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

\[
\text{let rec } \text{perms} = \text{function} \\
| [] -> [[]] \\
| xs -> \\
\quad \text{let rec } \text{perms1} \hspace{1pt} xs \hspace{1pt} ys = \\
\quad \quad \text{match } xs \text{ with} \\
\quad \quad | [] -> [] \\
\quad \quad | x::xs -> \\
\quad \quad \quad \text{List.map} \hspace{1pt} (\text{List.cons} \hspace{1pt} x) \hspace{1pt} (\text{perms} \hspace{1pt} (\text{List.rev} \hspace{1pt} ys @ xs)) \hspace{1pt} @ \\
\quad \quad \quad \text{perms1} \hspace{1pt} xs \hspace{1pt} (x::ys) \\
\quad \text{in} \\
\quad \text{perms1} \hspace{1pt} xs \hspace{1pt} []
\]

(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]`?

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.

\[
\text{perms} \hspace{1pt} [1; 2; 3] = \\
[ [1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1] ]
\]

(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`.

\[
\text{let rec } \text{lperms} = \text{function} \\
| [] -> \text{Cons} \hspace{1pt} ([], \text{fun} \hspace{1pt} () -> \text{Nil}) \\
| xs -> \\
\quad \text{let rec } \text{fun} \hspace{1pt} \text{perms1} \hspace{1pt} xs \hspace{1pt} ys = \text{function} \\
\quad \quad | [] -> \text{Nil} \\
\quad \quad | x::xs -> \\
\quad \quad \quad \text{appendq} \hspace{1pt} (\text{mapq} \hspace{1pt} (\text{List.cons} \hspace{1pt} x) \hspace{1pt} (lperms \hspace{1pt} (\text{List.rev} \hspace{1pt} ys @ xs)))) \\
\quad \quad \quad \hspace{1pt} (\text{perms1} \hspace{1pt} xs \hspace{1pt} (x::ys)) \\
\quad \text{in} \\
\quad \text{perms1} \hspace{1pt} xs \hspace{1pt} []
\]

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

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

Laziness can be obtained through writing functions of the form \( \text{fun } () \rightarrow E \), for then the expression \( E \) is not evaluated until the function is called, with argument \( () \).

The function above uses lazy list primitives correctly as regards types, but the only occurrence of \( \text{fun } () \rightarrow \) protects an instance of \( \text{Nil} \). All recursive calls to \text{lperms} take place when the function is called, and therefore all permutations are computed.

---

**c)** Modify the function \text{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 \( \text{fun } () \rightarrow \) within the recursive calls. One way of doing this is by modifying the function \text{mapq}. There are other solutions.

```ocaml
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 \text{Lazy} module, but this can also be achieved with:

```ocaml
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
```

---

2
| [] -> Nil
| x::xs ->
    mapapp (List.cons x) (lperms (List.rev ys @ xs))
    (lazy (perms1 xs (x::ys)))
in
perms1 xs []