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. (a) Recall the (non-tail-recursive) definition of the factorial function from the last lecture:

```scala
def fac(x:Int) : Int = {
  if (x < 2)
    1
  else
    x*fac(x-1)
}
```

Use the recursion removal techniques described in lecture 4b to convert this to a non-recursive Scala function using an explicit stack. It is easiest to develop a solution by first temporarily pretending that you have labels and gotos available in Scala, and then converting the resulting code to use structured control operators like `while`.

(b) Do the same with the Fibonacci function:

```scala
def fib(x:Int) : Int = {
  if (x < 2)
    x
  else
    fib(x-1) + fib(x-2)
}
```

Note: This example is much more complicated! In general, each entry in the explicit stack needs to encode which recursive call is to be “returned to,” and the values of any variables that are defined before the call and used after it. For `fib`, there are two “return points” corresponding to the two recursive calls. After returning from the first call we need to remember the value of `n` over the first call, and the value of `t1` over the second call. A suitable Scala data type for stack elements is therefore:

```scala
sealed abstract class C
case class C1(val n : Int) extends C
case class C2(val t1 : Int) extends C
```
2. The following Scala program uses exceptions to propagate an error condition. Rewrite the program to use the Maybe type instead following the model on slide 6.

```scala
case object Failure extends Exception

def f (a:Int) : Int = 
  if (a > 10) a-10 else throw Failure

def g (b:Int) : Int = 
  f(b+b) + 10

def h (c:Int) = 
  try {
    println(g(c)*5)
  } catch {
    case Failure => println("oops")
  }
```

3. Consider a function `samevals(t_1, t_2)` that takes two binary trees, not necessarily of the same shape, and returns a boolean saying whether or not their values as produced by an in-order traversal are the same. For example, `samevals` should return true for these trees:

```
       6
      / \
     9   1
    / \   \
   7   4   \
```

It is surprisingly difficult to write this function using ordinary recursive procedures (try it!). One simple approach is to extract the values from each tree into a list and then compare the lists, but this always requires $O(n)$ extra storage and $O(n)$ time, even if there is a mismatch in values early on.

But the problem (a variant of the well-known “same fringe” problem) is easy to solve using coroutines. Write a version of the `samevals` function using the facilities of Lua coroutines that are illustrated on lecture 5a slice 16, including the `walk` function. Pseudo-code is fine; there is no need to get the details of Lua syntax right—although if you want to try, Lua is very easy to learn: there’s an online book at https://www.lua.org/pil/contents.html and you can test your code at https://www.lua.org/cgi-bin/demo.