```
(*
                              CS51 Lab 8
                               Functors
                                Part 2
 *)

(* Objective:

This lab provides practice in the use of functors.
 *)

(*======================================================================
Functors - Part 2.

In this second part of the lab, you'll rewrite the `Stack` module from
the previous lab so that its element type is polymorphic, while adding
some functionality at the same time.

Occasionally, it's useful to be able to take a data structure and
"serialize" it (also called "marshalling"), that is, to transform its
internal computational representation into a format, like a string,
that can be saved in a file or transmitted digitally. It can later be
"deserialized" to recreate the internal representation. We'll add to
the stack module extremely basic support for serialization.

   (You may notice that this provides an opportunity to break an
   abstraction barrier: serializing an object, manipulating its string
   representation, and then deserializing back to an object could
   allow for some invariant to be violated. Even so, serialization has
   its uses when the circumstances warrant.)

In order to serialize stacks from an abstract stack data type, we will
need to have access to a function that can serialize the *elements* of
the stack.

We can use a functor that takes as argument a module for the stack
elements (including element serialization functionality) and generates
a stack module that bundles together all of the stack functionality,
including serialization.

In order to do this, we'll first define a module interface, called
SERIALIZE, that captures the requirements for the elements of a
stack. It requires that the module expose both a data type and a
function (`serialize`) that converts values of that type into a string
representation. The SERIALIZE module type is an appropriate signature
for the argument of a functor that generates serializable stack
modules. *)

module type SERIALIZE =
  sig
    type t
    val serialize : t -> string
  end ;;

(* Now we'll define a STACK interface. Notice that unlike the
`INT_STACK` interface from the previous lab, we'll specify the
`element` type to be an abstract type as part of the signature, and
add additional functions for serialization, as well as a couple of
higher-order functions over stacks. *)

module type STACK =
  sig
    exception Empty
    type element
    type stack
```

```
    val empty : stack
    val push : element -> stack -> stack
    val top : stack -> element
    val pop : stack -> stack
    val serialize : stack -> string
    val map : (element -> element) -> stack -> stack
    val filter : (element -> bool) -> stack -> stack
    val fold_left : ('a -> element -> 'a) -> 'a -> stack -> 'a
  end ;;

(*.....................................................................
Exercise 1A: Rewrite your stack implementation from the last lab to
use the `element` abstract data type for elements, and also complete
the other functions that the signature requires. We've provided a good
start below.

The serialize function should construct a string in the following
form:

    "N0:N1:N2:...:N"

where N0 was the *first* element pushed onto the stack, N1 was the
*second*, and so on, with N being the *most recent* element pushed to
the stack, that is, the one to be popped next. (This would make
deserialization easier, since elements could be pushed as soon as they
were read.)

(Don't forget that you can implement other functions that are not
specified by the signature if you would like. The module signature
will act as an abstraction barrier to prevent the extra functions from
leaking out of the module.)
.....................................................................*)

module MakeStack (Element: SERIALIZE)
       : (STACK with type element = Element.t) =
  struct
    exception Empty

    type element = Element.t
    type stack = element list

    let empty : stack = []

    let push (el : element) (s : stack) : stack =
      failwith "push not implemented"

    let pop_helper (s : stack) : (element * stack) =
      failwith "pop_helper not implemented"

    let top (s : stack) : element =
      failwith "top not implemented"

    let pop (s : stack) : stack =
      failwith "pop not implemented"

    let map (f : element -> element) (s : stack) : stack =
      failwith "map not implemented"

    let filter (f : element -> bool) (s : stack) : stack =
      failwith "filter not implemented"

    let fold_left (f : 'a -> element -> 'a) (init : 'a) (s : stack) : 'a =
      failwith "fold_left not implemented"

    let serialize (s : stack) : string =
```

```
      failwith "serialize not implemented"
  end ;;

(*.....................................................................
Exercise 1B: Now, make a module `IntStack` by applying the functor
that you just defined to an appropriate module for serializing integers.
.....................................................................*)

module IntStack = struct end ;;

(*.....................................................................
Exercise 1C: Make a module `IntStringStack` that creates a stack whose
elements are `int * string` pairs. Its serialize function should output
elements as strings of the form:

    "(N,'S')"

where N is the int, and S is the string. For instance, a stack with
two elements might be serialized as the string

    "(1,'pushed first'):(2,'pushed second')"     .

For this oversimplified serialization function, you may assume that
the string will be made up of alphanumeric characters only.
.....................................................................*)

module IntStringStack = struct end ;;
```