NAME: SUSIE AGERHOLM BALLE
ITU EMAIL: SABA@ITU.DK

QUESTION 1

Task 1

```
def checksumFun(in: String): Int =
   in.toList.foldLeft(0)((acc, a) => (acc + a.toInt) % 0xffff)
```

Task 2

A foldLeft operation is tailrecursive by definition while a foldRight will not be. Because you have direct access to the next item on list foldLeft computation of part results can happen while doing the fold. This is not the case in a foldRight operation, where computation of all the individual parts have to be postponed and placed on the stack untill the whole list has been folded.

foldLeft stack trace: (((z + x1) + x2) +x3)
foldRigth stack trace: (z: (x1 + (x2 + x3)))

QUESTION 2

Task 3

```
object test {

  import fpinscala.monads._
  import scala.language.higherKinds

  def onList[A] (f: A => A): List[A] => List[A] =
     { xs => xs.map(x => f(x)) }

  println(onList({x: Int => x + 1})(List(1,2,3,4)))       //> List(2, 3, 4, 5)


}
```

Task 4

```
def onCollection[C[_], A] (f: A => A)(functorC: Functor[C]): C[A] => C[A] =
    { xs => functorC.map(xs)(f) }

//> onCollection: [C[_], A](f: A => A)(functorC: fpinscala.monads.Functor[C])C[A
      //| ] => C[A]
```

QUESTION 3

Task 5

```
 def foldBack[A](l: List[A])(implicit M: Monoid[A]): A =
    (l ++ l.reverse).foldRight (M.zero) (M.op)


 //> foldBack: [A](l: List[A])(implicit M: fpinscala.monoids.Monoid[A])A
```

```
  val intAddition = new Monoid[Int] {
          def op(a1: Int, a2: Int) = a1 + a2  //compose
          val zero = 0
  }

//> intAddition  : fpinscala.monoids.Monoid[Int]{val zero: Int} = test$$anonfun$

//| main$1$$anon$1@5ebec15
println(foldBack(List(1,2,3))(intAddition))     //> 12
```

QUESTION 4

Task 6

```
 def run[A] (init: A)(progs: List[Computation[A]]): (A, List[String]) = {
    def helper(init: A)(progs: List[Computation[A]])(last: (A, List[String])): (A, List[String]) = progs
match {
      case h :: _ => { a: A => h(a)* match {
         case Left(a) => helper(a)(progs.tail)(a, last._2)
         case Right(b) => last match {
           case (a, h1::t1) => helper(last._1)(progs.tail)(a, b::h1::t1)
           case (a, Nil) => helper(last._1)(progs.tail)(a, List(b))
         }
       }
      }
      case Nil => last
    }
    helper(init)(progs)((init, List()))
  }
```

*I had a problem with the syntax for applying the computation before pattern match but did not have time to fix this issue unfortunately...

QUESTION 5

Task 7
//

Task 8

The function should of course be tested on both finite and infine trees in order to secure, that also processing of infinite trees will terminate appropriately.

Some protection against too eager evaluation comes from the fact that the Branch constructor arguments are call by name – this means that the arguments are not evaluated on function call but only when it is actually called inside the function. This should be taken advantage of when writing the multiply function.

QUESTION 6

Task 9

val zero: Zero
val one: Succ[Zero]
val two: Succ[Succ[Zero]]

Task 10

//