Programmation fonctionnelle – TP

Démineur

Décembre 2018

L'objectif de ce TP est de programmer le célèbre jeu *Démineur*[1]. On rappelle que le but du jeu est de trouver toutes les mines cachées dans une grille de taille variable avec la seule information du nombre de mines autour des cases déjà découvertes. On peut poser des drapeaux sur les cases encore couvertes pour indiquer un endroit où l'on pense qu'il y a une mine.

Les notes de bas de page donnent des indices, qu'on n'est pas obligé de lire.

On pourra se référer à Hoogle[2] pour la documentation sur l'API standard.

Pour faciliter la mise au point, on pourra enfin utiliser la fonction `trace` pour afficher certaines informations lors de l'évaluation des fonctions.

1 Structures de données de base et affichage

Q1. Proposer un type algébrique `Cell` pour représenter une case de la grille. Une case peut-être couverte, découverte, ou sélectionnée (pour l'affichage plus tard). Ce type algébrique aura donc trois *constructeurs de données* : `Covered`, `Uncovered`, et `Selected`. Dans les deux premiers cas, il faut mémoriser le nombre[3] de mines autour. Quand la case est couverte, il faut de plus mémoriser si elle contient une mine, ou si elle est marquée par un drapeau ;

Q2. Proposer un type algébrique `Grid` pour représenter la grille. On pourra aussi appeler `Grid` le *constructeur de données* correspondant. Ce type contient uniquement une liste de listes de `Cell` ;

Q3. Écrire une instance de la classe de types `Show` pour `Cell`, afin d'afficher un caractère adéquat pour les différentes situations :

```
instance Show Cell where
    show Selected = ...
    show (Uncovered n) = ...
    ...
```

Q4. Écrire une instance de la classe de types `Show` pour `Grid`, afin d'afficher la grille[4].

1. `https://fr.wikipedia.org/wiki/Démineur_(jeu)`
2. `https://www.haskell.org/hoogle/`.
3. Dans tout le TP il sera suffisant et plus simple d'utiliser des `Int` plutôt que des `Integer`.
4. Indice : on pourra utiliser notamment les fonctions `unlines` et `concatMap`.

2 Créer la grille avec les mines

Q5. Il faut d'abord déterminer, étant donné un nombre `n` de mines souhaitées, une ensemble de `n` couples de coordonnées différents et choisis aléatoirement. Étant donné une valeur `g` de type `StdGen` (on verra comment l'obtenir plus tard), la fonction `randomRs (x, y) g` fournit une liste infinie d'entiers uniformément répartis dans l'intervalle `[x,y]`.

Pour assurer que nos couples sont tous différents, on va les mettre dans un `Set`. Pour cela, il faut ajouter au début du fichier :

```
import Data.Set (Set)           -- on n'importe que Set
import qualified Data.Set as S -- puis tout mais en qualifiant
```

Remarquons le mot-clé `qualified` qui indique que les noms des fonctions de `Data.Set` devront être préfixés par `S.` (afin d'éviter des ambiguités avec les fonctions sur les listes dont certaines ont le même nom) : p. ex., si `s` est de type `Set Int` et `x` de type `Int` alors `S.insert x s` ajoute l'élément `x` dans l'ensemble `s`.

Proposer une fonction `randSet` qui, pour un nombre `n` de mines donné, deux `StdGen` donnés, le nombre de lignes et le nombre de colonnes de la grille de jeu, donne un ensemble contenant exactement `n` couples de coordonnées [5] [6] [7].

Q6. Écrire une fonction `grid` qui, étant donnés, une hauteur, une largeur, et un ensemble de couples de coordonnées, produit une grille dans laquelle toutes les cases sont couvertes et sans drapeau et seules les cases dont les coordonnées sont dans l'ensemble contiennent une mine [8].

3 Calculer le nombre de mines dans le voisinage

Plutôt que de faire une fonction qui regarde autour de chaque case, on va faire une fonction globale qui calcule tous les voisinages d'un seul coup. L'idée est que si on déplace la grille dans chacune des huits directions, en comblant les trous avec des zéros, on obtient huit grilles dont la somme composante par composante donne les voisinages recherchés.

$$\begin{bmatrix} * & & * & \\ & & & * \\ & * & * & \\ & & * & \end{bmatrix} \xrightarrow{\text{mines}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \xrightarrow{\text{up}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Q7 Écrire une fonction `mineIndic` qui pour une `Cell` donnée renvoit 1 si elle contient une mine et 0 sinon [9] ;

Q8 Écrire une fonction `mines` qui pour une grille donnée renvoit une liste de listes d'entiers contenant des 1 aux emplacements des mines et des 0 partout ailleurs [10] ;

5. Indice : on pourra utiliser `zip` pour créer une liste de couples à partir de deux listes d'entiers.
6. Indice : on pourra utiliser la variante de `foldl'` appelée `scanl'` pour obtenir une liste infinie d'ensembles correspondant aux ajouts successifs des couples de la liste infinie.
7. Indice : on pourra utiliser `dropWhile` et `head` pour récupérer le premier ensemble suffisamment grand.
8. Indice : on pourra écrire `grid` à partir d'une liste en compréhension définie à partir des indices des cases.
9. Indice : on pourra distinguer les (deux) cas avec du *pattern matching*
10. Indice : pour appliquer une fonction à une liste, on utilise `map` ; pour une liste de listes, on « mappe » `map` sur chaque ligne.

Q9 Écrire une fonction `moveUp` qui déplace une liste de listes d'entiers « vers le haut », c'est-à-dire, supprime la première ligne et en insère une de zéros à la fin [11] ;

Q10 Écrire une fonction `moveDown` qui déplace une liste de listes d'entiers « vers le bas » [12] ;

Q11 Écrire les fonctions `moveLeft` et `moveRight` qui déplacent une liste de listes d'entiers « vers le gauche » et « vers la droite » respectivement [13] ;

Q12 À partir de la liste de trois éléments contenant la grille de 0 et de 1 initiale, et ses translations vers le haut et le bas, on peut obtenir les 6 autres directions en déplaçant chacun des éléments de cette liste vers la gauche et vers la droite.

Écrire une fonction `gridMoves` qui donne la liste des huit grilles correspondant aux huit déplacements possibles [14].

Q13 Écrire une fonction `matrixSum` qui fait la somme composante par composante de deux listes de listes d'entiers [15].

Q14 Écire une fonction `neighbourMap` qui donne, à partir d'une grille de jeu, la somme des huits listes de listes calculées par `gridMoves` [16] ;

Q15 Écrire une fonction `updateCell` qui étant donnée une `Cell` et un nombre de mines `n` produit une nouvelle `Cell` dans laquelle le nombre de mines mémorisé vaut `n` ;

Q16 Écrire une fonction `updateGrid` qui étant donnée une `Grid` et une liste de listes d'entiers représentant les nombre de mines autour de chaque case, produit une nouvelle `Grid` incorporant ces nombres de mines.

4 Découvrir une case

Q17 Étant donné un type `a` quelconque, écrire une fonction `applyi` qui prend en entrée une fonction `f :: a -> a`, un indice entier `i` et une liste `xs` de `a`, et produit une copie de `xs` dans laquelle `f` a été appliquée à l'élément d'indice `i` [17] ;

Q18 Étant donné un type `a` quelconque, écrire une fonction `applyij` qui prend en entrée une fonction `f :: a -> a`, deux indices entiers `i` et une liste `xss` de listes de `a`, et produit une copie de `xss` dans laquelle `f` a été appliquée à l'élément de coordonnées `(i,j)` ;

Q19 Lorsque l'on découvre une case, si elle n'a aucune mine autour d'elle, il faut découvrir récursivement toutes les cases autour.

Écrire une fonction `uncover` qui prend un couple de coordonnées `(i, j)` et une grille, et produit la grille obtenue en découvrant la case de coordonnées `(i, j)` [18] [19] ;

11. Indice : on pourra utiliser les fonctions `tail`, `replicate`, `length` et `head`.
12. Indice : on pourra utiliser la fonction `init`.
13. Indice : on pourra utiliser la fonction `transpose`.
14. Indice : utiliser `map` plutôt que d'énumérer les huit transformations.
15. Indice : `zipWith` combine deux listes composantes par composantes. Pour des matrices, il faut « zipper » les lignes avec `zipWith`.
16. Indice : on pourra utiliser la fonction `foldl'` pour combiner `matrixSum` sur toute la liste.
17. Indice : on pourra utiliser les fonction `splitAt`, `head`, et `tail`.
18. Indice : on pourra utiliser la fonction `const` pour définir une fonction constante.
19. Indice : pour cumuler les effets des découvrements des cases alentours, on pourra utiliser `foldl'` sur la liste en compréhension décrivant les coordonnées des voisins.

5 Boucle principale de jeu (*Read-Eval-Print Loop*, REPL)

Q20 Écrire une fonction `covIndic` qui renvoit 1 si une `Cell` est couverte et 0 sinon ;

Q21 Écrire une fonction `won` qui étant donnés une grille et le nombre de mines qu'elle contient, revoit un booléen indiquant si l'on a gagné, c'est-à-dire si toutes les cases ne contenant pas de mines ont été découvertes [20] ;

Q22 Écrire une fonction `toggleFlag` qui permet d'ajouter un drapeau sur une `Cell` (couverte) ou de le retirer s'il y en a déjà un ;

Q23 Compléter la fonction `loop` ci-dessous [21] :

```
loop :: Int -> Int -> Int -> Grid -> IO ()
loop i j n b@(Grid xs) -- le paramètre b se décompose en (Grid xs)
 | won n b = putStrLn "Victoire␣!"
 | otherwise = do
    -- affiche la grille avec la case i, j sélectionnée
    putStrLn $ show $ Grid $ applyij (const Selected) i j xs
    -- lit un caractère
    c <- getChar
    case c of
        'i'        -> loop (max (i - 1) 0) j n b -- bouge le curseur
                                                 -- vers le haut
        'k'        -> ... -- bouge le curseur vers le bas
        'j'        -> ... -- bouge le curseur vers la gauche
        'l'        -> ... -- bouge le curseur vers la droite
        'f'        -> ... -- pose ou enlève un drapeau sur la case i, j
        'u'        -> ... -- découvre la case i, j; BOUM ?
        otherwise -> loop i j n b -- ne fait rien
```

Q24 Compléter la fonction `main` ci-dessous et ajouter `import System.IO` :

```
main::IO ()
main = do
    -- désactive l'attente de la touche entrée pour l'acquisition
    hSetBuffering stdin NoBuffering
    -- désactive l'écho du caractère entré sur le terminal
    hSetEcho stdin False
    -- récupère deux StdGen pour la génération aléatoire
    g  <- newStdGen
    g' <- newStdGen
    -- nombre de mines, lignes, colonnes
    let nmines = 5
    let l = 7
    let c = 10
    ... -- créer la grille, ajouter les mines, mettre à jour les voisins

    loop ... -- démarrer la REPL
```

Q25 Jouer !

Q26 Écrire une fonction `maybeFindMine` qui, uniquement à partir de ce qui est découvert, donne les coordonnées d'une case couverte contenant une mine si possible, et `Nothing` sinon. Modifier la fonction `loop` pour placer le curseur sur une telle case, quand elle existe, sur appui de la touche `'t'` [22].

20. Indice : on pourra utiliser les fonctions `map` et `sum`.
21. Pour les utilisateurs de MS Windows, voir l'annexe à la fin de de sujet.
22. `'t'` comme « tricher ».

A Pour les malheureux utilisateurs de MS Windows

Pour ceux qui utilisent MS Windows, il semble qu'il y a un bug dans la fonction `getChar` de GHC. Le code suivant (à ajouter au début du fichier) pourrait aider, en remplaçant `getChar` par `getHiddenChar` dans la Question 23.

```
{-# LANGUAGE ForeignFunctionInterface #-}
import Data.Char
import Control.Monad (liftM)
import Foreign.C.Types

getHiddenChar = liftM (chr.fromEnum) c_getch
foreign import ccall unsafe "conio.h␣getch"
  c_getch :: IO CInt
```