Scala

Sudoku : fonctionnel (partie 2)

Vérifiez les conventions à suivre pour l'écriture du code (cf Moodle).

Vous allez réaliser ce TD en Scala d'une manière purement fonctionnelle cette fois-ci.

Le TD peut tenir en un seul fichier : vous pouvez définir plusieurs classes et objets dans un même fichier scala. Faites un jar lançable par scala –jar (ne mettez pas les librairies scala dans un jar Java).

Vous indiquerez EXPLICITEMENT les types de retours des méthodes et des fonctions dans les déclarations systématiquement.

Dans ce TD, vous allez vous restreindre au cas de la résolution d'une grille de Sudoku. Ce TD est (très) guidé. Des notions complémentaires sont introduites en cours de route : il est très important de réaliser les exercices dans l'ordre afin de bien comprendre la manière de faire fonctionnelle et la programmation Scala : ici la manière de faire est typique Scala (vous ne pourrez pas faire la même chose en Java par exemple – ou difficilement), il s'agit donc d'un TD plus spécialisé fonctionnel et Scala : ici, un seul paradigme est utilisé, le paradigme fonctionnel.

Vous rendrez un fichier avec chaque étape (questions) + le programme final (en plus des consignes données).

1. Redéfinition des imports.

Normalement, vous pouvez définir un tableau d'entiers de la manière suivante :

```
val table = Array[Int] (
        Array[Int] (1, 2, 3, 4, 5….),
        ….,
        …
)
```

Il est possible de réaliser un alias sur un type en import de la manière suivante :

```
import scala.{Array => A}
```

puis d'utiliser A en place d'Array :

```
val table = A(
        A(1, 2, 3, 4, 5 ….)
        ….,
)
```

Vous pouvez utiliser également $:

Redéfinissez l'import pour permettre d'écrire :

```
val table = $(
  $(5, 3, 0,  0, 7, 0,  0, 0, 0),
```

```
    $(6, 0, 0,  1, 9, 5,  0, 0, 0),
    $(0, 9, 8,  0, 0, 0,  0, 6, 0),

    $(8, 0, 0,  0, 6, 0,  0, 0, 3),
    $(4, 0, 0,  8, 0, 3,  0, 0, 1),
    $(7, 0, 0,  0, 2, 0,  0, 0, 6),

    $(0, 6, 0,  0, 0, 0,  2, 8, 0),
    $(0, 0, 0,  4, 1, 9,  0, 0, 5),
    $(0, 0, 0,  0, 8, 0,  0, 7, 9)
  )
```

On souhaite écrire les tableaux sous une forme texte :

5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

2. Sans utiliser de boucle. En exploitant **map** et **mkstring**, réalisez une ligne :

println(table. ?. ?. ?. ?. ?. ?) qui affiche ce tableau (le nombre de ? n'est pas significatif).

Le tableau table va être dupliqué et modifié : l'idée est d'utiliser un tableau table immuable et donc de ne jamais le modifier. Vous allez donc utiliser la méthode **updated** des collections pour réaliser en même temps une duplication et une modification.

Sur un tableau à une dimension, **val t = Array[Int](1,2,3,4)**, on peut rapidement réaliser une copie avec modification d'une valeur de la manière suivante : **val nt = t.updated(1, 99).**

t ne sera pas modifié et nt contiendra **(1, 99, 2, 3, 4).**

3. Utilisez le même principe pour créer une copie de table avec une modification de la case (4, 5) :

val ntable = table. ?????????

N'utilisez qu'updated (éventuellement plusieurs fois).

Vérifiez que cela marche en affichant **ntable** avec la ligne basée sur **println**, **mkstring** et **map**.

4. Pour réaliser la recherche en fonctionnel, l'itération de parcours sur les cases à remplir va se faire de manière récursive. La suite des questions va vous guider dans la réalisation du programme fonctionnel. Prenez soin de bien analyser chaque étape : les notions abordées seront considérées comme acquises (suite des TD et examen)

Le parcours du tableau à 2 dimensions se fait normalement par deux boucles imbriquées. Ici, on souhaite parcourir le tableau de manière linéaire, c'est-à-dire remplacer les deux boucles imbriquées par une seule boucle qu'on peut ensuite rendre récursive (cf le TD1).

On souhaite numéroter les cases du tableau, de 0 à 80. Comment peut-on convertir ce numéro de case en coordonnées dans le tableau ? Réalisez une fonction qui convertir un numéro de case en couple (x,y) :

def parcours_1(i_ : Int = 0) : Tuple2[Int, Int]

Notez qu'avec une valeur par défaut, vous n'êtes pas obligé de lancer parcours_1(0), vous pouvez écrire directement parcours() c'est ok.

Vous allez maintenant réaliser :

def parcours_2(i_ : Int) : Unit qui est une fonction récursive qui parcours les valeurs de 0 à 80. Faites une première version qui affiche les **valeurs** et la coordonnée associée dans le tableau :

0 -> (0, 0)
1 -> (0,1)
2 -> (0,2) etc..
Modifiez ensuite cette version pour que la condition d'arrêt porte, non plus sur la valeur de i_, mais sur la valeur du couple des coordonnées. Utilisez case et match.

La fonction sera de la forme :

```
def parcours_2(i_ : Int = 0) {
    ( ?, ? ) match {
      case(?, ? ) => println("derniere case")
      case( ?, ?) => {
        println(x + " " + y)
        parcours_2(i_ +1)
      }
    }
  }
```

À vous de voir qui mettre pour le code en rouge. Faites une pause et vérifiez que vous comprenez bien ce code.

Modifiez ce code pour qu'au lieu d'avoir une condition d'arrêt sur la dernière case, on ait une condition d'arrêt sur la case qui suivrait la dernière case (une case de trop).

```
def parcours_3(i_ : Int = 0) {
    ( ?, ? ) match {
      case(?, ? ) => println("on arrete, cette case ne doit pas être analysée")
      case( ?, ?) => {
        println(x + " " + y)
        parcours_3(i_ +1)
      }
```

```
    }
  }
```

Maintenant, on souhaite réaliser un mécanisme qui fabrique les coordonnées présentes sur une ligne de manière fonctionnelle. Les coordonnées varient de 0 à 9. Utilisez map sur les indices de 0 à 9 pour obtenir la liste des coordonnées (sous forme de couple) d'une ligne :

Par exemple

val ligne3 = ?

donne une collection avec (3,0), (3,1), …. (3, 8)

réponse : **(0 until 9).map(v => (3, v))**

Faites la même chose pour une colonne. Le jeu du sudoku présuppose également des carrés de 3 x3 cases. Il y a 9 carrés de 3 cases.

Réalisez de la même manière, un mécanisme en une ligne qui transforme les nombres de 0 à 9 en une collection de coordonnées autours de x et y (x et y donnés).

(0 until 9).map (???????) avec x et y

Par exemple pour x=0 et y = 0

donne (0,0), (0,1), (0,2), (1,0), (1,1), (1, 2), (2,0), (2,1), (2,2)

utilisez + / %

Pour résoudre le problème du sudoku, vous allez devoir parcourir, case par case, le tableau et essayer, case par case, y placer une valeur numérique. Pour chaque case, vous allez devoir essayer plusieurs valeurs numériques : mais ce n'est pas la peine d'essayer des numéros qui sont déjà dans la même ligne, dans la même colonne ou dans le même carré de 3x3.

Donc, pour une case donnée, vous allez devoir calculer les valeurs numériques possibles : c'est-à-dire les chiffres de 1 à 9 MOINS les chiffres qui sont déjà sur cette ligne, cette colonne ou bien dans ce carré 3X3 courant.

Pour déterminer les chiffres à écarter, vous allez donc créer la liste des coordonnées des cases à scruter. Par exemple, si vous êtes en (x,y), il faut chercher les coordonnées de cases de la ligne, de la colonne et du carré en (x,y). Puis fusionner ces coordonnées en un ensemble (set), et enfin, faire un map pour transformer la collection de ces coordonnées en collection des valeurs numériques présentes dans le tableau pour ces coordonnées.

Réalisez la fonction :

```
def parcours_5(i_ : Int = 0) {
    (?, ?) match {
      case(?,?) => println("stop ! une case de trop ")
      case(x,y) => {
```

```
      println(x + " " + y)
      def indicesAVoir(j_ : Int) = List( ??????? )
      println((0 until 9).map(indicesAVoir))
      parcours_5(i_ +1)
    }
  }
}
```

Cette fonction parcourt les valeurs de case 0 à 80 et affiche pour chaque case, la liste des cases à scruter.

Le problème, c'est que ce sont des collections de collection. Utilisez **flatMap** au lieu de **map**. Analysez bien la différence et le résultat.

Maintenant, vous allez utiliser table pour collecter, non plus les coordonnées des cases à scruter, mais la collection des valeurs numériques de ces cases. Cet ensemble sera donc, pour une case (x,y), l'ensemble des valeurs numériques impossibles pour la case courante (puisqu'elles sont déjà dans la ligne, colonne ou dans le carré 3x3).

```
def parcours_7(i_ : Int = 0) {
  ? match {
    ? => println("stop")
    case(x,y) => {
      println(x + " " + y)
      def nombresDejaPris(j_ : Int) = List( ????? )
      println("deja pris "+(0 until 9).flatMap(nombresDejaPris))
      parcours_7(i_ +1)
    }
  }
}
```

Aucun intérêt d'avoir des doublons, alors convertissez le résultat en Set (.toSet).

flatMap(nombresDejaPris).toSet

Du coup, les nombres que vous pouvez essayer pour la case courante sont les nombres de 1 à 9 SANS les nombres déjà pris dans la ligne courante, colonne courante ou carré courant :

```
def parcours_9(i_ : Int = 0) {
  ?? match {
    ??  => println("derniere case")
    case(x,y) => {
      println(x + " " + y)
      def nombresDejaPris(j_ : Int) = ??
      val dejaPris = (0 until 9).flatMap(nombresDejaPris).toSet
```

```
      val aEssayer = ??
      println("a essayer : "+aEssayer)
      parcours_9(i_ +1)
    }
   }
  }
```

Complétez votre code (**aEssayer**). Regardez l'opération ensembliste **diff** dans scaladoc. Si nécessaire, utilisez **toSeq** (idem, regardez dans scaladoc).

Jusqu'à présent, on avait fait un accès au tableau directement : vous allez maintenant modifier votre fonction pour que le tableau soit passé en paramètre. En plus, on souhaite que la fonction renvoie la solution au problème du sudoku. Donc, on veut renvoyer un tableau. Mais s'il n'y a pas de solution, on souhaite renvoyer None.

```
def parcours_11(t_ : Array[Array[Int]], i_  : Int = 0) : Option[Array[Array[Int]]] = {
   ? match {
    ? => ?
    case(x,y) => {
     parcours_11( ?? , i_ +1)
    }
   }
  }
```

Première chose à faire : on renvoie une valeur quand on arrive sur la dernière case (ou la case suivante). La valeur sera un **Some**(un objet de type tableau).
Modifiez le code de parcours_11 pour que le résultat renvoyé par cette fonction soit Some(tableau) dans lequel les valeurs des cases ont été remplacées par i_ (ou le numéro linéaire de la case).
Utilisez **updated** dans ?? pour passer une copie de tableau avec la case courante modifiée.

Pause : ne dépassez pas cette ligne sans être certain d'avoir compris ce qui précède.

Maintenant, vous allez vous attaquer au problème fonctionnel du sudoku.
La fonction complète possède la structure suivante :

```
def parcours_12(t_ : Array[Array[Int]], i_  : Int = 0) : Option[Array[Array[Int]]] = {
   (?, ?) match {
    case( ??) => Some(t_) // renvoie le résultat qui est le tableau courant t_
    case(x,y) if t_(x)(y) != 0 =>  ?? // on passe à la case suivante, inutile de modifier celle-ci
                                      // il y a déjà quelque chose dedans
    case(x,y) => { // sinon, on est sur une case et on va devoir essayer plusieurs numéros
     def nombresDejaPris(j_ : Int) = ?
     val dejaPris = ?
     val aEssayer = ?
```

```
      if (aEssayer.isEmpty) None // pas de solution (rien à essayer)
      else {
        ?????????????
      }

    }
  }
}
```

La valeur de ???????????????? sera renvoyée par la fonction. Donc c'est bien là que vous allez réaliser le calcul de la solution. Plus précisément, dans cette partie du code, vous allez essayer successivement de placer une des valeurs de **aEssayer** en (x,y) puis de continuer le placement sur la case suivante.
Si **aEssayer** est vide, ça veut dire que vous essayez de remplir une case vide du tableau et que vous n'avez aucune possibilité de numéro pour cette case : tous les numéros ont déjà été pris dans les lignes, les colonnes ou les carrés 3x3. Donc vous tombez sur une case à remplir mais rien à mettre dedans : c'est une impasse et vous ne trouverez pas de solution là. Donc on renvoie None.
Si par contre, **aEssayer** n'est pas vide, on peut continuer à chercher.

Il faut envisager ce que seraient les solutions si on prenait comme valeur pour la case courante, une des valeurs **d'aEssayer**.

L'idée classique en impératif est de faire une boucle. Pas en fonctionnel : l'idée principale en fonctionnel est de transformer la collection de **aEssayer** en collection de solutions.

Donc vous allez passer par un map : **aEssayer.map(?)**

Supposons que la collection des aEssayer soit (3, 5, 6). Ça veut dire que pour la case courante, on peut prendre 3 ou 5 ou 6. Si on prend 3 pour la case courante, alors la recherche de la solution continue avec un tableau dans lequel la case (x,y) prend la valeur 3 ET un indice de i_ +1.

Proposez un contenu pour map pour transformer la collection des valeurs possibles de la cases courante en collection de matrices solutions (Some() ou None).

Le problème c'est que vous obtenez une collection de solutions, mais que la fonction renvoie UNE solution Some() ou None.

Donc, vous allez renvoyer la première solution trouvée. C'est-à-dire, dans la collection des solutions, la première solution non None.

aEssayer.map(?).filterNot(_ == None) pour enlever ***les Nones***.

Oui mais le résultat peut être vide (que des nones, vous imaginez ?).

Donc il faut tester la taille de la collection résultat. Si la collection est non vide, on rend la première matrice.

```
val ts = aEssayer.map(?).filterNot(_ == None)
if (ts.length > 0) ts(0) else None
```

Complétez le code:

```
def parcours_12(t_ : Array[Array[Int]], i_  : Int = 0) : Option[Array[Array[Int]]] = {
    (?, ?) match {
     case(?, ?) => Some(t_)
     case(x,y) if t_(x)(y) != 0 => parcours_12(t_, i_ +1)
     case(x,y) => {
      def nombresDejaPris(j_ : Int) = ?
      val dejaPris = ?
      val aEssayer = ?

      // si on place le nombre j_ sur la case courante et qu'on avance ?
      def placer(j_ : Int) = parcours_12( ?, i_ +1)

      if (aEssayer.isEmpty) None // pas de solution (rien à essayer)
      else {
       val ts = aEssayer.map(placer).filterNot(_ == None)
       if (ts.length > 0) ts(0) else None
      }
     }
    }
  }
```

Et vérifiez que le code fonctionne :

```
  parcours_12(table) match {
     case Some(res) => println("12: \n"+ res.map( _.mkString(" ") ).mkString("\n"))
     case None => println("pas de solution")
}
```

Comparez ce code avec la version mixte impérative / fonctionnelle du TD précédent.

Remarquez les choses suivantes :

La solution est bien une fonction SANS objets

Aucune itération

On a rajouté des variables intermédiaires pour la lisibilité, mais on peut enlever TOUTES les variables (par substitution des valeurs).

Il n'y a pas d'effets de bord.

Le code est considérablement plus compact que la version impérative. (ma solution fait par exemple 13 lignes)

Il n'est pas plus lent.

Il est encore possible de l'améliorer en piochant un peu plus loin coté Scala et fonctionnel.
Très spécifique Scala (ne pas apprendre)

Il est possible (entre autres) de remplacer

```
if (aEssayer.isEmpty) None // pas de solution (rien à essayer)
else {
 val ts = aEssayer.map(placer).filterNot(_ == None)
 if (ts.length > 0) ts(0) else None
}
```

par cette seule ligne :

aEssayer.collectFirst(Function.unlift(placer))

remplace le map filter etc..

collectFirst renvoie la première solution (non **None**) renvoyée par **placer** (donc ne calcule pas toutes les solutions et écarte les **Nones**). **collectFirst** prend une fonction partielle en argument (et ne traite que ce qui est défini pour la fonction partielle) mais **placer** n'est pas une fonction partielle, c'est une fonction qui renvoie **Option[Array[Array[Int]].** Il est possible de convertir une fonction à Option en fonction partielle en utilisant **Function.unlift**.

En vous inspirant de ce qui précède dans ce TD, réalisez une fonction qui prend une chaîne de caractères en paramètre et renvoie l'ensemble des anagrammes de cette chaîne. On vous demande bien ici de réaliser cet exercice de manière purement fonctionnelle.

```
def anagrams(chaine_ : String): Set[String]
```