— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2013 – Paper 1

2 Foundations of Computer Science (LCP)

algorithms, lists, curried functions, higher-order functions

The function `perms` returns all $n!$ permutations of a given n-element list.

```
fun cons x y = x::y;

fun perms [] = [[]]
  | perms xs =
      let fun perms1 ([],ys) = []
            | perms1 (x::xs,ys) =
                map (cons x) (perms (rev ys @ xs)) @
                perms1 (xs,x::ys)
      in  perms1 (xs,[])  end;
```

(a) Explain the ideas behind this code, including the function `perms1` and the expression `map (cons x)`. What value is returned by `perms [1,2,3]`?

[7 marks]

Answer: The base case is `[[]]` because the empty list has one permutation, namely `[]`. The idea of the code is that the permutations of a list containing some element x consist of (a) those that begin with x, the tail computed by a recursive call, and (b) those that do not begin with x. The function `perms1` walks down a list, choosing successive list elements to play the role of x above. The expression `map (cons x)` modifies the list of permutations obtained from the recursive call by inserting x as the first element of each. Here, `cons` is a curried function.

```
perms [1,2,3] =
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

---

lazy lists

(b) A student modifies `perms` to use an ML type of lazy lists, where `appendq` and `mapq` are lazy list analogues of `@` and `map`.

```
fun lperms [] = Cons ([], fn() => Nil)
  | lperms xs =
      let fun perms1 ([],ys) = Nil
            | perms1 (x::xs,ys) =
          appendq (mapq (cons x) (lperms (rev ys @ xs)),
                   perms1 (xs,x::ys))
      in  perms1 (xs,[])  end;
```

Unfortunately, `lperms` computes all $n!$ permutations as soon as it is called. Describe how lazy lists are implemented in ML and explain why laziness is not achieved here. [5 marks]

Answer: ML does not have a proper lazy evaluation mechanism. Lazy lists can be simulated using the following datatype declaration:

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);
```

Laziness can be obtained through writing functions of the form `fn() => E`, for then the expression E is not evaluated until the function is called, with argument `()`.

Thw function above uses lazy list primitives correctly as regards types, but the only occurrence of `fn() =>` protects an instance of `Nil`. All recursive calls to `lperms` take place when the function is called, and therefore all permutations are computed.

---

lazy lists

(c) Modify the function `lperms`, without changing its type, so that it computes permutations upon demand rather than all at once. [8 marks]

Answer: The trick is to insert an occurrence of `fn() =>` within the recursive calls. One way of doing this is by modifying the function `mapq`. There are other solutions.

```
fun mapq2 f Nil yf  = yf()
  | mapq2 f (Cons(x,xf)) yf = Cons(f x, fn()=> mapq2 f (xf()) yf);

fun lperms [] = Cons ([], fn() => Nil)
  | lperms xs =
      let fun perms1 ([],ys) = Nil
            | perms1 (x::xs,ys) =
                mapq2 (cons x) (lperms (rev ys @ xs))
                      (fn() => perms1 (xs,x::ys))
      in  perms1 (xs,[])  end;
```

All ML code must be explained clearly and should be free of needless complexity.