Exercise 1: Multiset (10 points)

Implementing a functional multiset

```scala
def emptyMultiSet: MultiSet = { y => 0 }
def singleton(x: Int): MultiSet = { y =>
  if (y == x) 1 else 0
}
def union(a: MultiSet, b: MultiSet): MultiSet = { y =>
  a(y) + b(y)
}
def intersect(a: MultiSet, b: MultiSet): MultiSet = { y =>
  min(a(y), b(y))
}
def diff(a: MultiSet, b: MultiSet): MultiSet = { y =>
  max(a(y) - b(y), 0)
}
```

Using a functional multiset

```scala
def primeFactors(n: Int): MultiSet = {
  def rec(i: Int, n: Int): MultiSet = {
    (i until n).find{ n % _ == 0 } match {
      case None => singleton(n)
      case Some(x) => union(singleton(x), rec(x, n/x))
    }
  }
  rec(2, n)
}
```

Exercise 2: Monads (10 points)

Left unit (3 points)

Show that `unit(x) flatMap f == f(x)` holds for lists.
unit(x) flatMap f
==
  x :: Nil flatMap f   // by 5
==
  f(x) ++ Nil flatMap f // by 1
==
  f(x) ++ Nil     // by 2
==
  f(x)            // by 6

Right unit (3 points)

Show that m flatMap unit == m holds for lists.
We will show this by structural induction on m.
case m is Nil: Nil flatMap unit == Nil // by 2
case m is x :: xs: Induction hypothesis is that xs flatMap unit == xs holds for some size n. We show it holds for size n + 1.

  x :: xs flatMap unit
==
  unit(x) ++ xs flatMap unit   // by 1
==
  (x :: Nil) ++ xs flatMap unit // by 5
==
  (x :: Nil) ++ xs     // by induction hypothesis
==
  x :: (Nil ++ xs)     // by 4
==
  x :: xs                        // by 3

Associativity (4 points)

Show that m flatMap f flatMap g == m flatMap (x => f(x) flatMap g) holds for lists.
We again do a proof by structural induction on m.
case m is Nil:

    Nil flatMap f flatMap g
==
    Nil flatMap g     // by 2
==
    Nil

case m is x :: xs:
We first expand the RHS to

    m flatMap (x => f(x) flatMap g)
==
    f(x) flatMap g ++ xs.flatMap(x => f(x) flatMap g)     // by 1
The induction hypothesis is that for some size n it holds: $xs \text{ flatMap } f \text{ flatMap } g == xs \text{ flatMap ( } x \Rightarrow f(x) \text{ flatMap } g )$

\[
x :: xs \text{ flatMap } f \text{ flatMap } g
==
(f(x) ++ xs \text{ flatMap } f ) \text{ flatMap } g \quad // \text{ by 1}
==
(f(x) \text{ flatMap } g) ++ ( xs \text{ flatMap } f \text{ flatMap } g ) \quad // \text{ by 7}
==
f(x) \text{ flatMap } g ++ xs \text{ flatMap ( } x \Rightarrow f(x) \text{ flatMap } g ) // \text{ by induction hypothesis}
\]

which is the same as the expanded RHS, so we’re done.

**Exercise 3: Comprehending Observables (10 points)**

```scala
def oneOf[T](ls: List[T]): Generator[T] = 
  for (i <- choose(0, ls.length)) yield ls(i)
```

Separating chocolate kinds (3 points)

```scala
val chocolatesByKind: Observable[(String, Observable[Chocolate])] = chocolateChannel.groupBy (_.kind)
```

Bunching chocolates together (3 points)

```scala
val chocolatesBunched: Observable[Observable[Bunch]] = for (
  (kind, chocolateStream) <- obs
) yield chocolateStream.buffer(chocolateNumbers(kind))
```

Making packets

```scala
val chocolatePackets: Observable[Packet] = Observable.zip(chocolatesBunched)
```

**Exercise 4: Batch Logging using Actors (010 points)**

Publisher behavior

```scala
class Publisher extends Actor {
  import Publisher._
  var logger = Set[ActorRef]()

  def receive = {
    case Subscribe => logger += sender
    case Unsubscribe =>
      logger -= sender
      println("logger "+ sender + " just quit!")
  }
}
```
case Update(msg, l) =>
  logger.foreach(log => log ! LogEntry(msg, l))
}

Logger behavior

class Logger(collector: ActorRef, pub: ActorRef, debugLevel: Int)
  extends Actor {
    import Publisher._
    import Logger._

    var log = Seq[String]()

    pub ! Subscribe

    def receive = {
      case LogEntry(msg, l) =>
        if (l > debugLevel) log = log :+ msg
        if (log.length > 42) {
          collector ! LogFull(log)
          log = Seq()
        }

      case StopLogging =>
        pub ! Unsubscribe
    }
  }