

TP n°5 – Le morpion

Ce TP porte sur le jeu du morpion (connu aussi sous le nom Tic-tac-toe). On ne développera que la version humain versus humain.

Le jeu se présente sous la forme d'une grille carrée généralement de 3 x 3. On veillera à définir un fonctionnement indépendant de la taille afin de pouvoir en changer facilement.

Les deux joueurs dessinent dans la grille alternativement le symbole X pour l'un des joueurs et le symbole O pour l'autre joueur. Le gagnant est celui ayant le premier aligner le même symbole sur une ligne entière ou sur une diagonale entière.

	0	0
0	X	0
X	X	X

Le jeu mène souvent à une situation d'ex-aequo lorsque la grille est complète mais sans aucune ligne ou colonne gagnante.

Déclarations basiques

Vous aurez besoin de fonctions définies dans les modules *Data.Char*, *Data.List* et *System.IO*.

La taille de la grille sera définie simplement par l'expression suivante :

```
size :: Int
size = 3 -- On fixe ici globalement la taille de la grille
```

On définira un joueur comme O ou X ou B (pour un vide) de la manière suivante :

```
data Player = O | X | B
    deriving (Eq, Ord)
```

a) Ecrivez l'instance Show Player.

```
Exemples : show X == " X "
           show B == " B "
```

On définira le type grille comme une liste de liste de joueur :

```
type Grid = [[Player]] -- La grille est une liste de lignes
```

- b) Ecrivez la fonction `next :: Player -> Player` qui retourne le joueur suivant.

Exemples : `next x == O`
`next B == B`

Fonctions utilitaires pour la grille

- c) Ecrivez la fonction `empty :: Grid` qui retourne une grille vide. Utilisez la fonction `replicate :: Int -> a -> [a]` de *Data.List*.

Exemple : `empty == [[B,B,B],[B,B,B],[B,B,B]]` pour une grille de 3x3.

- d) Ecrivez la fonction `full :: Grid -> Bool` qui teste si une grille est pleine (ce qui est différent de savoir si la grille contient une ligne ou une diagonale gagnante).

Indice : transformez la grille en liste plate avec la fonction `concat :: [[a]] -> [a]`, puis testez en utilisant la fonction `all :: (a -> Bool) -> [a] -> Bool` et.

Exemple : `full [[X,O,X],[X,O,X],[O,X,O]] == True`

- e) Ecrivez la fonction `diag :: Grid -> [Player]` qui retourne la diagonale gauche-droite de la grille. Le principe consiste à construire une liste en prenant l'élément (1^{ère} ligne, 1^{ère} colonne) puis l'élément (2^{ème} ligne, 2^{ème} colonne) et ainsi de suite. Utilisez l'opérateur `(!!) :: [a] -> Int -> a` qui retourne le nième élément demandé (1^{er} élément indice 0).

Exemples : `[[1,2,3],[4,5,6]] !! 0 == [1,2,3]`
`[[1,2,3],[4,5,6]] !! 0 !! 0 == 1`

Exemple : `diag [[X,O,X],[X,O,X],[O,X,O]] == [X,O,O]`

- f) Ecrivez la fonction `wins :: Player -> Grid -> Bool` qui teste si un joueur à gagner ou pas. Il faut tester les lignes, les colonnes et les deux diagonales. On utilisera la fonction `transpose :: [[a]] -> [[a]]` pour les colonnes et la fonction *diag* et `reverse :: [a] -> [a]` pour les diagonales.

Exemple : `transpose [[1,2,3],[4,5,6],[7,8,9]]`
`== [[1,4,7],[2,5,8],[3,6,9]]`

Exemple : `map reverse [[1,2,3],[4,5,6],[7,8,9]]`
`== [[3,2,1],[6,5,4],[9,8,7]]`

Exemples : `wins x [[X,O,X],[X,O,X],[O,X,O]] == False`
`wins x [[X,O,X],[X,O,X],[X,X,O]] == True`

- g) Ecrivez la fonction `won :: Grid -> Bool` qui teste si la grille est gagnante pour l'un des deux joueurs.

Exemple : `won [[X,O,X],[X,O,X],[X,X,O]] == True`

Affichage de la grille

Afin d'afficher la grille, vous allez commencer par écrire plusieurs fonctions utilitaires qui seront utilisées dans la fonction principale d'affichage.

```
[*Main>
[*Main> showGrid [[X,O,X],[X,O,X],[X,X,O]]
X | O | X
-----
X | O | X
-----
X | X | O
[*Main> showGrid [[B,B,B],[B,B,B],[B,B,B]]
  |  |
-----
  |  |
-----
  |  |
[*Main>
```

- h) Commencez par écrire la fonction `insVert :: [String] -> [String]` qui insère la chaîne "|" entre le 1^{er} et le 2^{ème} élément d'une liste de chaînes.

Exemples : `insVert [" X "] == [" X "]`
`insVert [" X ", " X "] == [" X ", "|", " X "]`
`insVert [" X ", " O ", " "] == [" X ", "|", " O ", " "]`

- i) Ecrivez une fonction récursive `showRow :: [Player] -> [String]`.

Exemple : `showRow [X,O,B] == [" X ", "|", " O ", "|", " "]`

- j) Ecrivez une fonction `insHoriz :: [String]` qui retourne une liste de chaînes dépendant de la taille de la grille.

Exemples :

`insHoriz == ["---", "---", "---", "- ", "- "]` pour `size = 3`

`insHoriz == ["---", "---", "---", "---", "- ", "- ", "- "]` pour `size = 4`

- k) Ecrivez la fonction récursive `showGrid :: Grid -> IO ()`.

- l) Assurez-vous que l'affichage fonctionne pour une grille de taille 4 par exemple en changeant uniquement la valeur de `size`.

Modification de la grille

Pour modifier la grille, et afin de simplifier le travail, nous allons prendre la convention suivante : les cases de la grille sont numérotées ligne par ligne en partant de zéro sur la 1^{ère} case à gauche.

Exemples :

0	1	2
3	4	5
6	7	8

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- m) Ecrivez la fonction `valid :: Grid -> Int -> Bool` qui teste si la case indiquée existe et si elle est libre.

Exemples : `valid [[B,B,B],[B,B,B],[B,B,B]] 8 == True`
`valid [[X,O,X],[X,O,X],[X,B,O]] 8 == False`
`valid [[B,B,B],[B,B,B],[B,B,B]] 20 == False`

- n) Ecrivez la fonction récursive `cut :: Int -> [a] -> [[a]]` qui coupe une liste en sous-listes de la taille donnée. *take* et *drop* sont vos amies !

Exemple : `cut 3 [X,O,X,X,O,X,X,B,O] == [[X,O,X],[X,O,X],[X,B,O]]`

- o) Ecrivez la fonction `move :: Grid -> Int -> Player -> [Grid]` qui retourne une liste vide ou contenant la nouvelle grille. On testera si la case donnée en 2^{ème} paramètre est valide. Travaillez sur une liste plate de la grille et utilisez *cut* pour reconstituer la grille après insertion du mouvement.

Exemple : `move [[B,B,B],[B,B,B],[B,B,B]] 0 X`
`== [[[X,B,B],[B,B,B],[B,B,B]]]`

Jeu humain vs humain

Vous avez à votre disposition les fonctions suivantes :

```
cls :: IO ()
cls = putStr "\ESC[2J"

goto :: (Int, Int) -> IO ()
goto (x,y) = putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")
```

```

run :: Grid -> Player -> IO ()
run g p = do cls           -- clear screen
             goto (1,1)    -- go to the upper left position
             showGrid g
             run' g p

getNat :: String -> IO Int
getNat message =
    do putStr message
       xs <- getLine
       if xs /= [] && all isDigit xs
       then return (read xs)
       else do putStrLn "Error: invalid number"
               getNat message

tictactoe :: IO ()
tictactoe = run empty 0

prompt :: Player -> String
prompt p = "Player " ++ show p ++ ", enter your move: "

```

p) Complétez le code de la fonction *run'* suivante :

```

run' :: Grid -> Player -> IO ()
run' g p | ...      = putStrLn "Player 0 wins!\n"
         | ...      = putStrLn "Player X wins!\n"
         | ...      = putStrLn "It's a draw!\n"
         | otherwise =
             do i <- ...
              case move g i p of
                [] -> do putStrLn "Error: invalid move"
                    ...
                [g'] -> ...

```

q) Maintenant amusez-vous !

Références web

Site officiel Haskell

<https://www.haskell.org>

Site des modules officiels Haskell

<https://downloads.haskell.org/~ghc/latest/docs/html/libraries/>