```
(*
                              CS51 Lab 4
                 Error Handling, Options, and Exceptions
 *)

(*=====================================================================
Part 1: Option types and exceptions

In Lab 2, you implemented a function `max_list` that returns the maximum
element in a non-empty integer list. Here's a possible implementation
for `max_list`:

    let rec max_list (lst : int list) : int =
      match lst with
      | [elt] -> elt
      | head :: tail -> max head (max_list tail) ;;

(This implementation makes use of the polymorphic `max` function from
the `Stdlib` module.)

As written, this function generates a warning that the match is not
exhaustive. Why? What's an example of the missing case? Try entering
the function in `ocaml` or `utop` and see what information you can
glean from the warning message. Go ahead; we'll wait.

The problem is that there is no reasonable value for the maximum
element in an empty list. This is an ideal application for option
types.

.....................................................................
Exercise 5:

Reimplement `max_list`, but this time, it should return an `int option`
instead of an `int`. Call it `max_list_opt`. The `None` return value
should be used when called on an empty list.

(Using the suffix `_opt` is a standard convention in OCaml for
functions that return an option type for this purpose. See, for
instance, the functions `nth` and `nth_opt` in the `List` module.)
.....................................................................*)

let max_list_opt (lst : int list) : int option =
  failwith "max_list_opt not implemented" ;;

(*.....................................................................
Exercise 6: Alternatively, we could have `max_list` raise an exception
upon discovering the error condition. Reimplement `max_list` so that it
does so. What exception should it raise? (See Section 10.3 in the
textbook for some advice.)
.....................................................................*)

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

(*.....................................................................
Exercise 7: Write a function `min_option` to return the smaller of its
two `int option` arguments, or `None` if both are `None`. If exactly one
argument is `None`, return the other. The built-in function `min` from
the Stdlib module may be useful. You'll want to make sure that all
possible cases are handled; no nonexhaustive match warnings!
.....................................................................*)

let min_option (x : int option) (y : int option) : int option =
  failwith "min_option not implemented" ;;
```

```
(*.....................................................................
Exercise 8: Write a function `plus_option` to return the sum of its two
`int option` arguments, or `None` if both are `None`. If exactly one
argument is `None`, return the other.
.....................................................................*)

let plus_option (x : int option) (y : int option) : int option =
  failwith "plus_option not implemented" ;;

(*=====================================================================
Part 2: Polymorphism practice

.....................................................................
Exercise 9: Do you see a pattern in your implementations of
`min_option` and `plus_option`? How can we factor out similar code?

Write a polymorphic higher-order function `lift_option` to "lift"
binary operations to operate on option type values, taking three
arguments in order: the binary operation (a curried function) and its
first and second arguments as option types. If both arguments are
`None`, return `None`.  If one argument is `None`, the function should
return the other argument. If neither argument is `None`, the binary
operation should be applied to the argument values and the result
appropriately returned.

What is the type signature for `lift_option`? (If you're having
trouble figuring that out, call over a staff member, or check our
intended type at https://url.cs51.io/lab4-1.)

Now implement `lift_option`.
.....................................................................*)

let lift_option =
  fun _ -> failwith "lift_option not implemented" ;;

(*.....................................................................
Exercise 10: Now rewrite `min_option` and `plus_option` using the
higher-order function `lift_option`. Call them `min_option_2` and
`plus_option_2`.
.....................................................................*)

let min_option_2 =
  fun _ -> failwith "min_option_2 not implemented" ;;

let plus_option_2 =
  fun _ -> failwith "plus_option_2 not implemented" ;;

(*.....................................................................
Exercise 11: Now that we have `lift_option`, we can use it in other
ways. Because `lift_option` is polymorphic, it can work on things other
than `int option`s. Define a function `and_option` to return the boolean
AND of two `bool option`s, or `None` if both are `None`. If exactly one
is `None`, return the other.
.....................................................................*)

let and_option =
  fun _ -> failwith "and_option not implemented" ;;

(*.....................................................................
Exercise 12: In Lab 3, you implemented a polymorphic function `zip` that
takes two lists and "zips" them together into a list of pairs. Here's
a possible implementation of `zip`:

    let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
      match x, y with
```

```
      | [], [] -> []
      | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
```

A problem with the implementation of `zip` is that, once again, its match is not exhaustive and it raises an exception when given lists of unequal length. How can you use option types to generate an alternate solution without this property?

Do so below in a new definition of `zip`, called `zip_opt` to make clear that its signature has changed, which returns an appropriate option type in case it is called with lists of unequal length. Here are some examples:

```
    # zip_opt [1; 2] [true; false] ;;
    - : (int * bool) list option = Some [(1, true); (2, false)]
    # zip_opt [1; 2] [true; false; true] ;;
    - : (int * bool) list option = None
.....................................................................*)

let zip_opt =
  fun _ -> failwith "zip not implemented" ;;

(*======================================================================
Part 3: Factoring out None-handling

Recall the definition of `dotprod` from Lab 2. Here it is, adjusted to
an option type:

    let dotprod_opt (a : int list) (b : int list) : int option =
      let pairsopt = zip_opt a b in
      match pairsopt with
      | None -> None
      | Some pairs -> Some (sum (prods pairs)) ;;

It uses `zip_opt` from Exercise 12, `prods` from Lab 3, and a function
`sum` to sum up all the integers in a list. The `sum` function is
simply *)

let sum : int list -> int =
  List.fold_left (+) 0 ;;

(* and a version of `prods` is *)

let prods =
  List.map (fun (x, y) -> x * y) ;;

(* Notice how in `dotprod_opt` and other option-manipulating functions
we frequently and annoyingly have to test if a value of option type is
`None`; this requires a separate match, and passing on the `None`
value in the "bad" branch and introducing a `Some` in the "good"
branch. This is something we're likely to be doing a lot of. Let's
factor that out to simplify the implementation.

.....................................................................
Exercise 13: Define a function called `maybe` that takes a function of
type `'a -> 'b` and an argument of type `'a option`, and "maybe"
(depending on whether its argument is a `None` or a `Some`) applies the
function to the argument. The `maybe` function either passes on the
`None` if its first argument is `None`, or if its first argument is
`Some v`, it applies its second argument to that `v` and returns the
result, appropriately adjusted for the result type.

What should the type of the `maybe` function be?

Now implement the `maybe` function.
```

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

let maybe (f : 'a -> 'b) (x : 'a option) : 'b option =
  failwith "maybe not implemented" ;;

(*.....................................................................
Exercise 14: Now reimplement `dotprod_opt` to use the `maybe`
function. (The previous implementation makes use of functions `sum`
and `prods`, which we've provided for you above.)  Your new solution
for `dotprod` should be much simpler than the version we provided
above at the top of Part 3.
.....................................................................*)

let dotprod_opt (a : int list) (b : int list) : int option =
  failwith "dotprod not implemented" ;;

(*.....................................................................
Exercise 15: Reimplement `zip_opt` using the `maybe` function, as
`zip_opt_2` below.
.....................................................................*)

let rec zip_opt_2 (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
  failwith "zip_opt_2 not implemented" ;;

(*.....................................................................
Exercise 16: [Optional] For the energetic, reimplement `max_list_opt`
as `max_list_opt_2` along the same lines. There's likely to be a
subtle issue here, since the `maybe` function always passes along the
`None`.
.....................................................................*)

let rec max_list_opt_2 (lst : int list) : int option =
  failwith "max_list not implemented" ;;
```