Exercice 1 : Fonctions d'ordre supérieur

1) L'expression suivante

```
(List.range(0, 10) foldRight 0) { (i, acc) => i * i + acc }
```

calcule la somme des carrés des 10 premiers entiers.

2) La fonction suivante

```
def foo[a](l1: List[a], l2: List[a]): List[a] = l1 match {
  case Nil => l2
  case x :: xs => x :: foo(xs, l2)
}
```

concatène les deux listes passées en argument (mais n'inverse pas la première liste, contrairement à ce que certains ont pu penser).

3) On demandait ensuite une fonction équivalente qui ne soit pas récursive, c'est-à-dire qui ne s'appelle pas elle-même, comme le fait la fonction `foo`. Voici une solution simple qui utilise la méthode `foldRight` :

```
def concat[a](xs: List[a], ys: List[a]): List[a] =
  (xs foldRight ys) { (x, acc) => x :: acc };
```

Remarque : évidemment il fallait éviter d'utiliser la méthode `:::` de la classe `List`.

Exercice 2 : Entiers inductifs

On donnait la définition suivante des entiers :

```
trait Nat {
  def + (n: Nat): Nat = match {
    case Z    => n         //    Z + n = n
    case S(m) => S(m + n)  // S(m) + n = S(m + n)
  }
}
case object Z extends Nat;         // Zero
case class S(n: Nat) extends Nat;  // Successeur de n
```

1) La première propriété à montrer est l'associativité de l'opérateur + :

$$\forall m, n, p : Nat, \quad (m + n) + p \quad = \quad m + (n + p)$$

On fait la preuve par induction structurelle sur m.

Cas [$m = Z$].
À gauche, on a :

	$(m + n) + p$	
=	$(Z + n) + p$	remplacement de m par sa valeur
=	$n + p$	définition de +, premier cas

À droite, on a :

	$m + (n + p)$	
=	$Z + (n + p)$	remplacement de m par sa valeur
=	$n + p$	définition de +, premier cas

Cas [$m = S(m')$].
On dispose de l'hypothèse d'induction suivante :

$$\forall n, p : Nat, \quad (m' + n) + p \quad = \quad m' + (n + p)$$

À gauche, on a :

	$(m + n) + p$	
=	$(S(m') + n) + p$	remplacement de m par sa valeur
=	$S(m' + n) + p$	définition de +, deuxième cas
=	$S((m' + n) + p)$	définition de +, deuxième cas
=	$S(m' + (n + p))$	hypothèse d'induction

À droite, on a :

	$m + (n + p)$	
=	$S(m') + (n + p)$	remplacement de m par sa valeur
=	$S(m' + (n + p))$	définition de +, deuxième cas

2) On veut maintenant montrer que Z est un élément neutre à droite de l'opérateur + :

$$\forall n : Nat, \quad n + Z \quad = \quad n$$

Encore, une fois, on fait la preuve par induction structurelle sur m.

Cas [$n = Z$].

$$
\begin{array}{lll}
 & n + Z & \\
= & Z + Z & \text{remplacement de } n \text{ par sa valeur} \\
= & Z & \text{définition de +, premier cas} \\
= & n &
\end{array}
$$

Cas [$n = S(n')$].

On dispose de l'hypothèse d'induction suivante :

$$n' + Z \quad = \quad n'$$

$$
\begin{array}{lll}
 & n + Z & \\
= & S(n') + Z & \text{remplacement de } n \text{ par sa valeur} \\
= & S(n' + Z) & \text{définition de +, deuxième cas} \\
= & S(n') & \text{hypothèse d'induction} \\
= & \text{n} &
\end{array}
$$

3) La troisième propriété qu'il fallait montrer est :

$$\forall m, n : Nat, \quad m + S(n) \quad = \quad S(m + n)$$

Pour changer, on fait la preuve par induction structurelle sur m.

Cas [$m = Z$].

À gauche, on a :

$$
\begin{array}{lll}
 & m + S(n) & \\
= & Z + S(n) & \text{remplacement de } m \text{ par sa valeur} \\
= & S(n) & \text{définition de +, premier cas}
\end{array}
$$

À droite, on a :

$$
\begin{array}{lll}
 & S(m + n) & \\
= & S(Z + n) & \text{remplacement de } m \text{ par sa valeur} \\
= & S(n) & \text{définition de +, premier cas}
\end{array}
$$

Cas [$m = S(m')$].

On dispose de l'hypothèse d'induction suivante :

$$\forall n : Nat, \quad m' + S(n) \quad = \quad S(m' + n)$$

À gauche, on a :

$$
\begin{array}{lll}
 & m + S(n) & \\
= & S(m') + S(n) & \text{remplacement de } m \text{ par sa valeur} \\
= & S(m' + S(n)) & \text{définition de +, deuxième cas} \\
= & S(S(m' + n)) & \text{hypothèse d'induction}
\end{array}
$$

À droite, on a :

$$
\begin{array}{lll}
 & S(m + n) & \\
= & S(S(m') + n) & \text{remplacement de } m \text{ par sa valeur} \\
= & S(S(m' + n)) & \text{définition de +, deuxième cas}
\end{array}
$$

Exercice 3 : Décomposition en facteurs premiers

Pour décomposer un nombre n en facteurs premiers, on peut commencer avec le premier nombre premier 2 et regarder s'il divise n. Si c'est le cas, on continue en regardant si $n/2$ est divisible par le premier nombre premier. Sinon, on tente de diviser n par le prochain nombre premier, 3. En continuant ainsi, on arrive forcément dans la situation où n a été tellement divisé qu'il ne reste plus qu'à traiter le nombre 1. Ce dernier n'ayant pas de diviseurs premiers, la liste de ces facteurs premiers est vide.

C'est cette idée d'algorithme toute simple qui est exprimée dans la fonction `primeFactors` suivante, qui fait appel à une fonction récursive auxiliaire `pf` et au flot des nombres premiers vu en cours :

```
def primeFactors(n: int): List[int] = {
  def pf(n: int, primes: Stream[int]): List[int] =
    if (n == 1)
      List()
    else {
      val next = primes.head;
      if (n % next == 0)
        next :: pf(n / next, primes)
      else pf(n, primes.tail)
    }
  pf(n, primes)
}
```

En fait plutôt que de parcourir la suite des nombres premiers à la recherche de diviseurs pour n, on peut aussi parcourir la suite des nombres entiers, tout simplement. C'est ce qui est fait dans la fonction équivalente suivante :

```
def primeFactors(n: int): List[int] = {
  def pf(n: int, next: int): List[int] =
    if (n == 1)
      List()
    else if (n % next == 0)
      next :: pf(n / next, next)
    else
      pf(n, next + 1);
  pf(n, 2)
}
```

Les plus pessimistes d'entre nous pourraient craindre d'ajouter ainsi dans la liste des nombres qui ne sont pas premiers, vu qu'on parcourt la suite des entiers sans se soucier de savoir si l'on a affaire à un nombre premier ou pas. Mais cela n'a pas lieu, car on peut se convaincre que si $next$ divise n à une étape donnée alors forcément il est premier.

Pour finir en beauté, voici un exemple d'exécution de cette dernière fonction :

```
primeFactors(12)  →  pf(12, 2)
                  →  2 :: pf(6, 2)
                  →  2 :: 2 :: pf(3, 2)
                  →  2 :: 2 :: pf(3, 3)
                  →  2 :: 2 :: 3 :: pf(1, 3)
                  →  2 :: 2 :: 3 :: Nil
```

Exercice 4 : Flots

1) La fonction `zip` doit apparier les éléments de deux flots infinis. Elle doit donc retourner un flot de paires. Voici une solution simple qui additionne les têtes des deux flots pour obtenir la tête du flot de paires et s'appelle récursivement sur leurs queues pour obtenir la queue du flot de paires :

```
def zip[a,b](xs: Stream[a], ys: Stream[b]): Stream[Pair[a,b]] =
  Stream.cons(Pair(xs.head, ys.head), zip(xs.tail, ys.tail));
```

Remarque : évidemment il fallait éviter d'utiliser la méthode `zip` de la classe `Stream`.
2) L'opération inverse `unzip` consiste à prendre un flot de paires et à retourner une paire de flots. De manière générale, quand on désire retourner plusieurs valeurs dans une fonction, on peut utiliser l'une des classes `Pair`, `Triple`, `Tuple4`, `Tuple5`, etc.

Dans la solution qui suit, on utilise la méthode `map`. En effet, si `ps` est un flot de paires, alors `ps map { p => p._1 }` représentera le flot des premières composantes de chaque paire :

```
def unzip[a,b](ps: Stream[Pair[a,b]]): Pair[Stream[a], Stream[b]] =
  Pair(ps map { p => p._1 }, ps map { p => p._2 });
```

Voici pour finir une solution récursive élégante qui ne fait pas appel à la méthode `map` :

```
def unzip[a,b](ps: Stream[Pair[a,b]]): Pair[Stream[a],Stream[b]] =
  Pair(Stream.cons(ps.head._1, unzip(ps.tail)._1),
       Stream.cons(ps.head._2, unzip(ps.tail)._2));
```

Exercice 5 : Intervalles

Dans cet exercice, les ensembles d'entiers sont représentés par des listes de paires :

```
type Set = List[Pair[int, int]];
```

1) L'ensemble d'entiers représenté par une liste de paires est l'union des intervalles représentés par ces paires. Donc un élément appartient à cet ensemble si et seulement si il existe une paire (l, r) dans la liste tel que $l \leq x \leq r$, ce qui s'exprime très naturellement en Scala :

```
def contains(s: Set, x: int): boolean =
  s exists { case Pair(l, r) => l <= x && x <= r };
```

Pour ceux qui auraient encore un peu de mal avec les fonctions anonymes à base de `case` comme `{ case Pair(l, r) => l <= x && x <= r }`, il faut savoir qu'on aurait aussi pu écrire (de façon moins élégante toutefois) :

```
def contains(s: Set, x: int): boolean =
  s exists { p => p match { case Pair(l, r) => l <= x && x <= r } };
```

Enfin, voici une version optimisée qui a l'avantage, par rapport à celles utilisant `exists`, de ne pas parcourir toute la liste dans le cas où l'élément n'appartient pas à l'ensemble :

```
def contains(s: Set, x: int): boolean = s match {
  case Nil => false
  case Pair(l, r) :: rest =>
    if (x < l) false
    else if (x > r) contains(rest, x)
    else true
}
```

2) La taille d'un ensemble s'obtient simplement en sommant la taille des différents intervalles le composant. La taille d'un intervalle n'est rien d'autre que la différence entre les deux bornes, plus un. C'est l'occasion ou jamais de mettre à profit la méthode `foldLeft` :

```
def size(s: Set): int =
  (s foldLeft 0) { (acc, i) => acc + (i._2 - i._1 + 1) };
```

3) Pour finir, voici la fonction qui calcule l'union de deux ensembles :

```
def union(s1: Set, s2: Set): Set = Pair(s1, s2) match {
  case Pair(Nil, _) => s2
  case Pair(_, Nil) => s1
  case Pair(Pair(l1, r1) :: rest1, Pair(l2, r2) :: rest2) =>
    if (r2 < r1)
      union(s2, s1)
    else if (r1 + 1 >= l2)
      union(rest1, Pair(min(l1, l2), r2) :: rest2)
    else
      Pair(l1, r1) :: union(rest1, s2)
}
```

Commentons-la quelque peu.

Si l'un des deux ensembles est vide, alors l'union est simplement égale à l'autre ensemble (deux premières branches du filtrage de motifs).

Sinon, on compare les premiers intervalles `Pair(l1, r1)` et `Pair(l2, r2)` de chacune des deux listes.

Si le deuxième intervalle finit avant le premier ($r_2 < r_1$) alors on s'appelle récursivement en inversant les deux ensembles. Cela nous permet de supposer dans la suite que le deuxième intervalle finit bien après le premier, ce qui simplifie certains tests.

Si les deux intervalles sont adjacent ou se recoupent ($r_1 + 1 \geq l_2$), alors on peut former un nouvel intervalle correspondant à l'union des deux intervalles avec l'expression `Pair(min(l1,l2),r2)`. L'erreur serait alors de retourner

```
Pair(min(l1, l2), r2) :: union(rest1, rest2)
```

, car il y a encore la possibilité que `Pair(min(l1, l2), r2)` soit adjacent au premier intervalle de `rest1`. Pour prendre en compte cette éventualité, on place le nouvel intervalle en tête de `rest2` et on appelle récursivement `union`.

Le dernier cas correspond à la situation où les intervalles sont disjoints et non adjacents. L'union des deux ensembles commencent alors forcément par l'intervalle `Pair(l1, r1)` et on appelle récursivement la fonction `union` pour continuer la fusion des deux ensembles.

Remarque : la fonction `union` utilise la fonction `min` qui calcule le minimum de deux entiers et qu'on peut définir ainsi :

```
def min(x: int, y: int): int = if (x <= y) x else y;
```