```
(*
                              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: Interfaces as abstraction barriers
-> lab7_part4.ml -- Part 4: Polymorphic abstract types (this file)
 *)

(*======================================================================
Part 4: Polymorphic abstract types

You may have noticed that the stack module in Part 3 focused
exclusively on stacks of `int` values. But this doesn't have to be so:
we can also create modules that implement polymorphic abstract data
types, even ones protected by a signature.

Below is a signature for a stack data structure, but providing a
polymorphic abstract type so that we can generalize stacks to be `int`
stacks, `string` stacks, stacks of all sorts. *)

module type STACK =
  sig
    exception EmptyStack
    type 'a stack
    val empty : 'a stack
    val push : 'a -> 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val pop : 'a stack -> 'a stack
  end ;;

(*.....................................................................
Exercise 4A: Complete the implementation below of a module that
satisfies this stack module signature. First, decide how you'll
represent the stack (perhaps as lists, though the abstraction barrier
means that you're free to choose otherwise), then implement each of
the functions in the signature based on your decision. You may want to
look at Part 3 for inspiration, but this implementation may differ
from your previous implementation based on a design choice we describe
below.

Then, implement each of the values in the stack signature above. 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 ofthe argument stack and a stack with the
first element removed.

Notice that the `pop_helper` function does *not* appear in the
signature, and will therefore *not* be accessible to functions
*outside* of the module. But you'll be able to use it *inside* the
implementation of the module.

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

```
module Stack : STACK =
  struct
    exception EmptyStack

    type 'a stack = 'a      (**** Replace this with the correct
                                implementation type ****)

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

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

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

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

    (* pop s -- Returns a stack with the topmost element from s
       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 ;;

(*.....................................................................
Exercise 4B: Write a function `sample_stack` that takes a unit
argument and uses your Stack module to return a new stack with the
following strings pushed in order: `"Computer"`, `"Science"`, `"51"`.
.....................................................................*)

let sample_stack () =
  failwith "sample_stack not implemented" ;;

(*.....................................................................
Exercise 4C: Write an expression that generates a stack by applying
the `sample_stack` function above and extracts its top element of the
stack, naming the top element `top_el`.
.....................................................................*)

let top_el : string =
  "replace me with an expression using the Stack module" ;;
```