**CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 4c**

1. Function `s` and `t` are not tail-recursive, as they each perform an addition after the return of the recursive call. (The fact that the recursive call comes last on the line in `s` makes no difference; it's the order of operations that counts.)

Function `even` is tail-recursive, and can be rewritten thus:

```
def even(x:Int) : Boolean = {
  var y = x
  while (y > 1) { y = y - 2 }
  return (y == 0)
 }
```

(We have to assign `x` to a new `var` before we can change it, because Scala parameters are always immutable `val`s. Notice the convenience of returning the value of a boolean expression directly. The `return` keyword is not strictly necessary.)

`fac` is not tail-recursive, since it performs a multiplication after the return of the recursive call.

`facn` is tail-recursive, and can be rewritten thus:

```
def facn(x:Int,y:Int) = {
  var x1 = x
  var y1 = y
  while (x1 >= 2) {
    y1 = x1 * y1  // need to do this first to avoid overwriting x1 too soon
    x1 = x1 - 1
  }
  y1
}
```

`fib`, `g`, and `h` are not tail-recursive, since each has at least one recursive call that is followed by further computation within the function before it returns.

2.

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

```
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.

```
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 rearrangment to get rid of the label `rp` and introduce suitable `while` loops to get rid of `top` and `ret`:

```
def fac(x0:Int) :Int = {
  var x = x0
  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) Starting from the version with explicit intermediate results and control flow and the definition of `C`, we can convert directly to an (invalid Scala) version with explicit labels and `goto`s.

```
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:

```
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):

```
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) {
    stack.pop() match {
      case C1(oldn) => {
        stack.push(C2(r))
        n = oldn - 2
        while (n >= 2) {
          stack.push(C1(n))
          n -= 1
        }
        r = n
      }
      case C2(t) =>
        r += t
    }
  }
  return r;
}
```