Exercice 1 : Décomposition fonctionnelle et orientée-objet (20 points)

Etant-donné le programme suivant.

```
abstract class A
case class B(x: A, y: A) extends A
case class C(x: Int) extends A
def g(f: (Int, Int) => Int)(a: A): Int = a match {
  case B(x, y) => f(g(f)(x), g(f)(y))
  case C(x) => x
}
```

1. Donnez un expression exemplifiant l'utilisation de la fonction `g` et expliquez l'action de cette expression. Pour cela, référez-vous à une représentation intuitive du type de donnée défini par `A`.
2. Définissez en Scala une structure de données correspondante à A et dotée d'une fonction ou d'une méthode dont l'effet est équivalent à celui de g, mais en utilisant uniquement la décomposition orientée objet (et non la décomposition fonctionnelle comme ci-dessus).

SOLUTION

1.
```
g((a, b) => a + b)(B(C(2), C(3))) = 5
```

Cette expression calcule la somme des valeurs de toutes les feuilles d'un arbre binaire.

2.
```
abstract class IntTree {
  def fold(f: (Int, Int) => Int): Int
}
class Join(l: IntTree, r: IntTree) extends IntTree {
  def fold(f: (Int, Int) => Int) = f(l fold f, r fold f)
}
class Leaf(v: Int) extends IntTree {
  def fold(f: (Int, Int) => Int) = v
}
```

Exercice 2 : Manipulation de textes (20 points)

Nous représentons les chaînes de caractères de la manière suivante.

```
abstract class Text
case class Chars(cs: List[Char]) extends Text
case class Concat(t1: Text, t2: Text) extends Text
```

Cette représentation n'est pas plus riche qu'un simple `List[Char]`, son seul avantage est de permettre la concaténation en temps constant. La concaténation de `t1` et `t2` s'obtient en construisant un noeud `Concat(t1, t2)`.

Partie 1

Définissez pour la classe `Text` les méthodes `isEmpty`, `head`, `tail` et `map`. La sémantique de ces méthodes doit être identique à celle obtenue avec les méthodes de même nom sur une `List[Char]` équivalente.

```
abstract class Text {
  def isEmpty: Boolean = this match {
    case Chars(cs) => cs.isEmpty
    case Concat(t1, t2) => t1.isEmpty && t2.isEmpty
  }
  def head: Char = this match {
    case Chars(cs) => cs.head
    case Concat(t1, t2) => if (!t1.isEmpty) t1.head else t2.head
  }
  def tail: Text = this match {
    case Chars(cs) => Chars(cs.tail)
    case Concat(t1, t2) =>
      if (t1 isEmpty) t2.tail
      else { val t = t1.tail
             if (t isEmpty) t2
             else Concat(t, t2) }
  }
  def map(f: Char => Char): Text = this match {
    case Chars(cs) => Chars(cs map f)
    case Concat(t1, t2) => Concat(t1 map f, t2 map f)
  }
}
```

Partie 2

Définissez une fonction `equal` qui retourne `true` si les deux objets `t1` et `t2` de type `Text` représentent la même séquence de caractères (qui n'est pas forcément représentée de la même façon), `false` autrement. L'efficacité de votre implantation ne sera pas prise en compte. Il peut être utile d'utiliser des méthodes de la partie 1 dans votre définition.

```
def equal(t1: Text, t2: Text): Boolean =
  if (t1 isEmpty) t2 isEmpty
  else !t2.isEmpty && t1.head == t2.head && equal(t1.tail, t2.tail)
```

Exercice 3 : Preuve inductive (20 points)

Prouvez, par induction structurelle sur la variable `xs`, l'égalité suivante.

```
unlines(lines(xs)) = xs
```

Justifiez chaque étape en vous référant uniquement aux lemmes et définitions suivants.

```
    List() ::: ys = ys                                  // app1
  (x :: xs) ::: ys = x :: (xs ::: ys)                   // app2

def lines(chars: List[Char]): List[List[Char]] = chars match {
  case List() => List(List())                           // lines1
  case ch :: cs if ch == '\n' => List() :: lines(cs)    // lines2
  case ch :: cs =>                                      // lines3
    val l :: lss = lines(cs)
    (ch :: l) :: lss
}

def unlines(ls: List[List[Char]]): List[Char] = ls match {
  case List() => List()                                 // unlines1
  case List(lastLine) => lastLine                       // unlines2
  case l :: lss => l ::: ('\n' :: unlines(lss))         // unlines3
}
```

Une expression de la forme **`val`** `a = b` se traduira dans la preuve en postulant explicitement l'égalité de `a` et de `b` dans les étapes suivantes.
SOLUTION

```
INDUCTION BASE
unlines(lines(List()))
= unlines(List(List())) // lines1
= List()                // unlines2

INDUCTION STEP
unlines(lines(x :: xs))
// case x == '\n':
= unlines(List() :: lines(xs))            // lines2
= List() ::: ('\n' :: unlines(lines(xs))) // unlines3 since lines(xs) != List()
= List() ::: ('\n' :: xs)                 // IH
= x :: xs                                 // app1

// case x != '\n':
= unlines((x :: l) :: lss) with l :: lss = lines(xs) // lines3

// subcase lss == List(), i.e. List(l) = lines(xs)
= x :: l                                  // unlines2
IH: unlines(lines(xs)) = xs <=> unlines(List(l)) = xs => l = xs (unlines2)
= x :: xs

// subcase lss != List()
= (x :: l) ::: ('\n' :: unlines(lss))     // unlines3
= x :: (l ::: ('\n' :: unlines(lss)))     // app2
= x :: unlines(l :: lss)                  // unlines3 backward
= x :: unlines(lines(xs))                 // expand
= x :: xs                                 // IH
```