NOTE: The code in the exam is available on Codio, and you're welcome to use that rather than trying to type in this code yourself. ***However, using Codio for the exam is completely optional and you can do well in the exam even if you decide not to use it.***

Appendix A: Higher-Order List Processing Functions

Here are the higher-order list processing functions:

```
let rec transform (f: 'a -> 'b) (l: 'a list): 'b list =
  begin match l with
    | [] -> []
    | h :: t -> (f h) :: (transform f t)
  end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
  begin match l with
    | [] -> base
    | h :: t -> combine h (fold combine base t)
  end
```

Appendix B: Generic Binary Tree

Here is the definition of a generic binary tree:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Appendix C: Queue Code

Signature for the purely function queue abstract type.

```
module type Q = sig
  type 'a queue

  val empty : 'a queue
  val is_empty : 'a queue -> bool

  val enq : 'a queue -> 'a -> 'a queue
  val deq : 'a queue -> ('a queue * 'a)     (* fails if queue is empty *)
end
```

One implementation of the `Q` signature.

```
module ListQ : Q = struct
  (* INVARIANT: queue elements are stored in the order
     in which they will be dequeued: the head of the list
     (if any) will be the element returned by deq.
  *)
  type 'a queue = 'a list

  let empty : 'a queue = []
  let is_empty (q : 'a queue) : bool =
    q = []

  let rec enq (q : 'a queue) (x : 'a) : 'a queue =
    begin match q with
    | [] -> [x]
    | y::ys -> y::(enq ys x)
    end

  let deq (q : 'a queue) : 'a queue * 'a =
    begin match q with
    | [] -> failwith "empty queue"
    | x::xs -> (xs, x)
    end
end
```