```
(*
                              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 (this file)
   lab12_part4.ml -- Part 4: Polymorphic abstract types
 *)

(*======================================================================
Part 3: Appending mutable lists

Recall the definition of the mutable list type from Section 15.4 of
the textbook: *)

type 'a mlist = 'a mlist_internal ref
 and 'a mlist_internal =
    | Nil
    | Cons of 'a * 'a mlist ;;

(* Mutable lists are just like regular lists, except that each Nil or
cons is a *reference* to a mutable list, so that it can be updated. *)


(*......................................................................
Exercise 5: Construct a function `mlist_empty : unit -> int mlist` that
returns an empty `mlist`.
 *)

let mlist_empty () : int mlist =
  failwith "mlist_empty not implemented" ;;

(* Now call the function `mlist_empty` within the REPL. You'll notice
that the value returned looks like a record structure with a single
`contents` field. That's because internally, OCaml implements `ref`
values as mutable records. *)

(*......................................................................
Exercise 6: Construct a function `mlist_42 : unit -> int mlist` that
returns an `mlist` with the single integer element 42.
 *)

let mlist_42 () : int mlist =
  failwith "mlist_42 not implemented" ;;

(* Now call the function `mlist_42` within the REPL. Again, you'll see
the telltale record structure with a single `contents` field. In the
future, you'll know why that is.

It's cumbersome to manually build up these mutable lists. Let's nuild
a function that allows us to generate mutable versions of regular
(immutable) lists. *)
```

```
(*......................................................................
Exercise 7: Define a polymorphic function `mlist_of_list` that
converts a regular list to a mutable list, with behavior like this:

    # let xs = mlist_of_list ["a"; "b"; "c"] ;;
    val xs : string mlist =
      {contents = Cons ("a",
         {contents = Cons ("b",
            {contents = Cons ("c",
               {contents = Nil})})})}

    # let ys = mlist_of_list [1; 2; 3] ;;
    val ys : int mlist =
      {contents = Cons (1,
         {contents = Cons (2,
            {contents = Cons (3,
               {contents = Nil})})})}
......................................................................*)

let mlist_of_list (lst : 'a list) : 'a mlist =
  failwith "mlist_of_list not implemented" ;;

(*......................................................................
Exercise 8: Define a function `mlength` to compute the length of an
`mlist`. Try to do this without looking at the solution that is given
in the book. (Don't worry about cycles...yet.)

    # mlength (ref Nil) ;;
    - : int = 0
    # mlength (mlist_of_list [1; 2; 3; 4]) ;;
    - : int = 4
......................................................................*)

let mlength (mlst : 'a mlist) : int =
  failwith "length not implemented" ;;

(*......................................................................
Exercise 9: What is the time complexity of the `mlength` function in
terms of the length of its list argument? Provide the tightest
complexity class, recorded using the technique from lab 10.
......................................................................*)

type complexity =
  | Unanswered
  | Constant
  | Logarithmic
  | Linear
  | LogLinear
  | Quadratic
  | Cubic
  | Exponential ;;

let length_complexity : complexity = Unanswered ;;

(*......................................................................
Exercise 10: Now, define a function `mappend` that takes a first
mutable list and a second mutable list and, as a side effect, causes
the first to *become* (as a side effect) the appending of the two
lists. A question to think about before you get started:

    What is an appropriate return type for the `mappend` function?
    (You can glean our intended answer from the examples below, but
    try to think it through yourself first.)
```

```
Examples of use:

    # let m1 = mlist_of_list [1; 2; 3] ;;
    val m1 : int mlist =
      {contents = Cons (1,
         {contents = Cons (2,
            {contents = Cons (3,
               {contents = Nil})})})}

    # let m2 = mlist_of_list [4; 5; 6] ;;
    val m2 : int mlist =
      {contents = Cons (4,
         {contents = Cons (5,
            {contents = Cons (6,
               {contents = Nil})})})}

    # mlength m1 ;;
    - : int = 3

    # mappend m1 m2 ;;
    - : unit = ()

    # mlength m1 ;;
    - : int = 6

    # m1 ;;
    - : int mlist =
    {contents = Cons (1,
       {contents = Cons (2,
          {contents = Cons (3,
             {contents = Cons (4,
                {contents = Cons (5,
                   {contents = Cons (6,
                      {contents = Nil})})})})})})}
......................................................................*)

let mappend _ =
  failwith "mappend not implemented" ;;

(* What happens when you evaluate the following expressions
   sequentially in order?

      # let m = mlist_of_list [1; 2; 3] ;;
      # mappend m m ;;
      # m ;;
      # mlength m ;;

   Do you understand what’s going on? *)
```