Exercice 1

Démontrez l'égalité

```
flatten(xss:::yss) = flatten(xss):::flatten(yss)
```

par *induction structurelle* sur `xss`.

La fonction `flatten` permet d'aplatir le contenu d'une liste de listes ; par exemple :

```
flatten(List(List(1, 2), List(4, 5))) = List(1, 2, 4, 5)
```

Utilisez pour votre démonstration les définitions suivantes des fonctions `flatten` et `:::` (également présentes dans la classe `List.scala` de la librairie standard) :

```
def flatten[A](l: List[List[A]]): List[A] = l match {
  case Nil => Nil                              // 1ère clause
  case head :: tail => head ::: flatten(tail) // 2ième clause
}

def :::[B >: A](prefix: List[B]): List[B] = prefix match {
  case Nil => this                               // 1ère clause
  case head :: tail => head :: (tail ::: this)  // 2ième clause
}
```

et le lemme suivant (associativité de : : :) présenté durant le cours :

```
(xss:::yss):::zss = xss:::(yss:::zss)  // lemme 1
```

Indication : justifier *chaque* étape de réduction comme présenté dans le cours (en indiquant par exemple *(selon lemme 1)*).

Exercice 2

On considère les deux fonctions de renversement de liste suivantes :

```
def reverseSpec[A](l: List[A]): List[A] = l match {
  case Nil => Nil
  case x :: xs => reverseSpec(xs) ::: List(x)
 }
```

```
def reverseImpl[A](l: List[A]): List[A] = {
  def aux(l: List[a], acc: List[A]): List[A] = l match {
    case Nil => acc
    case x :: xs => aux(xs, x :: acc)
  }
  aux(l, Nil)
}
```

La première, `reverseSpec`, est très intuitive et peut être vue comme une *spécification* du renversement de liste. La seconde, `reverseImpl`, est plus efficace car elle est récursive terminale et ne fait pas appel à la coûteuse méthode `:::`. Elle peut donc être vue comme une *implantation* du renversement de liste.

Montrez par induction que pour toutes listes `l` et `acc`

$$\texttt{aux}(\texttt{l}, \texttt{acc}) = \texttt{reverseSpec}(\texttt{l}) \texttt{:::} \texttt{acc}$$

et déduisez-en que les deux fonctions de renversement sont équivalentes.

*I*ndice : dans le cours on a montré que l'opérateur `:::` est associatif et qu'il admet la liste vide comme élément neutre à gauche et à droite. Souvenez-vous d'autre part que la fonction `List` satisfait l'équivalence suivante :

$$\texttt{List}(e_1, ..., e_n) = e_1 \texttt{::} ... \texttt{::} e_n \texttt{::} \texttt{Nil}.$$

et que une méthode 'op' dont le nom finit avec ' :' est traité de manière suivante :

```
a op b ==> val x = b; x.op(a)
```

Exercice 3

(a) Si Scala ne serait pas equippe de la notation for, comment pourrait-on exprimer ce programme :

```
for(val x <- xs; x >=0) yield Math.sqrt(x)
```

(b) Un triplet $\langle a, b, c \rangle$ est pythagoricien si $a^2 + b^2 = c^2$. Trouver tous les triplets pythagoriciens avec $a \leq b \leq c \leq 100$ en utilisant la notation for et `List.range`.

Exercice 4

Le problème des n reines est défini ainsi : comment placer sur un échiquier de taille $n \times n$ un ensemble de n reines de manière à ce qu'aucune des reines ne soit en conflit avec une autre reine. Deux reines sont en conflit si elles se trouvent sur la même ligne, la même colonne ou la même diagonale sur l'échiquier. La figure 1 présente une des deux solutions au problème des 4 reines.

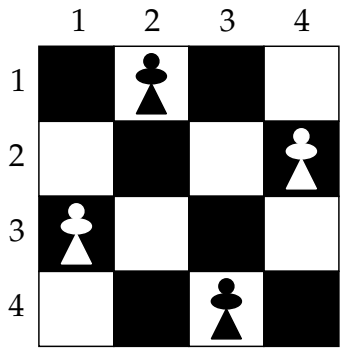

FIG. 1 – Une solution au problème des 4 reines

Une solution partielle au problème a été vue au cours et se trouve sur notre page Web. Complétez-la en écrivant la fonction `isSafe`.

La fonction `queens` retourne une liste des solutions possibles au problème des n reines. Une solution est représentée comme une liste d'entiers spécifiant les colonnes sur lesquelles placer les différentes reines. La ligne est implicitement donnée par la position de la colonne dans cette liste, cette dernière étant triée par numéro de ligne décroissant. Ainsi, la solution de la figure 1 est représentée par la liste `List(3,1,4,2)`.

Une observation que l'on peut immédiatement faire est qu'il ne peut y avoir qu'une reine par ligne dans une solution correcte. La fonction `placeQueens` se base sur cette observation pour placer n reines dans les n premières lignes de l'échiquier, de manière récursive. Pour placer la n^{e} reine, elle commence par chercher (récursivement) toutes les possibilités de placement des $n-1$ reines dans les $n-1$ premières lignes. Ensuite, elle essaie d'ajouter une reine de plus à chacune de ces possibilités, et ne garde que les solutions non conflictuelles.

La fonction `isSafe` indique s'il est possible d'ajouter une reine à la colonne donnée sans entrer en conflit avec les reines déjà placées. Le paramètre `delta` donne le nombre de lignes entre celle contenant la reine en cours de placement et celle en tête de la liste `queens`.