```
(*
                              CS51 Lab 7
                   Modules and Abstract Data Types
 *)

(* Objective: This lab practices concepts of modules, including files
as modules, signatures, and polymorphic abstract data types.

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

   lab7_part1.ml -- Part 1: Implementing modules
   lab7_part2.ml -- Part 2: Files as modules
-> lab7_part3.ml -- Part 3: Polymorphic abstract types and
                            Interfaces as abstraction barriers (this file)
 *)

(*======================================================================
Part 3: Polymorphic abstract types and
        interfaces as abstraction barriers

It may feel at first that it's redundant to have both signatures and
implementations for modules. However, as we saw at the end of Part 2,
the use of signatures allows defining a set of functionality that
might be shared between two related (but still different) modules.

Another advantage of specifying signatures is that it provides a way
to hide, within a single module, various helper functions or
implementation details that we (as the developer of a module) would
rather not expose. In other words, it provides a rigid *abstraction
barrier* that when deployed properly makes it impossible for a
programmer using your module to violate desired invariants. This
abstraction barrier is a key tool toward satisfying the edicts of
prevention and compartmentalization.

Let's say we want to build a stack data structure. A stack is similar
to a queue (as described in Section 12.2 of the textbook) except that
the *last* element to be added to it is the *first* one to be
removed. That is, it operates as a LIFO (Last-In First-Out) structure.

To get started, we can implement stacks using lists. "Push" and "pop"
are commonly the names of the operations associated with adding an
element to the top or removing the topmost element, respectively. (The
terminology is reminiscent of a stack of plates in a cafeteria: you
begin with an empty pile, "push" one or more plates onto the top of
the stack, and at any point might "pop" a plate off the top of the
pile.)

In imperative programming languages, a "pop" function typically
*modifies* the stack by removing the topmost element from the stack as
a side effect and also returns the value of that element. However, in
the functional paradigm, side effects are eschewed. As a result, we'll
separate "pop" into two functions: "top", which returns the topmost
element from the stack (without removing it), and "pop", which returns
a new stack that has the topmost element removed.

We helped you out a little and defined `top` and `pop` for you,
below. These rely on a helper function called `pop_helper`, which
you'll need to implement. It should accept a stack and return a pair
containing the first element of the argument stack and a stack with
the first element removed. The `pop_helper` function isn't part of the
functionality of a stack implementation, just a help in implementing
that functionality.

You'll want to take advantage of the `EmptyStack` exception provided
```

```
in the module; raise it if an attempt is made to examine or pop the
top of an empty stack.

......................................................................
Exercise 3A: Complete this implementation of a polymorphic stack module.
.....................................................................*)

module ListStack =
  struct
    exception EmptyStack

    (* Stacks will be implemented as integer lists with the most
    recently pushed elements at the front of the list. *)
    type 'a stack = 'a list

    (* empty -- An empty stack *)
    let empty : 'a stack = failwith "empty not implemented"

    (* push elt s -- Returns a stack like stack `stk` but with element
       `elt` added to the top *)
    let push (elt : 'a) (stk : 'a stack) : 'a stack =
      failwith "push not implemented"

    (* pop_helper stk -- Returns a pair of the top element of `stk`
       and a stack containing the remaining elements after removing the
       top element *)
    let pop_helper (stk : 'a stack) : 'a * 'a stack =
      failwith "pop_helper not implemented"

    (* top stk -- Returns the value of the topmost element on stack
       `stk`, raising the `EmptyStack` exception if there is no
       element to be returned. *)
    let top (stk: 'a stack) : 'a =
      fst (pop_helper stk)

    (* pop stk -- Returns a stack with the topmost element from `stk`
       removed, raising the `EmptyStack` exception if there is no
       element to be removed. *)
    let pop (stk : 'a stack) : 'a stack =
      snd (pop_helper stk)
  end ;;

(* Now let's use this implementation and consider some implications.

......................................................................
Exercise 3B: Write a function `small_stack` that takes an argument of
type unit and uses the `ListStack` implementation to create and return
a new stack with the integer values `5` and then `1` pushed in that
order.
.....................................................................*)

let small_stack () : int list =
  failwith "not implemented" ;;

(*.....................................................................
Exercise 3C: Now, use `ListStack` functions to write an expression
that defines `last_el` (not as `0` as below but instead) as the value
of the topmost element from `small_stack`.
.....................................................................*)

let last_el = 0 ;;

(* Based on our requirements above, what should the value `last_el` be?

Look more closely at the type of `small_stack`: It's of type `unit ->
```

```
int list`.  This is expected, since that’s how we defined it, but this
also means that we have the ability to operate on the data structure
without using the provided *abstraction* specified by the module.

In other words, the `ListStack` module was intended to enforce the
invariant that the elements will *always* be pushed and popped in a
LIFO manner. But, we could altogether circumvent that merely by using
arbitrary list operations on the stack.

......................................................................
Exercise 3D: Write a function that, given a stack, returns a stack
with the elements in reverse order, *without using any of the
`ListStack` methods* (but feel free to use `List` library functions).
....................................................................*)

let invert_stack (s : ’a ListStack.stack) : ’a ListStack.stack =
  failwith "not implemented" ;;

(* Now what would be the result of the `top` operation on a stack
inverted with `invert_stack`? Let’s try it.

......................................................................
Exercise 3E: Write an expression using `ListStack` methods to get
the top value from a `small_stack` inverted with `invert_stack` and
name the result `bad_el`.
....................................................................*)

let bad_el = 0 ;;

(* This is bad. We have broken through the *abstraction barrier*
defined by the `ListStack` module, by extracting an element from the
stack that was not the "last in". You may wonder: "if I know that the
module is defined by a list, why not have the power to update it
manually?"

Two reasons:

1. As we’ve just done, it was entirely possible for a user of the
   module to completely modify a value in its internal representation.
   Imagine what would happen for a more complex module that allowed us
   to break an invariant! From Problem Set 3, what would break if a
   person could change a bignum representing zero to directly set the
   negative flag? Or construct a list of coefficients some of which
   are larger than the base?

2. What if the developer of the module wants to change the module
   implementation for a more sophisticated version? Stacks are fairly
   simple, yet there are multiple reasonable ways of implementing the
   stack regimen. And more complex data structures could certainly
   need several iterations of implementation before settling on a
   final version.

Let’s preserve the abstraction barrier by writing an interface for this
module. The signature will manifest an *abstract* data type for stacks,
without revealing the *concrete* implementation type. Because of that
abstraction barrier, users of modules satisfying the signature will have
*no access* to the values of the type except through the abstraction
provided by the functions and values in the signature.

......................................................................
Exercise 3F: Define an interface for a polymorphic stack, called
`STACK`, which should be structured as much as possible like the
implementation of `Stack` above. But be careful in deciding what type
of data your function types in the signature should use; it’s not `’a
list`, even though that’s the type you used in your implementation.
```

```
......................................................................*)

module type STACK =
  sig
    (* ... your specification of the signature goes here ... *)
  end ;;

(* Now, we'll restrict the `ListStack` module to enforce the
`STACK` interface to form a module called `SafeListStack`. *)

module SafeListStack = (ListStack : STACK) ;;

      (* An aside: The idiomatic way to restrict a module to a
      particular signature is to define it that way from the
      start. That is, the definition of the `SafeListStack` module
      (and the `ListStack` module for that matter) should have started

          module SafeListStack : STACK = struct ... end

      In this lab, we used this two-step restriction-after-the-fact
      approach for pedagogical purposes, but you should in general
      apply a signature to a module from the start. *)

(*......................................................................
Exercise 3G: Let's redo Exercise 3B using the `SafeListStack`
abstract data type we've just built. Write a function `safe_stack`
that takes a unit and uses `SafeListStack` methods to return a
stack of type `SafeListStack.stack`, with the integers 5 and then 1
pushed onto it.

What type is `safe_stack`? You should no longer be able to
perform list operations directly on it, which means the stack
preserves its abstraction barrier.
......................................................................*)

let safe_stack () =
  failwith "not implemented" ;;
```