5 Compiler Construction (TGG)

Functional programmers will often rewrite a recursive function such as

```
fun fact1 n =
    if n <= 1
    then 1
    else n * (fact1 (n -1))
```

to one such as

```
fun fact2 n =
let fun aux (m, a) =
        if m <= 1
        then a
        else aux(m-1, m * a)
in aux (n, 1) end
```

using an accumulator (the parameter `a` of `aux`) and *tail recursion.*

(*a*) Clearly explain the optimisation such programmers are expecting from the compiler and how that optimisation might improve performance. [4 marks]

(*b*) The desired optimisation can be performed by a compiler either directly on the source program or on lower-level intermediate representations. Treating it as a source-to-source transformation, rewrite `fact2` to ML code that has been transformed by this optimisation. You will probably use references and assignments as well as the construct `while EXP do EXP`. [8 marks]

(*c*) Suppose that the programmer used instead a function as an accumulator.

```
fun fact3 n =
let fun aux (m, h) =
        if m <= 1
        then h(1)
        else aux(m-1, fn r => m * (h r))
in aux (n, fn x => x) end
```

Will your optimisation still work in this case? Explain your answer in detail. [8 marks]