Ficha 4

Exercício 1

1.1)

and)

```
def andFR(xs : List[Boolean]) = (xs foldRight true) (_ && _)

def andFL(xs: List[Boolean]) = (xs foldLeft true) (_ && _)
```

or)

```
def orFR(xs : List[Boolean]) = (xs foldRight false) (_ || _)

def orFL(xs: List[Boolean]) = (xs foldLeft false) (_ || _)
```

concat)

```
def concatFR[A](xs : List[A], ys : List[A]) = (xs foldRight ys) (_
::_)

def concatFL[A](xs : List[A], ys : List[A]) = (ys foldLeft xs) (
(lst,y) => lst ++ List(y))
```

OU

```
def concatFL[A](xs: List[A], ys: List[A]): List[A] = (ys foldLeft
xs) (_:+_)
```

1.2)

```
def remDup[A](ls: List[A]): List[A] = (ls foldRight List[A]())
((x,xs) => x::(xs.dropWhile(_ == x)))
```

Exercício 2

a)

```
def same(m: Match): Boolean = m._1._1.equals(m._2._1)

def noItself( j: Fixtures) : Boolean = { (j foldRight true)
((m1,m2)=> !same(m1) && m2) }
```

b)

```
def existCount(t: Team, j:Fixtures): Int =
{
  (j foldRight 0) ( (m1,m2)=>
    if((t.equals(m1._1._1) || t.equals(m1._2._1)))
      1+m2
    else
   m2)
}
```

```
def withoutRep(j : Fixtures) : Boolean =
{
  (j foldRight true) ( (m1,m2)=>
    if(existCount(m1._1._1,j)>1 || existCount(m1._2._1,j)>1)
      false
  else
   m2)
}
```

c)

```
def teams(j: Fixtures): List[Team] = { (j foldRight List[Team]())
((m,j) => (m._1._1):: (m._2._1)::j)}
```

d)

```
def draws(j: Fixtures): List[(Team, Team)] =
{
  (j foldRight List[(Team, Team)]()) ( (x, j) =>
    if(x._1._2 == x._2._2)
      ((x._1._1),(x._2._1))::j
    else
   j)
}
```

e)

```
def points(j: Fixtures):List[(Team, Int)] =
{
  (j foldRight List[(Team, Int)]()) ( (x, j) =>
  {
    if(x._1._2 == x._2._2)
      ( (x._1._1),1)::( (x._2._1),1 )::j
    else if(x._1._2 < x._2._2)
      ((x._1._1),0)::( (x._2._1),3)::j
    else
   ((x._1._1),3)::( (x._2._1),0)::j})
}
```

OU

```
def points2(j: Fixtures):List[(Team, Int)] =
{
  (j foldRight List[(Team, Int)]()) ( (x, j) =>
  {
    x match
  {
      case a if(x._1._2 == x._2._2) => ( (x._1._1),1)::(
(x._2._1),1 )::j
      case a if(x._1._2 < x._2._2) => ((x._1._1),0)::(
```

```
(x._2._1),3)::j
        case _ => ((x._1._1),3)::( (x._2._1),0)::j
      }
    })
}
```

Exercício 3

a)

```
def isort(l:Pol) : Pol =
{
   l match {
      case Nil => Nil
   case x::xs => insert(x, isort(xs))
   }
}
def insert(x:(Float, Int), lst:Pol): Pol =
{
   lst match{
      case Nil => List(x)
      case y::ys => if(x._2 < y._2) x::y::ys else y :: insert(x, ys)
   }
}
```

b)

```
def norm(p: Pol): Pol =
{
   val max = isort(p).last._1
   p map ( x => ((x._1 / max), x._2))
}
```

//Com recursividade explícita

```
def normRec(p: Pol): Pol =
{
  def auxMap(p1:Pol, f: ((Float, Int)) => (Float, Int) ): Pol = p1 match {
    case Nil => Nil
    case x::xs => f(x)::auxMap(xs, f)
  }
  val max = isort(p).last._1
  def fun(x:(Float, Int)):(Float, Int) =  ((x._1 / max), x._2)
  auxMap( p , fun)
}
```

c)

```
def addPol(xs: Pol, ys: Pol): Pol =
{
   (xs foldRight ys) ( (x,xs) =>
     if(!containsElemSameDegree(x, ys))
        x::xs
     else
```

```
    addSameDegree(x, xs) )
}
def addSameDegree( e:(Float,Int) , p: Pol ): Pol =
{
   (p foldRight List[(Float, Int)]() ) ( (x,p) =>
     if( x._2 == e._2 )
        ((x._1 + e._1, x._2)::p)
     else
    x::p )
}
def containsElemSameDegree(e:(Float,Int) , p: Pol): Boolean =
{
   (p foldRight false) ((x,p) =>
     if(x._2 == e._2)
        true
   else
    p )
}
```

//Com recursividade explícita

```
def addPol_RecExp(xs: Pol, ys: Pol): Pol = xs match {
  case Nil => ys
  case (x::t) =>  if(!containsElemSameDegree_RecExp(x, ys)) x::addPol_RecExp(t, ys)
  else addSameDegree_RecExp(x, addPol_RecExp(t,ys))
}

def addSameDegree_RecExp( e:(Float,Int) , p: Pol ): Pol = p match {
  case Nil => List[(Float, Int)]()
  case (x::t) =>  if(x._2 == e._2) ((x._1 + e._1, x._2)::addSameDegree_RecExp(e, t))
  else x::addSameDegree_RecExp(e, t)
}

def containsElemSameDegree_RecExp(e:(Float,Int) , p: Pol): Boolean  = p match {
  case Nil => false
  case x::t => if(x._2 == e._2) true else containsElemSameDegree_RecExp(e, t)
}
```

d)

```
def valuePol(v:  Float, p:Pol):  Float =
{
   (p foldRight 0.toFloat) ( (x,p) => (x._1 *
Math.pow(v,x._2.toFloat)).toFloat + p)
}
```

//Com recursividade explícita

```
def valuePol_RecExp(v: Float, p:Pol): Float = p match {
  case Nil => 0.toFloat
  case x::t => (x._1 * Math.pow(v,x._2.toFloat)).toFloat + valuePol_RecExp(v, t)
}
```

e)

```
def degreePol(p: Pol)  : Int = isort(p).last._2
```

f)

```
def derivatePol(p : Pol): Pol = p map ( x =>
   if(x._2 == 0)
      (0,0)
   else
   (x._2 * x._1,x._2 - 1) )
```

//Com recursividade explícita

```
def derivatePol_RecExp(p : Pol): Pol = p match {
  case Nil => Nil
  case x::t => if(x._2 == 0) (0f,0)::derivatePol_RecExp(t)
           else (x._2 * x._1, x._2 - 1 )::derivatePol_RecExp(t)
}
```

Exercício 4

1)

```
def merge(l1: List[Int], l2: List[Int]) : List[Int] =
{
   ( l1 foldRight l2) ( (x,t) => insert(x,t))
}
```

2)

```
def isort(l: List[Int]): List[Int] = (l foldRight List[Int]()) ( (x,t)
=> insert(x,t) )
```

3)

```
import scala.math.Ordering
def isortP[T](l: List[T])(implicit ord: Ordering[T]): List[T] =
{
   def insert(x: T, lst:List[T]): List[T] =
   {
      lst match{
         case Nil => List(x)
         case y::ys =>
            if(ord.lt(x,y))
               x::y::ys
            else
       y :: insert(x,ys)
      }
   }
   (l foldRight List[T]()) ( (x,l) => insert(x,l) )
}
```

Exercício 5

```
def merge(l1: List[Int])(l2: List[Int]) : List[Int] =
{
  (l1 foldRight l2) ( (x, l1) => insert(x, l1))
}
val mergeWith00 = merge(List(0,0)) _
List(List(1,2),List(3,4)) map merge(List(0,0))
List(List(1,2),List(3,4)) map mergeWith00
```

Exercício 6

```
def separate[A](lst:List[A]): (List[A],List[A]) = (lst
foldRight(List[A](),List[A]())) (
  (a, l) =>
    if((l._1.length + l._2.length)%2!=0)
       (a::l._1, l._2)
    else
   (l._1, a::l._2))
```

or

def **separate**[A](lst:List[A]): (List[A],List[A]) = (lst foldRight (List[A](),List[A]())) ((a, l) => if(l._1.length < l._2.length) (a::l._1, l._2) else (l._1, a::l._2))