Exercice 1

Il est très simple de modéliser une pile au moyen d'une liste : empiler un élément revient à l'ajouter en tête de la liste, et dépiler un élément revient à extraire la tête de la liste. La classe `Stack` contient donc simplement une variable stockant la pile sous forme de liste, en plus des méthodes `push` et `pop`.

À noter que la classe `Stack` doit être polymorphe, c.-à-d. fonctionner quel que soit le type des éléments qu'on y stocke. Elle prend donc un paramètre de type, ici `a`, qui représente le type de ses éléments.

```
class Stack[a] {
  private var contents: List[a] = List();
  def push(elem: a): unit = { contents = elem :: contents; () }
  def pop: a = contents match {
    case List() => error("pop: pile vide")
    case top :: rest => contents = rest; top
  }
}
```

Exercice 2

Il s'agissait de prouver la propriété suivante :

$$(l_1 \; append \; l_2).length \;=\; l_1.length \;+\; l_2.length$$

On fait la preuve par induction structurelle sur l_1.

Cas $l_1 = IntNil$:

D'un côté

$$\begin{array}{rcl} (l_1 \; append \; l_2).length & = & \\ (IntNil \; append \; l_2).length & = & \text{(substitution de } l_1 \text{ par sa valeur)} \\ l_2.length & & \text{(définition de } append \text{ } (IntNil)) \end{array}$$

Et de l'autre

$$\begin{array}{rcl} l_1.length \;+\; l_2.length & = & \\ IntNil.length \;+\; l_2.length & = & \text{(substitution de } l_1 \text{ par sa valeur)} \\ 0 \;+\; l_2.length & = & \text{(définition de } length \text{ } (IntNil)) \\ l_2.length & & \text{(propriété triviale du } + \text{ Scala)} \end{array}$$

Cas $l_1 = IntCons(head,\ tail)$:

D'un côté

$$\begin{aligned}
(l_1\ append\ l_2).length &= \\
(IntCons(head,\ tail)\ append\ l_2).length &= \text{(substitution de } l_1 \text{ par sa valeur)} \\
IntCons(head,\ tail\ append\ l_2).length &= \text{(définition de } append \text{ (}IntCons\text{))} \\
1\ +\ (tail\ append\ l_2).length &= \text{(définition de } length \text{ (}IntCons\text{))} \\
1\ +\ (tail.length\ +\ l_2.length) & \quad \text{(hypothèse d'induction sur } tail\text{)}
\end{aligned}$$

Et de l'autre

$$\begin{aligned}
l_1.length\ +\ l_2.length &= \\
IntCons(head,\ tail).length\ +\ l_2.length &= \text{(substitution de } l_1 \text{ par sa valeur)} \\
(1\ +\ tail.length)\ +\ l_2.length &= \text{(définition de } length \text{ (}IntCons\text{))} \\
1\ +\ (tail.length\ +\ l_2.length) & \quad \text{(propriété triviale du + Scala)}
\end{aligned}$$

Exercice 3

La définition de cette fonction se fait relativement aisément au moyen du filtrage de motifs, notre but étant de reconnaître les expressions d'une forme donnée.

La fonction suivante simplifie les expressions de manière récursive. L'idée est qu'une expression élevée à la puissance 1 est l'expression elle-même ; une expression élevée à la puissance n est l'expression elle-même multipliée par son élévation à la puissance $n-1$.

Il faut faire bien attention à tout transformer, en particulier les sous-expressions des sommes et des produits, ainsi que les expressions que l'on élève à la puissance.

```
def removePower(e: Expr): Expr = e match {
  case Const(_) => e
  case Plus(l, r) => Plus(removePower(l), removePower(r))
  case Times(l, r) => Times(removePower(l), removePower(r))
  case Power(e, 1) => removePower(e)
  case Power(e, n) => removePower(Times(e, Power(e, n - 1)))
}
```

Exercice 4

Pour obtenir toutes les permutations d'une liste, le plus simple est d'utiliser une solution récursive :

- la liste vide n'admet qu'une permutation : elle-même,
- les permutations d'une liste non-vide s'obtiennent en calculant l'ensemble des permutations de sa queue, puis en ajoutant à chacune d'elles l'élément en tête de la liste originale, à chaque position.

Par exemple, pour obtenir les permutations de la liste suivante :

```
List(1,2,3)
```

on obtient d'abord les permutations de sa queue, qui sont :

```
List(2,3) List(3,2)
```

et on ajoute ensuite à chacune d'elles l'élément en tête de la liste originale, à savoir `1`, à toutes les positions possibles : en tête, au milieu, et en queue. On obtient alors :

```
List(1,2,3) List(2,1,3) List(2,3,1)
List(1,3,2) List(3,1,2) List(3,2,1)
```

En admettant qu'on ait à disposition une fonction `insertAt` permettant d'insérer un élément dans une liste à une position donnée, on a donc la solution suivante en Scala :

```
def permute[a](l: List[a]): List[List[a]] = l match {
  case List() => List(List())
  case head :: tail =>
    for (val perm <- permute(tail);
         val i <- List.range(0, perm.length + 1)) yield
           insertAt(perm, head, i)
};
```

Il reste maintenant à écrire la fonction `insertAt`, ce qui se fait aisément au moyen d'une solution récursive. Cette fonction est assez similaire à celle insérant un élément dans une liste déjà triée, telle que celle vue à la série 8.

```
def insertAt[a](list: List[a], elem: a, pos: int): List[a] =
  if (pos == 0)
    elem :: list
  else
    list.head :: insertAt(list.tail, elem, pos - 1);
```

Exercice 5

Pour séparer un flot en ses composantes d'indice pair et impair, une solution est d'écrire deux fonctions qui extraient respectivement les éléments d'indice pair et les éléments d'indice impair d'un flot ; appelons-les `even` et `odd`. Au moyen de ces deux fonctions, l'écriture de `separate` est très simple :

```
def separate[a](s: Stream[a]): Pair[Stream[a], Stream[a]] =
  Pair(even(s), odd(s));
```

La première fonction, `even`, prend un flot et retourne le flot de ses éléments d'indice pair. Comme l'énoncé ne spécifie rien quant à l'indice du premier élément d'un flot, considérons qu'il a l'indice 0. Le premier élément du flot retourné par `even` doit donc être le premier élément du flot original. La queue du flot retourné doit quant à elle être composée des éléments d'indice *impair* de la *queue* du flot original. Pour extraire ces éléments, on utilise simplement la fonction `odd` qui reste à définir. On obtient alors :

```
def even[a](s: Stream[a]): Stream[a] =
  Stream.cons(s.head, odd(s.tail));
```

Reste maintenant à définir cette fonction `odd`. On peut faire l'observation simple suivante : le flot des éléments d'indice impair d'un flot donné n'est rien d'autre que le flot des éléments d'indice *pair* de la *queue* du flot original. `odd` se définit donc trivialement en termes de `even`, de la manière suivante :

```
def odd[a](s: Stream[a]): Stream[a] =
  even(s.tail);
```

Les deux fonctions `even` et `odd` sont dites *mutuellement récursives* car chacune dépend de l'autre.

La fonction `merge` est plus simple à définir que la fonction `separate`. L'idée est toutefois assez similaire : le premier élément du flot résultat est le premier élément du premier flot ; la queue du flot résultat s'obtient en joignant récursivement le second flot et la queue du premier flot.

```
def merge[a](s1: Stream[a], s2: Stream[a]): Stream[a] =
  Stream.cons(s1.head, merge(s2, s1.tail));
```

C'est en inversant le rôle des flots `s1` et `s2` dans l'appel récursif à `merge` qu'on obtient le résultat escompté, à savoir un entrelacement des deux flots.