These questions are intended for self-study, to help review and deepen your understanding of the lecture. Sample answers are available. There is nothing to hand in.

1. Rewrite the following code so that it does not use nested functions, by lifting g and h to be top-level functions and adding their free variables as extra explicit parameters.

   ```scala
   def f (a:Int) = {
     def g (b:Int) = {
       def h (c: Int) = a+b+c
       h(1) + h(2)
     }
     g (a + 10)
   }
   ```

2. Consider the following code defining an insertion sort function. Show how to use this function to define two more specialized functions `sortup:List[Int] => List[Int]` and `sortdown:List[Int] => List[Int]` that sort a list in ascending and descending order, respectively, using anonymous functions to instantiate the `comp` parameter.

   ```scala
   def insSort (comp: (Int, Int) => Boolean) (ys:List[Int]) : List[Int] = {
     def ins (x:Int,xs:List[Int]) : List[Int] = xs match {
       case Nil => List(x)
       case h::t => if (comp (x,h)) x::h::t else h::(ins(x,t))
     }
     def srt (zs:List[Int]) : List[Int] = zs match {
       case Nil => Nil
       case h::t => ins(h,srt(t))
     }
     srt(ys)
   }
   ```

3. Define the following function by applying the `map` function defined on slide 12.

   (a) `above(n:Int) : List[Int] => List [(Int,Boolean)]`

   where `above(n)` pairs each member of a list of integers with a boolean indicating whether it is greater than `n`. For example `above (3) (List(1,2,4,3,5))` returns `List((1,false), (2,false), (4,true), (3,false), (5,true))`. Use an anonymous function argument.

   (b) `sumeach : List[List[Int]] => List[Int]`

   which takes a list of lists of integers and returns a list of integers representing the sums of the original nested lists. For example `sumeach (List(List(1,2),List(4,5,6)))` returns `List(3,15)`. Use the `sum` function from slide 13.
4. Use the Scala library’s version of `foldr` to implement the following functions, without using any imperative features or explicit recursion. Scala’s library writes this operator as `:` , and it takes its arguments in a slightly strange order: what slide 14 shows as `foldr (c, n) l` is written in Scala as `(l :\ n) c` so, for example, we can sum the elements of the list 1,2,3 by writing `(List(1,2,3) :\ 0) ((x,a) => a + x)`. Note: Consult the Scala List API and the Scala book chapter on “Working with Lists” for guidance and examples. Warning: in the online version (at least) of the Scala book there is a consistent type-setting error: the string “:\” incorrectly appears as “:\~”.

(a) `concat(xs: List[String]) : String` returns the concatenation of the strings in list `xs` into a single string.

(b) `max(xs: List[Int]) : Int` returns the maximum integer in list `xs`, which you can assume to be non-empty.

(c) `unzip[A,B](xys: List[(A,B)]) : (List(A), List(B))` takes a list `xys` of pairs and returns a pair consisting of a list of the first elements of `xys` and a list of the second elements of `xys`. 