```
(*
                              CS51 Lab 12
                  Imperative Programming and References
 *)

(*
Objective:

This lab provides practice with reference types and their use in
building mutable data structures and in imperative programming more
generally. It also gives further practice in using modules to abstract
data types.

There are 4 total parts to this lab. Please refer to the following
files to complete all exercises:

   lab12_part1.ml -- Part 1: Implementing modules
   lab12_part2.ml -- Part 2: Files as modules
   lab12_part3.ml -- Part 3: Interfaces as abstraction barriers
-> lab12_part4.ml -- Part 4: Polymorphic abstract types (this file)
 *)

(*======================================================================
Part 4: Adding serialization to imperative queues

In the textbook, we defined a polymorphic imperative queue signature
as follows:

    module type IMP_QUEUE =
      sig
        type 'a queue
        val empty : unit -> 'a queue
        val enq : 'a -> 'a queue -> unit
        val deq : 'a queue -> 'a option
      end

In this part, you'll add functionality to imperative queues for
serializing of the queue. (You added serialization to a pure
(immutable) stack module in Lab 8.) The signature needs to be
augmented first; we've done that for you here: *)

module type IMP_QUEUE =
  sig
    type elt
    type queue

    val empty : unit -> queue
    val enq : elt -> queue -> unit
    val deq : queue -> elt option
    val to_string : queue -> string
  end ;;

(* Notice that we've changed the module slightly so that the element
type (`elt`) is made explicit.

The `to_string` function needs to know how to convert the individual
elements to strings. That information is best communicated by
packaging up the element type and its own `to_string` function in a
module that can serve as the argument to a functor for making
`IMP_QUEUE`s. (You'll recall this technique from Lab 8.)

Given the ability to convert the elements to strings, the `to_string`
function should work by converting each element to a string in order
separated by arrows " -> " and with a final end marker to mark the end
of the queue "||". For instance, the queue generated by enqueing, in
```

order, integer elements 1, 2, and 3 would serialize to the string "1 -> 2 -> 3 -> ||" (as shown in the example below).

We've provided a functor for making imperative queues, which works almost identically to the final implementation from the book (based on mutable lists) except for abstracting out the element type in the functor argument.

```
.....................................................................
Exercise 11: Your job is to complete the implementation by finishing
the `to_string` function. (Read on below for an example of the use of
the functor.)
.....................................................................*)

module MakeImpQueue (Elt : sig
                             type t
                             val to_string : t -> string
                           end)
                  : (IMP_QUEUE with type elt = Elt.t) =
  struct
    type elt = Elt.t
    type mlist = mlist_internal ref
     and mlist_internal =
       | Nil | Cons of elt * mlist
    type queue = {front: mlist; rear: mlist}

    let empty () = {front = ref Nil; rear = ref Nil}

    let enq x q =
      match !(q.rear) with
      | Cons (_hd, tl) -> assert (!tl = Nil);
                          tl := Cons(x, ref Nil);
                          q.rear := !tl
      | Nil -> assert (!(q.front) = Nil);
               q.front := Cons(x, ref Nil);
               q.rear := !(q.front)

    let deq q =
      match !(q.front) with
      | Cons (hd, tl) ->
         q.front := !tl ;
         (match !tl with
          | Nil -> q.rear := Nil
          | Cons(_, _) -> ());
         Some hd
      | Nil -> None

    let to_string q =
      failwith "to_string not implemented"

  end ;;

(* To build an imperative queue, we apply the functor to an
appropriate argument. For instance, we can make an integer queue
module: *)

module IntQueue = MakeImpQueue (struct
                                  type t = int
                                  let to_string = string_of_int
                                end) ;;

(* And now we can test it by enqueueing some elements and converting
the queue to a string to make sure that the right elements are in
there. *)
```

```
let test () =
  let open IntQueue in
  let q = empty () in
  enq 1 q;
  enq 2 q;
  enq 3 q;
  to_string q ;;

(* Running the `test` function should have the following behavior:

    # test () ;;
    - : string = "1 -> 2 -> 3 -> ||"

*)
```