## Recitation Session 8

**Please do not write on this sheet of paper**
**And do not use laptops during the session**

Recall the definition of a `Monad` from the lecture. We say that a type `M` is a `Monad` if `M[T]` has a `flatMap` method with the following signature:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}
```

And there is a `unit` method for `M` with the following signature:

```
def unit[T](x: T): M[T]
```

Such that `flatMap` and `unit` fulfill the following laws:

*Left unit:*
`unit(x).flatMap(f) === f(x)`

*Right unit:*
`m.flatMap(unit) == m`

*Associativity:*
`m.flatMap(f).flatMap(g) === m.flatMap(x => f(x).flatMap(g))`

Consider the following definition of a list:

```
sealed trait IList[T] {
  def flatMap[U](f: T => IList[U]): IList[U] =
    this match {
      case INil() => INil()
      case ICons(h, t) => f(h) ++ t.flatMap(f)
    }

  def ++(that: IList[T]): IList[T] =
    this match {
      case INil() => that
      case ICons(h, t) => ICons(h, t ++ that)
    }

  def map[U](f: T => U): IList[U] =
    this match {
      case INil() => INil()
      case ICons(h, t) => ICons(f(h), t.map(f))
    }
}

object IList {
  def singleton[T](x: T): IList[T] = ICons(x, INil())
}

case class INil[T]() extends IList[T]

case class ICons[T](h: T, t: IList[T])  extends IList[T]
```

Prove that `IList` is a `Monad` for `unit = IList.singleton`. You can assume associativity of concatenation (++), that is, for all `a: IList[A]`, `b: IList[A]`, `c: IList[A]`,

```
a ++ (b ++ c) === (a ++ b) ++ c
```