```
(*
                              CS51 Lab 8
                               Functors
                                Part 1
 *)

(* Objective:

This lab provides practice in the use of functors.

This part of the lab has been adapted from the functors chapter of
Real World OCaml <http://dev.realworldocaml.org/functors.html>. *)

(*======================================================================
Functors - Part 1.

For the first part of this lab, you will explore a realistic and
useful application of functors -- a library to support interval
computation.

Intervals come up in many different contexts. As a concrete example,
calendars need to associate events with time intervals (3-4pm or
11:30am-3:30pm). Intervals can be empty (like the interval starting at
4pm and ending at the previous 3pm; it contains no time at all). We
might want to know if a value is contained within an interval. (For
example, 4pm is not contained within 11:30am-3:30pm, but is contained
within 2-5pm.)  We may want to know the intersection of two
intervals. (The intersection of 3-4pm and 11:30am-3:30pm is 3-3:30pm.)

Importantly, intervals can be defined over different data types. For
instance you may want to use intervals over floating point values or,
as in the example above, intervals of times. But regardless of the
type the intervals are over, you'll want some common basic operations:
constructing an interval, checking if an interval is empty, checking
if a point is contained within an interval, and intersecting two
intervals. (Of course, in an actual interval library, you'd want
several other operations as well, but this will be enough to get us
started.)

Let's examine how we can build a *generic* interval library using
functors.

An interval is defined by its endpoints. In order to operate on
intervals, we need to be able to manipulate the endpoints as well, in
particular by comparing two endpoints for how they are ordered. Thus,
to define intervals over a certain kind of endpoints, we'll need the
type of the endpoints and a comparison function.

The `ORDERED_TYPE` signature from the textbook (Section 12.3) will
serve us nicely here. It uses the OCaml convention for comparison
functions used by the `Stdlib.compare` function:

    compare x y < 0     (* x < y *)
    compare x y = 0     (* x = y *)
    compare x y > 0     (* x > y *)
 *)

module type ORDERED_TYPE =
  sig
    type t
    val compare : t -> t -> int
  end ;;

(*......................................................................
Exercise 1A: Complete the following functor for making interval modules.
```

```
We represent an interval with a variant type, which is either `Empty`
or `Interval (x, y)`, where `x` and `y` are the bounds of the
interval, and are themselves contained within the interval. The
representation has an invariant that `x` is always less than or equal
to `y` (where the notion of "less than or equal" is given by the
`compare` function provided by the `ORDERED_TYPE` that the intervals
are built from).

The functor definition starts out as

    module MakeInterval (Endpoint : ORDERED_TYPE) =
      struct
        ...
      end

Here, the argument to the `MakeInterval` functor is a module, given
the name `Endpoint` and constrained to satisfy the `ORDERED_TYPE`
module signature. Thus, we know that `Endpoint` will provide both a
type (`Endpoint.t`) and a comparison function over that type
(`Endpoint.compare`). We can and will use those in defining the module
being defined by the functor.

Now, complete the functor definition below.
......................................................................*)

module MakeInterval (Endpoint : ORDERED_TYPE) =
  struct
    type interval =
      | Interval of Endpoint.t * Endpoint.t
      | Empty

    (* create low high -- Returns a new interval covering `low` to
       `high` inclusive. If `low` is greater than `high`, then the
       interval is empty. *)
    let create (low : Endpoint.t) (high : Endpoint.t) : interval =
      failwith "create not implemented"

    (* is_empty intvl -- Returns true if and only if `intvl` is
       empty *)
    let is_empty (intvl : interval) : bool =
      failwith "is_empty not implemented"

    (* contains intvl x -- Returns true if and only if the value `x`
       is contained within `intvl` *)
    let contains (intvl : interval) (x : Endpoint.t) : bool =
      failwith "contains not implemented"

    (* intersect intvl1 intvl2 -- Returns the intersection of `intvl1`
       and `intvl2` *)
    let intersect (intvl1 : interval) (intvl2 : interval) : interval =
      failwith "intersect not implemented"
    end ;;

(*......................................................................
Exercise 1B: Using the completed functor above, instantiate an
*integer* interval module.
......................................................................*)

module IntInterval =
  struct end ;;  (* <-- replace this line with an appropriate module
                        definition using the `MakeInterval` functor   *)

(*......................................................................
Exercise 1C: Using your newly created integer interval module, create
```

```
two non-empty intervals named `intvl1` and `intvl2` that have some
overlap, and calculate their intersection as `intvl1_intersect_intvl2`.
......................................................................*)

let intvl1 = failwith "not implemented" ;;
let intvl2 = failwith "not implemented" ;;
let intvl1_intersect_intvl2 = failwith "not implemented" ;;

(* There's currently a problem with the `MakeInterval` functor. It's
not abstract enough. Notably we are working with an invariant that a
valid non-empty interval has an upper bound that is greater than or
equal to its lower bound. However, this is only enforced by the
`create` function, and as it turns out, we can actually bypass the
`create` function due to our lack of an abstraction barrier.

This expression returns `true`, as expected.

    IntInterval.is_empty (IntInterval.create 4 3) ;;

This, however, returns `false`. Yikes.

    IntInterval.is_empty (IntInterval.Interval (4, 3)) ;;

**Make sure you understand why this is a problem. If you don't see the
issue, call over a staff member to discuss.**

To make our functor more abstract, we need to restrict the output of
`MakeInterval` to an interface that prevents users from directly
creating interval implementations themselves without using the
`create` function.

.......................................................................
Exercise 2A: Complete the following interface for an interval
module. Note in particular that we should add a new type `endpoint` to
give us a way of abstractly referring to the type for the endpoints of
an interval.
......................................................................*)

module type INTERVAL =
  sig
    type interval
    type endpoint
    (* ... complete the interface here ... *)
  end ;;

(*......................................................................
Exercise 2B: Augment the `MakeSafeInterval` functor using the code you
wrote for `MakeInterval` as a starting point, such that it returns a
module restricted to the `INTERVAL` signature. (Much of the
implementation can be copied from `MakeInterval` above.) **Don't
forget to specify the module type.**
......................................................................*)

module MakeSafeInterval (Endpoint : ORDERED_TYPE) =
  struct
    (* ... complete the module implementation here ... *)
  end ;;

(* We have successfully made our returned module abstract, but believe
it or not, it is now too abstract. In fact, we have not exposed the
type of endpoints to the user, meaning we cannot even create intervals
now. The abstraction barrier is too strong. To demonstrate the problem
...

.......................................................................
```

```
Exercise 2C: Create an `IntSafeInterval` module using the new
`MakeSafeInterval` functor.
......................................................................*)

module IntSafeInterval =
  struct end ;;  (* <-- replace this line with an appropriate module
                        definition using the `MakeSafeInterval` functor *)

(* Now, try evaluating the following expression in the REPL:

    IntSafeInterval.create 2 3 ;;

A type error will appear:

    Error: This expression has type int but an expression was expected of type
           IntInterval.endpoint

To make the interface slightly less abstract, we can make use of a
sharing constraint, which informs the compiler that a given type
*within* the implementation is equal to some other type from *outside*
the implementation. In this case, we want to inform the compiler the
type of our endpoint is an `int`, and more generally that the type of
our endpoint is `Endpoint.t`, where `Endpoint` was the `ORDERED_TYPE`
module that is the argument of the functor. We can do so with the
following syntax:

    <Module_type> with type <type> = <type'>

For instance, we can create int interval and float interval interfaces
that reveal the type of endpoints as follows: *)

module type INT_INTERVAL =
  INTERVAL with type endpoint = int ;;

module type FLOAT_INTERVAL =
  INTERVAL with type endpoint = float ;;

(* Modules that satisfy these interfaces will allow users to actually
construct intervals of the desired types.

While sharing constraints solve the abstraction issue discussed
earlier, they now present a new problem. They will result in code
duplication in implementation. The solution to this is to use sharing
constraints in the MakeInterval functor, exposing that the type of an
endpoint is equal to Endpoint.t. The functor can then be used to
create interval modules of various types without duplicating
code. That's what we'll do next. *)

(*......................................................................
Exercise 3A: Define a new functor `MakeBestInterval`. It should take an
ORDERED_TYPE module for the endpoints of the intervals, and return a
module satisfying INTERVAL *with appropriate sharing constraints
to allow the creation of generic interval modules*.

If you do this correctly, your code for `MakeBestInterval` should be
almost identical to your code for `MakeSafeInterval` with the
exception of any needed sharing constraints.
......................................................................*)

(* ... place your implementation of the MakeBestInterval functor here ... *)

(* We now have a fully functioning functor that can create interval
modules of whatever type we want, with the appropriate abstraction
level.
```

```
......................................................................
Exercise 3B: Use the `MakeBestInterval` functor to create a new int
interval module called `IntBestInterval`, and test that it works as
expected.

You may for instance want to try the problematic lines from Exercise 1C.

The following expression should still return `true`, as expected:

    IntBestInterval.is_empty (IntBestInterval.create 4 3) ;;

and this expression should no longer return `false`. What does it
return instead?

    IntBestInterval.is_empty (IntBestInterval.Interval (4, 3)) ;;
.....................................................................*)

module IntBestInterval = struct end ;;
```