TUTORIAL 3

Question 2 in the exam paper focuses on functions and their use in a functional approach. The first part of the question typically involves a discussion of recursion.

1. This Scala function calculates the factorial of an integer using recursion:

```
def factorial(n: Int): Int = {
  if (n <= 1)
     n
  else
     factorial(n - 1) * n
}
```

It be called like this to find the factorial of 4:

```
factorial(4)
```

You can trace the way the function works by tracing the recursive calls and their return values using a table like the one below. The function calls are made recursively until the stopping condition is reached, then the last call returns its value to the next-to-last, and so on.

Function call	Stopping condition (n<=1)	Returned value
factorial(4)	false	3*4 = 24
factorial(3)	false	2*3 = 6
factorial(2)	false	1*2 = 2
factorial(1)	true	1

Similarly, this Scala function reverses a string using recursion:

```
def reverse(str:String): String = {
  if (str == null || str.equals(""))
     str
  else
     reverse(str.substring(1)) + str.substring(0, 1)
}
```

Calling:

```
reverse("hello")
```

gives:

```
"olleh"
```

Trace the way this function works by writing out a table similar to the factorial example

2. This Scala function is a tail-recursive version of the *factorial* function.

```
def factorial (n: Int): Int = {

  def fact(n: Int, accumulator: Int): Int = {
    if (n <= 1)
      accumulator
    else
      fact(n - 1, n* accumulator)
  }

  fact(n 1)
}
```

a. What is the advantage of using tail-recursive functions?
b. How can you tell from the code that the function is tail-recursive?
c. Trace the way this function works by writing out a table similar to the *factorial* example in Q1. Add an extra column in the table to trace the value of the variable *accumulator* within each call. Deduce the purpose of *accumulator*.

3. How could you modify the *reverse* function to make it tail-recursive?