1. (a) It is best to start by making intermediate results and control flow as explicit as possible.

```scala
def fac(x:Int) :Int = {
    var r = 0
    if (x < 2)
        r = 1
    else {
        val t = fac(x-1)
        r = x * t
    }
    return r
}
```

Now we can convert to an (invalid Scala) version with explicit labels and `goto`s and an explicit stack. Here the stack only needs to remember the old value of `x` when we make the recursive call.

```scala
def fac(x0:Int) :Int = {
    var x = x0
    val stack:Stack[Int] = Stack()
    var r
    top:
    if (x < 2) {
        r = 1
        goto ret
    } else {
        stack.push(x)
        x -= 1
        goto top
    }
    rp:
    t = r
    r = x * t
    goto ret
    ret:
    if (stack.nonEmpty) {
        x = stack.pop()
        goto rp
    } else
        return r
}
```

Finally, we can do a little rearrangement to get rid of the label `rp` and introduce suitable `while` loops to get rid of `top` and `ret`:

```scala
def fac(x0:Int) :Int = {
    var x = x0
    var r = 0
    if (x < 2)
        r = 1
    else {
        while (x > 1) {
            val t = fac(x-1)
            r = x * t
            x -= 1
        }
    }
    return r
}
```
val stack:Stack[Int] = Stack()
var r = 0
while (x >= 2) {
  stack.push(x)
  x -= 1
}
r = 1
while (stack.nonEmpty) {
  x = stack.pop()
  r = x * r
}
return r
}

This version makes it clear how recursive factorial works (and how inefficiently it uses space!)

(b) Again, it is best to start by making intermediate results and control flow as explicit as possible.

def fib (n: Int) : Int = {
  var r = 0
  if (n < 2)
    r = n
  else {
    val t1 = fib(n-1)
    val t2 = fib(n-2)
    r = t1 + t2
  }
  return r
}

This can be converted directly to an (invalid Scala) version with explicit labels and gotos.

int fib (int n0) {
  var n = n0
  var r : Int
  var t1 : Int
  var t2 : Int
  val stack : Stack[C] = Stack()
  top:
  if (n < 2)
    r = n;
  else {
    stack.push(C1(n))
    n -= 1
goto top
  rp1:
    t1 = r
    stack.push(C2(t1))
    n -= 2
goto top
  rp2:
t2 = r
r = t1 + t2
}
ret:
if stack.nonEmpty {
    stack.pop() match {
      case C1(oldn) => {
        n = oldn
        goto rp1
      }
      case C2(oldt1) => {
        t1 = oldt1
        goto rp2
      }
    }
    return r
}

After some rearrangement:

int fib (int n0) {
    var n = n0
    var r : Int
    var t1 : Int
    var t2 : Int
    val stack : Stack[C] = Stack()

    top:
        while (n >= 2) {
            stack.push(C1(n))
            n -= 1
        }
    r = n
    ret:
        while (stack.nonEmpty) {
            stack.pop() match {
              case C1(oldn) => {
                n = oldn
                t1 = r
                stack.push(C2(t1))
                n -= 2
                goto top
              }
              case C2(oldt1) => {
                t1 = oldt1
                t2 = r
                r = t1 + t2
              }
            }
        }
    return r
And finally we add a flag (since Scala lacks a built-in break or continue) to let us rewrite into legal Scala:

```scala
def fib(n0:Int) : Int = {
  var n = n0
  var r = 0
  val stack : Stack[C] = Stack()
  var flag = true
  while (flag) {
    while (n >= 2) {
      stack.push(C1(n))
      n -= 1
    }
    r = n
    flag = false
    while (!flag && stack.nonEmpty) {
      stack.pop() match {
        case C1(oldn) => {
          stack.push(C2(r))
          n = oldn - 2
          flag = true
        }
        case C2(oldt1) =>
        r += oldt1
      }
    }
  }
  return r;
}
```

Or, perhaps a bit clearer (at the expense of some duplicated code):

```scala
def fib(n0:Int) : Int = {
  var n = n0
  var r = 0
  val stack : Stack[C] = Stack()
  while (n >= 2) {
    stack.push(C1(n))
    n -= 1
  }
  r = n
  while (stack.nonEmpty) {
    println(stack)
    stack.pop() match {
      case C1(oldn) => {
        stack.push(C2(r))
        n = oldn - 2
        flag = true
      }
      case C2(oldt1) =>
      r += oldt1
    }
  }
  return r;
}
```
n -= 1
}
r = n }
case C2(t) => {
r += t
}
}
return r;
}

2.

def f (a:Int) : Option[Int] =
    if (a > 10) Some(a-10) else None

def g (b:Int) : Option[Int] =
    f(b+b) match {
        case Some(v) => Some(v+10)
        case None => None
    }

def h (c:Int) =
    g(c) match {
        case Some(x) => println (x*5)
        case None => println("oops")
    }

3. Here’s a Lua solution using the facilities shown on the slides. (It can be made shorter and neater using other Lua features that were not introduced.)

function samevals (t1,t2)
    function cofun1 () walk (t1) end
    local co1 = coroutine.create (cofun1)
    function cofun2 () walk (t2) end
    local co2 = coroutine.create (cofun2)
    repeat
        local _, v1 = coroutine.resume(co1)
        local _, v2 = coroutine.resume(co2)
        if (v1 ~= v2) then
            return false
        end
    until v1 == nil
    return true
end