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

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

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

```
(*...............................................................
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 max_list (lst : int list) : int =
  failwith "max_list not implemented" ;;
```

```
(*...............................................................
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 =
  failwith "min_option not implemented" ;;
```

```
(*...............................................................
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 =
  failwith "plus_option not implemented" ;;
```

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

Do you see a pattern in your implementations of

```
91   `min_option` and `plus_option`? How can we factor out similar code?
92
93   ...................................................................
94   Exercise 5: Write a polymorphic higher-order function `lift_option` to
95   "lift" binary operations to operate on option type values, taking
96   three arguments in order: the binary operation (a curried function)
97   and its first and second arguments as option types. If both arguments
98   are `None`, return `None`.  If one argument is `None`, the function
99   should return the other argument. If neither argument is `None`, the
100  binary operation should be applied to the argument values and the
101  result appropriately returned.
102
103  What is the type signature for `lift_option`? (If you're having
104  trouble figuring that out, call over a staff member, or check our
105  intended type at <https://url.cs51.io/lab4-1>.)
106
107  Now implement `lift_option`.
108  ................................................................*)
109
110  let lift_option =
111    fun _ -> failwith "lift_option not implemented" ;;
112
113  (*................................................................
114  Exercise 6: Now rewrite `min_option` and `plus_option` using the
115  higher-order function `lift_option`. Call them `min_option_2` and
116  `plus_option_2`.
117  ................................................................*)
118
119  let min_option_2 =
120    fun _ -> failwith "min_option_2 not implemented" ;;
121
122  let plus_option_2 =
123    fun _ -> failwith "plus_option_2 not implemented" ;;
124
125  (*................................................................
126  Exercise 7: Now that we have `lift_option`, we can use it in other
127  ways. Because `lift_option` is polymorphic, it can work on things other
128  than `int option`s. Define a function `and_option` to return the boolean
129  AND of two `bool option`s, or `None` if both are `None`. If exactly one
130  is `None`, return the other.
131  ................................................................*)
132
133  let and_option =
134    fun _ -> failwith "and_option not implemented" ;;
135
136  (*................................................................
```

```
137   Exercise 8: In Lab 3, you implemented a polymorphic function `zip` that
138   takes two lists and "zips" them together into a list of pairs. Here's
139   a possible implementation of `zip`:
140
141       let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
142         match x, y with
143         | [], [] -> []
144         | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
145
146   A problem with this implementation of `zip` is that, once again, its
147   match is not exhaustive and it raises an exception when given lists of
148   unequal length. How can you use option types to generate an alternate
149   solution without this property?
150
151   Do so below in a new definition of `zip` -- called `zip_opt` to make
152   clear that its signature has changed -- which returns an appropriate
153   option type in case it is called with lists of unequal length. Here
154   are some examples:
155
156       # zip_opt [1; 2] [true; false] ;;
157       - : (int * bool) list option = Some [(1, true); (2, false)]
158       # zip_opt [1; 2] [true; false; true] ;;
159       - : (int * bool) list option = None
160   ..............................................................*)
161
162   let zip_opt =
163     fun _ -> failwith "zip not implemented" ;;
164
165   (*==================================================================
166   Part 3: Factoring out None-handling
167
168   Recall the definition of `dotprod` from Lab 2. Here it is, adjusted to
169   an option type:
170
171       let dotprod_opt (a : int list) (b : int list) : int option =
172         let pairsopt = zip_opt a b in
173         match pairsopt with
174         | None -> None
175         | Some pairs -> Some (sum (prods pairs)) ;;
176
177   It uses `zip_opt` from Exercise 8, `prods` from Lab 3, and a function
178   `sum` to sum up all the integers in a list. The `sum` function is
179   simply *)
180
181   let sum : int list -> int =
182     List.fold_left (+) 0 ;;
```

```
183
184    (* and a version of `prods` is *)
185
186    let prods =
187      List.map (fun (x, y) -> x * y) ;;
188
189    (* Notice how in `dotprod_opt` and other option-manipulating functions
190    we frequently and annoyingly have to test if a value of option type is
191    `None`; this requires a separate match, and passing on the `None`
192    value in the "bad" branch and introducing a `Some` in the "good"
193    branch. This is something we're likely to be doing a lot of. Let's
194    factor that out to simplify the implementation.
195
196    .....................................................................
197    Exercise 9: Define a function called `maybe` that takes a first
198    argument, function of type `'arg -> 'result`, and a second argument,
199    of type `'arg option`, and "maybe" applies the first (the function) to
200    the second (the argument), depending on whether its argument is a
201    `None` or a `Some`. The `maybe` function either passes on the `None`
202    if its second argument is `None`, or if its second argument is `Some
203    v`, it applies its first argument to that `v` and returns the result,
204    appropriately adjusted for the result type.
205
206    What should the type of the `maybe` function be?
207
208    Now implement the `maybe` function.
209    .....................................................................*)
210
211    let maybe (f : 'arg -> 'result) (x : 'arg option) : 'result option =
212      failwith "maybe not implemented" ;;
213
214    (*...................................................................
215    Exercise 10: Now reimplement `dotprod_opt` to use the `maybe`
216    function. (The previous implementation makes use of functions `sum`
217    and `prods`, which we've provided for you above.)  Your new solution
218    for `dotprod` should be much simpler than the version we provided
219    above at the top of Part 3.
220    .....................................................................*)
221
222    let dotprod_opt (a : int list) (b : int list) : int option =
223      failwith "dotprod not implemented" ;;
224
225    (*...................................................................
226    Exercise 11: Reimplement `zip_opt` using the `maybe` function, as
227    `zip_opt_2` below.
228    .....................................................................*)
```

```
229
230  let rec zip_opt_2 (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
231    failwith "zip_opt_2 not implemented" ;;
232
233  (*.............................................................
234  Exercise 12: [Optional] For the energetic, reimplement `max_list_opt`
235  as `max_list_opt_2` along the same lines. There's likely to be a
236  subtle issue here, since the `maybe` function always passes along the
237  `None`.
238  ..................................................................*)
239
240  let rec max_list_opt_2 (lst : int list) : int option =
241    failwith "max_list not implemented" ;;
```