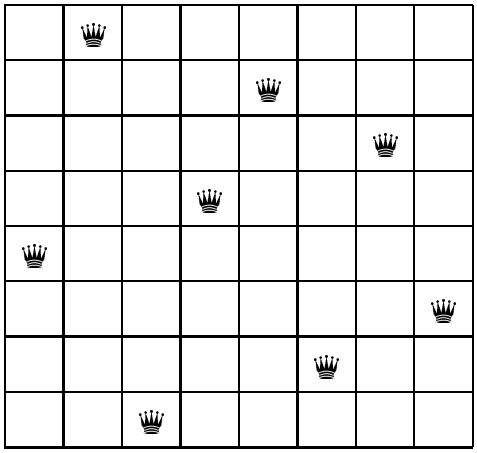
Question 1.9 En utilisant l'itérateur `List.filter`, dont le type est rappelé à la fin de ce sujet, écrire une fonction `extraire : grille -> bool -> grille`, telle que `extraire g s` renvoie une nouvelle grille ne contenant que les marques `s` d'une grille `g`.

Question 1.10 En utilisant les fonctions précédentes, écrire une fonction `qui_gagne : grille -> unit`, qui vérifie d'abord que la liste des marques est bien une grille valide. Si la grille n'est pas valide, alors la fonction doit rattraper l'exception levée par `bonne_grille_exn` et afficher à l'écran le message d'erreur. Si la grille est valide, alors la fonction doit afficher à l'écran qui des croix ou des cercles à gagner dans une grille (ou si la partie est nulle).

Question 1.11 (Question optionnelle, à ne faire que si vous avez répondu à toutes les autres questions). Quelle est une autre condition nécessaire, que l'on n'a pas mentionnée, pour qu'une liste des marques soit une grille valide du jeu de Morpion ? Dessiner un exemple d'une partie qui illustre la nécessité de cette condition. Comment faut-il modifier le programme pour remédier ce problème ?

2 Le problème des n reines (9 points)

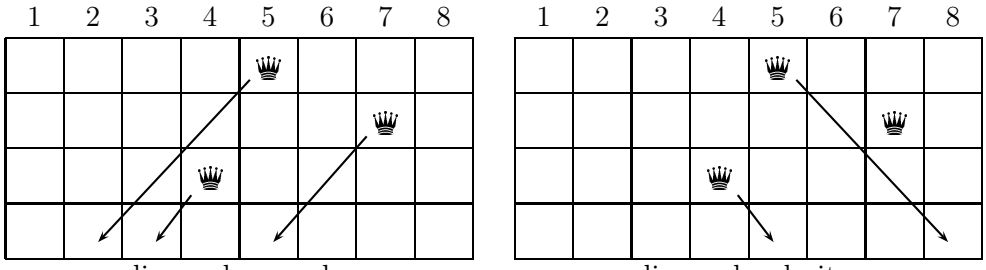
Le problème des n reines consiste à placer n reines sur un échiquier de taille $n \times n$ de telle sorte qu'aucune ne soit en prise : il ne faut donc pas plus d'une reine par ligne, par colonne et par diagonale. Voici un exemple pour $n = 8$.



Le but de cet exercice est d'écrire un programme en Ocaml qui permette de calculer le nombre de manières différentes de résoudre ce problème pour un nombre n quelconque de reines. Pour cela, nous allons utiliser un algorithme de recherche avec retour arrière (ou *backtracking* en anglais) qui va remplir les lignes de l'échiquier une à une. Afin de remplir une ligne, l'algorithme maintient 3 ensembles :

- a : contient les numéros des colonnes où il n'y a encore aucune reine de placée ;
- b : contient les numéros des colonnes sur lesquelles il n'est pas possible de placer une reine car ces positions se trouvent sur la diagonale *gauche* d'une reine déjà placée ;
- c : joue le même rôle que b mais pour les diagonales *droites*.

Par exemple, après avoir placé les 3 premières reines sur les colonnes 5, 7 et 4 (dans cet ordre) d'un échiquier 8×8, on a la situation suivante :



diagonales gauches diagonales droites

qui est représentée par les trois ensembles $a = \{1, 2, 3, 6, 8\}$, $b = \{2, 3, 5\}$ et $c = \{5, 8\}$. L'ensemble des colonnes où il est encore possible de placer une reine pour la 4^{e} ligne est donc tout simplement $(a \setminus b) \setminus c$, soit ici $\{1, 6\}$. L'algorithme consiste alors à essayer *une à une* ces positions : récursivement, on cherche les solutions pour lesquelles la 4^{e} reine est placée sur la colonne 1 puis, en revenant en arrière, récursivement celles pour lesquelles la reine est placée sur la colonne 6. À chaque appel récursif, les ensembles a, b et c sont modifiés en fonction de la colonne i choisie : i est supprimée de a et ajoutée aux ensembles b et c ; ces ensembles b et c sont alors mis à jour en décrémentant (resp. incrémentant) les éléments qu'ils contiennent afin de calculer les nouvelles diagonales des reines placées sur l'échiquier.

Dans la suite, nous allons simplement représenter les ensembles a, b et c par des listes d'entiers (`int list`).

Question 2.1 Écrire une fonction `list_of_int : int -> int list` telle que `list_of_int n` renvoie la liste `[1;2;...;n]` si `n` est positif et la liste vide sinon.

Question 2.2 Écrire les fonctions `succ_list : int list -> int list` et `pred_list : int list -> int list` telles que `succ_list [a1;a2;...;an]` renvoie la liste `[a1+1;a2+1;...;an+1]` et `pred_list [a1;a2;...;an]` renvoie la liste `[a1-1;a2-1;...;an-1]`.

Question 2.3 En utilisant un itérateur sur les listes, écrire la fonction `diff : 'a list -> 'a list -> 'a list` telle que `diff l1 l2` renvoie une liste contenant les éléments de `l1` qui ne sont pas éléments de `l2`.

Question 2.4 Écrire une fonction `remove : 'a list -> 'a -> 'a list` telle que `remove l x` renvoie une liste contenant les éléments de `l` sauf `x`.

La question suivante est l'algorithme principal du problème des n reines. Il s'agit d'écrire une fonction récursive prenant en arguments les 3 ensembles a, b et c décrits précédemment. Cette fonction renvoie le nombre de solutions qui prolongent la solution partielle décrite par a, b et c. En particulier, elle renvoie l'entier 1 quand l'ensemble a est vide (puisqu'il n'y a plus de reines à placer c'est qu'une solution a été trouvée).

Question 2.5 À l'aide des fonctions précédentes, et en utilisant un itérateur sur les listes, écrire une fonction récursive `nb_solutions : int list -> int list -> int list -> int` telle que `nb_solutions a b c` renvoie le nombre de solutions correspondant aux ensembles `a`, `b` et `c` donnés en paramètres.

Enfin, la question suivante consiste à écrire la fonction principale du programme qui se contente d'appeler la fonction `nb_solutions` avec comme paramètres les ensembles $a = \{1, 2, \ldots, n\}$ et $b = c = \emptyset$.

Question 2.6 Écrire une fonction `reines : int -> int` telle que `reines n` renvoie le nombre de solutions au problème des `n` reines.

Rappels.

`List.iter : ('a -> unit) -> 'a list -> unit`

`List.iter f [a1; ...; an]` est `f a1; ... ; f an`

`List.map : ('a -> 'b) -> 'a list -> 'b list`

`List.map f [a1; ...; an]` est `[f a1; ... ; f an]`

`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

`List.fold_left f a [b1; ...; bn]` est `f (... (f (f a b1) b2) ...) bn`

`List.filter : ('a -> bool) -> 'a list -> 'a list`

`List.filter p l` renvoie tous les éléments de `l` qui satisfont le prédicat `p` .

`List.mem : 'a -> 'a list -> bool`

`List.mem x l` renvoie `true` si `x` est dans la liste `l` et `false` sinon .

`List.for_all : ('a -> bool) -> 'a list -> bool`

`List.for_all prop [a1; ...; an] = true ssi prop(a1) et ... et prop(an)`

`List.exists : ('a -> bool) -> 'a list -> bool`

`List.exists prop [a1; ...; an] = true` ssi `prop(a1) ou ... ou prop(an)`

`List.filter : ('a -> bool) -> 'a list -> 'a list`

`List.filter prop [a1; ...; an]` renvoie les éléments de `[a1; ...; an]` qui satisfont `prop`, c'est-à-dire tel que `prop ai = true`