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


(*======================================================================
Readings:

    This lab builds on material from Chapter 10 of the textbook
    <http://book.cs51.io>, which should be read before the lab session.

======================================================================*)

(*======================================================================
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.
```

```ocaml

.....................................................................
Exercise 1:

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 rec max_list_opt (lst : int list) : int option =
  match lst with
  | [] -> None
  | head :: tail ->
     match (max_list_opt tail) with
     | None -> Some head
     | Some max_tail -> Some (max head max_tail) ;;

(*...................................................................
Exercise 2: 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 rec max_list (lst : int list) : int =
  match lst with
  | [] -> raise (Invalid_argument "max_list: empty list")
  | [elt] -> elt
  | head :: tail -> max head (max_list tail) ;;

(*...................................................................
Exercise 3: 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 =
  match x, y with
  | None,       None        -> None
  | None,       Some _right -> y
```

```
  | Some _left, None        -> x
  | Some  left, Some  right -> Some (min left right) ;;

(*......................................................................
Exercise 4: 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 =
  match x, y with
  | None,       None        -> None
  | None,       Some _right -> y
  | Some _left, None        -> x
  | Some  left, Some  right -> Some (left + right) ;;

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

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

........................................................................
Exercise 5: 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`.
......................................................................*)

(* SOLUTION: The type signature for `lift_option` is naturally
   polymorphic:

     ('a -> 'a -> 'a) -> 'a option -> 'a option -> 'a option     .

   Notice the nice symmetry, which is perhaps made clearer when
   parenthesized as
```

```
     ('a -> 'a -> 'a) -> ('a option -> 'a option -> 'a option)     .

   To think about: Both the first and second argument of `f` must be
   of the same type as the result type of `f` (and hence of each
   other). Do you see why?
 *)
let lift_option (f : 'a -> 'a -> 'a) (x : 'a option) (y : 'a option)
              : 'a option =
  match x, y with
  | None,       None        -> None
  | None,       Some _right -> y
  | Some _left, None        -> x
  | Some  left, Some right  -> Some (f left right) ;;

(*......................................................................
Exercise 6: Now rewrite `min_option` and `plus_option` using the
higher-order function `lift_option`. Call them `min_option_2` and
`plus_option_2`.

    Note: You might encounter inexplicable "weak type variable"
    warnings. If you do, you should make sure to type the arguments of
    the function. For more information read the detailed explanation
    in the lab's solution comments or Section 9.6 of the textbook.
......................................................................*)

let min_option_2 : int option -> int option -> int option =
  lift_option min ;;

(* You may have not added in the specific type information in your
   definition of `min_option_2`, and received an inscrutable warning
   involving "weak type variables", and type problems when submitting
   your code. Here's an example of that behavior:

    # let min_option_2 =
        lift_option min ;;
    val min_option_2 : '_weak1 option -> '_weak1 option -> '_weak1 option = <fun>
    # min_option_2 (Some 3) (Some 4) ;;
    - : int option = Some 3
    # min_option_2 (Some 4.2) (Some 4.1) ;;
    Error: This expression [namely, the 4.2] has type float but an expression
           was expected of type int

   The type variables like `'_weak1` (with the underscore) are "weak
   type variables", not true type variables. Weak type variables are
   discussed briefly in Section 9.7 of the textbook. They arise
```

```
   because in certain situations OCaml's type inference can't figure
   out how to express the most general types and must resort to this
   weak type variable approach.

   When a function with these weak type variables is applied to
   arguments with a specific type, the polymorphism of the function
   disappears. Notice that the first time we apply min_option_2 above
   to int options, things work fine. But the second time, applied to
   float options, there's a type clash because the first use of
   min_option_2 fixed the weak type variables to be ints. Since our
   unit tests try using min_option_2 in certain ways inconsistent with
   weak type variables, you'll get an error message saying that "The
   type of this expression, '_weak1 option -> '_weak1 option ->
   '_weak1 option, contains type variables that cannot be
   generalized."

   To correct the problem, you can add in specific typing information
   (as we've done in the solution above) or make explicit the full
   application of `lift_option`:

    let min_option_2 x y =
      lift_option min x y ;;

   rather than the partial application we used. Either of these
   approaches gives OCaml sufficient hints to infer types more
   accurately.

   For the curious, if you want to see what's going on in detail, you
   can check out the discussion in the section "A function obtained
   through partial application is not polymorphic enough" at
   <https://v2.ocaml.org/learn/faq.html#Typing>.  *)

let plus_option_2 : int option -> int option -> int option =
  lift_option (+) ;;

(*......................................................................
Exercise 7: 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 : bool option -> bool option -> bool option =
  lift_option (&&) ;;

```

```
(*......................................................................
Exercise 8: 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 this 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 rec zip_opt (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
  match x, y with
  | [], [] -> Some []
  | xhd :: xtl, yhd :: ytl ->
     (match zip_opt xtl ytl with
      | None -> None
      | Some ztl -> Some ((xhd, yhd) :: ztl))
  | _, _ -> None ;;

(*======================================================================
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)) ;;
```

```ocaml

It uses `zip_opt` from Exercise 8, `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 9: Define a function called `maybe` that takes a first
argument, function of type `'arg -> 'result`, and a second argument,
of type `'arg option`, and "maybe" applies the first (the function) to
the second (the argument), depending on whether its argument is a
`None` or a `Some`. The `maybe` function either passes on the `None`
if its second argument is `None`, or if its second argument is `Some
v`, it applies its first 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 : 'arg -> 'result) (x : 'arg option) : 'result option =
  match x with
  | None -> None
  | Some v -> Some (f v) ;;

(*.....................................................................
Exercise 10: 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 =
  maybe (fun pairs -> sum (prods pairs))
        (zip_opt a b) ;;

(*..................................................................
Exercise 11: Reimplement `zip_opt` using the `maybe` function, as
`zip_opt_2` below.
..................................................................*)

(* We remove the embedded match using a maybe: *)

let rec zip_opt_2 (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
  match x, y with
  | [], [] -> Some []
  | xhd :: xtl, yhd :: ytl ->
     maybe (fun ztl -> ((xhd, yhd) :: ztl))
           (zip_opt_2 xtl ytl)
  | _, _ -> None ;;

(*..................................................................
Exercise 12: [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 =
  match lst with
  | [] -> None
  | [single] -> Some single
  | head :: tail ->
     maybe (fun max_tail -> max head max_tail)
           (max_list_opt_2 tail) ;;

(* The subtle issue is this. Recall the previous definition of
   `max_list_opt` above:

        let rec max_list_opt (lst : int list) : int option =
          match lst with
          | [] -> None
          | head :: tail ->
             match (max_list_opt tail) with
             | None -> Some head
             | Some max_tail -> Some (max head max_tail) ;;
```

```
In this version, no special match case is needed for the case of a
singleton list. Instead, we can recur all the way to the empty list
case, and handle the singleton case in the first case in the
embedded match, where the `None` from the recursive call becomes
`Some head`. However, when using the `maybe` in the corresponding
case in `max_list_opt_2`, we can't allow the recursion to proceed
all the way to the empty list, where `None` would be returned,
because `maybe` always preserves `None`s; we can't "promote" the
`None` to a `Some`. Consequently, we need to handle the singleton
case explicitly. *)
```