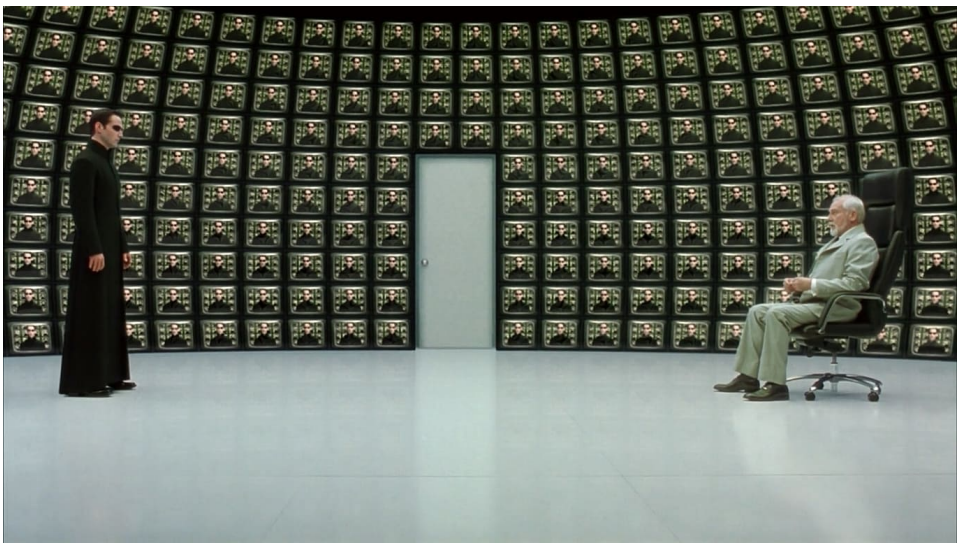
Lambda-calcul et programmation fonctionnelle : Devoir 3

Due : le lundi 6 février 2023 23:59



The Matrix Reloaded, Warner Bros. pictures

Un jour, un étudiant m'a dit « Je n'aime pas les matrices. » Pourtant, les matrices sont utilisées dans des nombreuses applications. En infographie, elles permettent la manipulations des images 3D. Des matrices stochastiques permettent la description des ensembles de probabilités, notamment utilisées pour déterminer l'importance des sites web et guident les moteurs de recherche sur internet. Les matrices sont encore utilisées en économie, en théorie de la décision et en théorie des jeux. Les matrices sont très importantes dans notre société. Nous pouvons ne pas les aimer, mais nous ne pouvons pas nous en passer. « The Matrix has you. »

Dans ce TP, vous devez écrire des fonctions en Haskell pour manipuler des matrices. Nous allons représenter une matrice A de dimensions $n \times m$ et à coefficients réels, par une fonction qui à tout couple d'entiers (i, j) associe le couple (True, a_{ij}) si i et j sont des indices légaux, c.-à-d., sont tels que $1 \leq i \leq n$ et $1 \leq j \leq m$, et $(\text{False}, 0)$ sinon.

Par exemple, la matrice A de dimensions 4×4 ci-dessous est représentée par la fonction qui à tout (i, j) associe $(\text{True}, 2i + j)$ si $1 \leq i, j \leq 4$, et faux sinon.

$$A = \begin{bmatrix} 3 & 4 & 5 & 6 \\ 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 10 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Cette représentation a quelques avantages par rapport à la représentation standard avec des tableaux. Elle est économique pour la représentation des matrices creuses. Par exemple la matrice :

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

peut être représenté par la fonction :

```
sparse (2, 3) = (True, 1)
sparse (3, 2) = (True, 1)
sparse (i, j)
    | 1 <= i && i <= 3 && 1 <= j && j <= 4 = (True, 0)
    | otherwise = (False, 0)
```

Pour chacune des fonctions du TP, vous devez indiquer son type et vous devez implémenter un test. Pour les tests, suivez les mêmes instructions que celles du TP 1.

Exercice 1. En Haskell, nous pouvons utiliser le mot clé `type` pour déclarer un alias de type. Par exemple, le type `String` est déclaré comme suit en Haskell :

```
type String = [Char]
```

Ceci permet des annotations de type plus claires dans le programme. Utilisez cette technique pour déclarer le type `Matrix` qui sera utilisé dans le reste du TP.

Exercice 2. Écrivez la fonction `matA` qui représente la matrice A mentionnée plus tôt dans le TP. Utilisez une définition succincte, n'énumérez pas tous les cas possibles. Exemples :

```
matA (2,3) == (True, 7)
matA (3,8) == (False, 0)
matA (4,4) == (True, 12)
```

Exercice 3. Écrivez la fonction `identity` qui, pour un entier n passé en argument, retourne la fonction qui représente la matrice identité de dimensions $n \times n$. Exemples :

```
identity 3 (2,2) == (True,1)
identity 3 (1,2) == (True,0)
identity 3 (2,7) == (False,0)
```

Exercice 4. Écrivez la fonction `dims` qui, pour une matrice A passée en argument, retourne les dimensions de A. Exemples :

```
dims sparse == (3,4)
dims (identity 3) == (3,3)
dims matA == (4,4)
```

Exercice 5. Écrivez la fonction `toList` qui, pour une matrice A passée en argument, retourne la matrice sous forme de liste de listes. Par exemple :

```
toList sparse == [[0.0,0.0,0.0,0.0],[0.0,0.0,1.0,0.0],[0.0,1.0,0.0,0.0]]
```

Exercice 6. Écrivez la fonction `add` qui, étant donné deux fonctions `matA` et `matB` représentant les matrices A et B, retourne une fonction qui représente la somme des matrices A et B (par définition, $A+B = (a_{ij}+b_{ij})$, pour chaque (i,j)). Exemples :

```
(add (identity 3) (identity 3)) (1,1) == (True, 2)
(add (identity 3) matA) (2,3) == matA (2,3)
(add matA matA) (1,1) == (True, 6)
```

Exercice 7. Comme pour le TP 1, écrivez la fonction :

```
test :: [Bool] -> Bool
```

qui reçoit les résultats d'un test et qui retourne vrai si tous les résultats du test sont vrai et faux sinon. Exemple :

```
test [True, True, True] == True
test [True, False ] == False
```

Écrivez aussi la fonction :

```
testAll :: [Char]
```

qui retourne la chaine de caractères `"Success!"` si tous les résultats des tests de toutes les fonctions du TP sont vrais et `"Fail!"`, sinon.