— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2013 – Paper 1

2 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

algorithms, lists, curried functions, higher-order functions

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

```
let rec perms = function
| [] -> [[]]
| xs ->
    let rec perms1 xs ys =
      match xs with
      | [] -> []
      | x::xs ->
          List.map (List.cons x) (perms (List.rev ys @ xs)) @
                   perms1 xs (x::ys)
    in
      perms1 xs []
```

(*a*) Explain the ideas behind this code, including the function `perms1` and the expression `List.map (List.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 `List.map (List.cons x)` modifies the list of permutations obtained from the recursive call by inserting x as the first element of each. Here, `List.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 OCaml type of lazy lists, where `appendq` and `mapq` are lazy list analogues of `@` and `List.map`.

```
let rec lperms = function
| [] -> Cons ([], fun () -> Nil)
| xs ->
    let rec fun perms1 xs ys = function
    | [] -> Nil
    | x::xs ->
        appendq (mapq (List.cons x) (lperms (List.rev ys @ xs)))
                (perms1 xs (x::ys))
    in
      perms1 xs []
```

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

Answer: OCaml's lazy values do not form part of the syllabus. Lazy lists can be simulated using the following variant type declaration:

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

Laziness can be obtained through writing functions of the form `fun () ->` 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 `fun () ->` 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 `fun () ->` within the recursive calls. One way of doing this is by modifying the function `mapq`. There are other solutions.

```
let rec mapapp f xq yf =
  match xq with
  | Nil ->
      yf ()
  | Cons (x, xf) ->
      Cons(f x, fun () -> mapapp f (xf ()) yf)

let rec lperms = function
| [] -> Cons ([], fun () -> Nil)
| xs ->
    let rec perms1 xs ys =
      match xs with
      | [] -> Nil
      | x::xs ->
          mapapp (List.cons x) (lperms (List.rev ys @ xs))
                 (fun () -> perms1 xs (x::ys))
    in
      perms1 xs []
```

An OCaml version of this Tripos would probably have prohibited the use of the `Lazy` module, but this can also be achieved with:

```
type 'a seq = Nil
            | Cons of 'a * 'a seq lazy_t

let rec mapapp f xq yf =
  match xq with
  | Nil ->
      Lazy.force yf
  | Cons (x, xf) ->
      Cons (f x, lazy (mapapp f (Lazy.force xf) yf))

let rec lperms = function
| [] -> Cons ([], lazy Nil)
| xs ->
    let rec perms1 xs ys =
      match xs with
```

```
    | [] -> Nil
    | x::xs ->
        mapapp (List.cons x) (lperms (List.rev ys @ xs))
               (lazy (perms1 xs (x::ys)))
  in
    perms1 xs []
```

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