Programmation fonctionnelle – TP

Puissance 4

Décembre 2017

L'objectif de ce TP est de programmer le célèbre jeu *Puissance 4*[1], avec une intelligence artificielle d'un niveau raisonnable. On rappelle que le but du jeu est d'aligner 4 pions d'une même couleur dans une grille contenant 7 colonnes de 6 cases chacune.

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

Q1. Proposer un type somme `Color` pour représenter les couleurs des joueurs ;

Q2. Proposer un type algébrique `Cell` pour représenter une case de la grille, qui peut être soit vide, soit remplie avec l'une des deux couleurs[3] ;

Q3. Une colonne sera naturellement une liste de `Cell` et une grille une liste de colonnes. Écrire les déclarations d'un alias[4] `Column` et d'un nouveau type algébrique classique `Grid` correspondant.

2 Créer la grille vide et afficher

Q4. Créer une grille initiale aux bonnes dimensions, `initial::Grid`, qui contient uniquement des cases vides[5] ;

Q5. Écrire une instance de la classe de types `Show` pour `Cell`, afin d'afficher un caractère (et des espaces) pour chacune des valeurs possibles ;

Q6. Écrire une instance de la classe de types `Show` qui crée une chaîne de caractères représentant la grille en deux dimensions (et éventuellement contenant aussi une numérotation pour les colonnes)[6].

1. `https://fr.wikipedia.org/wiki/Puissance_4`
2. `https://www.haskell.org/hoogle/`
3. Rappel : écrire `data Cell = Color | ...` est incorrect car `Cell` et `Color` sont deux types différents. `Color` doit donc être imbriqué dans un *constructeur de données*
4. Indice : en utilisant donc le mot clé `type` plutôt que `data`.
5. Indice : la fonction `replicate` peut être utile.
6. Indice : on pourra utiliser notamment les fonctions `transpose`, `unlines` et `concatMap`.

3 Jouer un coup

Q7. Proposer une fonction `addToken::Column->Color->Column` qui ajoute un pion de la couleur donnée à la colonne donnée. Si la colonne est pleine, le résultat est la colonne inchangée [7] [8].

Q8. Proposer une fonction `play::Grid->Color->Int->Grid` qui ajoute un pion de la couleur donnée dans la colonne dont le *numéro* est donné (entre 0 et 6) [9].

4 Trouver le gagnant

Q9. Proposer une fonction `summarize::[Cell]->[(Cell,Int)]` qui produit la liste des alignements *dans une colonne*, avec leurs longueurs [10]. Par exemple : 2 vides, 3 rouges, 1 vide, 2 jaunes, etc.

Q10. À partir de la fonction précédente, on peut facilement trouver les alignements sur les lignes [11]. Pour les diagonales c'est plus dur : on écrit une fonction `diagonalize::[[a]]->[[a]]` qui génère une liste des contenus des diagonales du haut à droite vers le bas à gauche [12] (ou l'inverse), qui fonctionne pour une matrice quelconque. Par exemple, pour la matrice ci-dessous, il faut produire la liste $[[1],[2,4],[3,5,7],[6,8],[9]]$.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

On pourra écrire la fonction demandée sous la forme d'une liste en compréhension [13].

Q11 Proposer une fonction `won::Grid->Maybe Color` qui indique le gagnant (l'un d'eux s'il y en a deux...) quand il y en a un, et `Nothing` sinon [14] [15] [16].

5 Intelligence artificielle

On implémente un simple algorithme `negaMax` qui consiste à développer les coups successifs de chaque joueur, en profondeur jusqu'à une profondeur donnée à laquelle on évalue statiquement la position. On remonte ensuite en propageant les scores, sachant que Rouge veut maximiser son score et Jaune le minimiser. Pour se ramener à toujours maximiser, on multiplie par -1 les scores quand c'est à Jaune de jouer. Voir les détails sur :
`https://en.wikipedia.org/wiki/Minimax` et `https://en.wikipedia.org/wiki/Negamax`.

7. Indice : regarder la fonction `span`.
8. Indice : le mot-clé `deriving` peut vous aider à définir `(==)` pour `Cell` et `Color`.
9. Indice : `splitAt` peut vous aider.
10. Indice : on pourra écrire une fonction `addCellToList::[(Cell,Int)]->Cell->[(Cell,Int)]` telle que `summarize = foldl' addCellToList []`.
11. Indice : utiliser la fonction `transpose`.
12. Indice : pour les diagonales bas à droite - haut à gauche, on peut aisément obtenir un résultat similaire en composant `diagonalize` et `reverse`.
13. Indice : la liste des listes d'éléments de `g` d'indices (i,j) tels que..., c'est `[[g!!i!!j | ... ] | ... ]`.
14. Indice : on pourra utiliser `listToMaybe`.
15. Indice : on peut utiliser une fonction λ avec *pattern-matching* pour récupérer la valeur de type `Color` dans la `Cell` : p.ex. `\(Full x)->x` si `Full` est le constructeur de données de `Color` dans `Cell`.
16. Indice : `fmap` peut être utilisée pour promouvoir une fonction, ici de type `Cell->Color`, dans le *foncteur* `Maybe`, c'est-à-dire pour la transformer ici en une fonction de type `Maybe Cell->Maybe Color`.

Q12. Proposer une fonction `legalMoves::Grid->[Int]` qui donne la liste des numéros des colonnes (entre 0 et 6) qui ne sont pas pleines ;

Q13. En utilisant `summarize` et `diagonalize` proposer une fonction d'évaluation d'une grille `evaluate::Grid->Int` qui renvoit un score entier, d'autant plus négatif (resp. positif) que Jaune (resp. Rouge) gagne. On propose de faire la somme des longeurs des séquences de Rouge moins la somme des longeurs des séquences de Jaune. Pour que les séquences les plus longues soient plus intéressantes, on propose de pondérer la longueur de chaque séquence de longueur n en la multipliant par 100^n.

Q14. Proposer une fonction `negaMax::Color->Int->Int->Grid->(Int,Int)` qui, à partir de la couleur du joueur qui doit jouer, la profondeur maximum, la profondeur actuelle, et la grille de jeu, renvoit le meilleur score trouvé, avec la colonne qui réalise ce score [17].

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

On souhaite gérer les coups des joueurs de façon générique. On définit donc une classe de types `Contestant` regroupant les fonctionnalités d'un joueur :

```
-- Un joueur générique
-- Notez le IO Int car, pour un joueur humain, il faut lire au clavier
class Contestant a where
    move::a->Grid->IO Int   -- donner un coup à jouer
    color::a->Color         -- donner sa couleur
```

On définit également deux types de joueurs :

```
data Human = Hum Color
data Computer = Com Color
```

Q15. Proposer une instance de `Contestant` pour `Computer` ;

Q16. Proposer une instance de `Contestant` pour `Human` [18] [19] [20]. Il faut vérifier que le coup proposé est légal et demander à nouveau tant qu'il ne l'est pas.

Q17. On écrit finalement la boucle principale du jeu (REPL). Proposer une fonction `loop::(Contestant a,Contestant b)=>Grid->a->b->IO()` qui récupère le coup du premier joueur donné, le joue, affiche le résultat, teste la victoire ou le nul, puis s'appelle récursivement en inversant les joueurs [21].

Q18. Écrire la fonction main comme ci-dessous, puis jouer !

```
main::IO () -- Point d'entrée dans le programme
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
    -- affiche la grille initiale
    putStr $ show2D initial
    -- lance la REPL
    loop initial (Hum Rouge) (Com Jaune)
```

17. Indice : utiliser `maximumBy` et `comparing` pour un max sur la première composante d'une liste de couples.
18. Indice : on pourra utiliser la notation `do` et les fonctions `elem`, `getChar` et `digitToInt`
19. Indice : dans un `do`, `let` n'est pas suivi de `in`.
20. Indice : `IO` étant une monade, on peut transformer un `Int` en `IO Int` avec `return`.
21. Indice : quand la fonction s'arrête, on renvoit un `IO ()` par `return ()`.