

```

1  (*
2           CS51 Lab 4
3           Error Handling, Options, and Exceptions
4   *)
5  (*
6           SOLUTION
7   *)
8
9
10 (*=====
11 Readings:
12
13 This lab builds on material from Chapter 10 of the textbook
14 <http://book.cs51.io>, which should be read before the lab session.
15 =====*)
16 =====
17
18 (*=====
19 Part 1: Option types and exceptions
20
21 In Lab 2, you implemented a function `max_list` that returns the maximum
22 element in a non-empty integer list. Here's a possible implementation
23 for `max_list`:
24
25 let rec max_list (lst : int list) : int =
26   match lst with
27   | [elt] -> elt
28   | head :: tail -> max head (max_list tail) ;;
29
30 (This implementation makes use of the polymorphic `max` function from
31 the `Stdlib` module.)
32
33 As written, this function generates a warning that the match is not
34 exhaustive. Why? What's an example of the missing case? Try entering
35 the function in `ocaml` or `utop` and see what information you can
36 glean from the warning message. Go ahead; we'll wait.
37
38 .
39 .
40 .
41
42 The problem is that *there is no reasonable value for the maximum
43 element in an empty list*. This is an ideal application for option
44 types.

```

```

45
46 .....  

47 Exercise 1:  

48  

49 Reimplement 'max_list', but this time, it should return an 'int option'  

50 instead of an 'int'. Call it 'max_list_opt'. The 'None' return value  

51 should be used when called on an empty list.  

52  

53 (Using the suffix '_opt' is a standard convention in OCaml for  

54 functions that return an option type for this purpose. See, for  

55 instance, the functions 'nth' and 'nth_opt' in the 'List' module.)  

56 .....*)  

57  

58 let rec max_list_opt (lst : int list) : int option =  

59   match lst with  

60   | [] -> None  

61   | head :: tail ->  

62     match (max_list_opt tail) with  

63     | None -> Some head  

64     | Some max_tail -> Some (max head max_tail) ;;  

65  

66 (*.....  

67 Exercise 2: Alternatively, we could have 'max_list' raise an exception  

68 upon discovering the error condition. Reimplement 'max_list' so that it  

69 does so. What exception should it raise? (See Section 10.3 in the  

70 textbook for some advice.)  

71 .....*)  

72  

73 let rec max_list (lst : int list) : int =  

74   match lst with  

75   | [] -> raise (Invalid_argument "max_list: empty list")  

76   | [elt] -> elt  

77   | head :: tail -> max head (max_list tail) ;;  

78  

79 (*.....  

80 Exercise 3: Write a function 'min_option' to return the smaller of its  

81 two 'int option' arguments, or 'None' if both are 'None'. If exactly one  

82 argument is 'None', return the other. The built-in function 'min' from  

83 the Stdlib module may be useful. You'll want to make sure that all  

84 possible cases are handled; no nonexhaustive match warnings!  

85 .....*)  

86  

87 let min_option (x : int option) (y : int option) : int option =  

88   match x, y with  

89   | None,      None      -> None  

90   | None,      Some _right -> y

```



```

183 because in certain situations OCaml's type inference can't figure
184 out how to express the most general types and must resort to this
185 weak type variable approach.
186
187 When a function with these weak type variables is applied to
188 arguments with a specific type, the polymorphism of the function
189 disappears. Notice that the first time we apply min_option_2 above
190 to int options, things work fine. But the second time, applied to
191 float options, there's a type clash because the first use of
192 min_option_2 fixed the weak type variables to be ints. Since our
193 unit tests try using min_option_2 in certain ways inconsistent with
194 weak type variables, you'll get an error message saying that "The
195 type of this expression, '_weak1 option -> '_weak1 option ->
196 '_weak1 option, contains type variables that cannot be
197 generalized."
198
199 To correct the problem, you can add in specific typing information
200 (as we've done in the solution above) or make explicit the full
201 application of `lift_option`:
202
203 let min_option_2 x y =
204     lift_option min x y;;
205
206 rather than the partial application we used. Either of these
207 approaches gives OCaml sufficient hints to infer types more
208 accurately.
209
210 For the curious, if you want to see what's going on in detail, you
211 can check out the discussion in the section "A function obtained
212 through partial application is not polymorphic enough" at
213 <https://v2.ocaml.org/learn/faq.html#Typing>. *)
214
215 let plus_option_2 : int option -> int option -> int option =
216     lift_option (+) ;;
217
218 (*.....*
219 Exercise 7: Now that we have `lift_option`, we can use it in other
220 ways. Because `lift_option` is polymorphic, it can work on things other
221 than `int option`s. Define a function `and_option` to return the boolean
222 AND of two `bool option`s, or `None` if both are `None`. If exactly one
223 is `None`, return the other.
224 .....*)
225
226 let and_option : bool option -> bool option -> bool option =
227     lift_option (&&) ;;
228

```

```

229 (*.....)
230 Exercise 8: In Lab 3, you implemented a polymorphic function `zip` that
231 takes two lists and "zips" them together into a list of pairs. Here's
232 a possible implementation of `zip`:
233
234 let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
235   match x, y with
236   | [], [] -> []
237   | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
238
239 A problem with this implementation of `zip` is that, once again, its
240 match is not exhaustive and it raises an exception when given lists of
241 unequal length. How can you use option types to generate an alternate
242 solution without this property?
243
244 Do so below in a new definition of `zip` -- called `zip_opt` to make
245 clear that its signature has changed -- which returns an appropriate
246 option type in case it is called with lists of unequal length. Here
247 are some examples:
248
249 # zip_opt [1; 2] [true; false] ;;
250 - : (int * bool) list option = Some [(1, true); (2, false)]
251 # zip_opt [1; 2] [true; false; true] ;;
252 - : (int * bool) list option = None
253 .....*)
254
255 let rec zip_opt (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
256   match x, y with
257   | [], [] -> Some []
258   | xhd :: xtl, yhd :: ytl ->
259     (match zip_opt xtl ytl with
260      | None -> None
261      | Some ztl -> Some ((xhd, yhd) :: ztl))
262   | _, _ -> None ;;
263
264 (*=====
265 Part 3: Factoring out None-handling
266
267 Recall the definition of `dotprod` from Lab 2. Here it is, adjusted to
268 an option type:
269
270 let dotprod_opt (a : int list) (b : int list) : int option =
271   let pairsopt = zip_opt a b in
272   match pairsopt with
273   | None -> None
274   | Some pairs -> Some (sum (prods pairs)) ;;

```

```

275
276 It uses `zip_opt` from Exercise 8, `prods` from Lab 3, and a function
277 `sum` to sum up all the integers in a list. The `sum` function is
278 simply *)
279
280 let sum : int list -> int =
281   List.fold_left (+) 0 ;;
282
283 (* and a version of `prods` is *)
284
285 let prods =
286   List.map (fun (x, y) -> x * y) ;;
287
288 (* Notice how in `dotprod_opt` and other option-manipulating functions
289 we frequently and annoyingly have to test if a value of option type is
290 `None`; this requires a separate match, and passing on the `None`
291 value in the "bad" branch and introducing a `Some` in the "good"
292 branch. This is something we're likely to be doing a lot of. Let's
293 factor that out to simplify the implementation.
294
295 .....
296 Exercise 9: Define a function called `maybe` that takes a first
297 argument, a function of type `arg -> result`, and a second argument,
298 of type `arg option`, and "maybe" applies the first (the function) to
299 the second (the argument), depending on whether its argument is a
300 `None` or a `Some`. The `maybe` function either passes on the `None`
301 if its second argument is `None`, or if its second argument is `Some
302 v`, it applies its first argument to that `v` and returns the result,
303 appropriately adjusted for the result type.
304
305 What should the type of the `maybe` function be?
306
307 Now implement the `maybe` function.
308 .....
309
310 let maybe (f : arg -> result) (x : arg option) : result option =
311   match x with
312   | None -> None
313   | Some v -> Some (f v) ;;
314
315 (*.....
316 Exercise 10: Now reimplement `dotprod_opt` to use the `maybe`
317 function. (The previous implementation makes use of functions `sum`
318 and `prods`, which we've provided for you above.) Your new solution
319 for `dotprod` should be much simpler than the version we provided
320 above at the top of Part 3.

```

```

321 .....*)
322
323 let dotprod_opt (a : int list) (b : int list) : int option =
324   maybe (fun pairs -> sum (prods pairs))
325     (zip_opt a b) ;;
326
327 (*.....
328 Exercise 11: Reimplement `zip_opt` using the `maybe` function, as
329 `zip_opt_2` below.
330 .....*)
331
332 (* We remove the embedded match using a maybe: *)
333
334 let rec zip_opt_2 (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
335   match x, y with
336   | [], [] -> Some []
337   | xhd :: xtl, yhd :: ytl ->
338     maybe (fun ztl -> ((xhd, yhd) :: ztl))
339       (zip_opt_2 xtl ytl)
340   | _, _ -> None ;;
341
342 (*.....
343 Exercise 12: [Optional] For the energetic, reimplement `max_list_opt`
344 as `max_list_opt_2` along the same lines. There's likely to be a
345 subtle issue here, since the `maybe` function always passes along the
346 `None`.
347 .....*)
348
349 let rec max_list_opt_2 (lst : int list) : int option =
350   match lst with
351   | [] -> None
352   | [single] -> Some single
353   | head :: tail ->
354     maybe (fun max_tail -> max head max_tail)
355       (max_list_opt_2 tail) ;;
356
357 (* The subtle issue is this. Recall the previous definition of
358   `max_list_opt` above:
359
360   let rec max_list_opt (lst : int list) : int option =
361     match lst with
362     | [] -> None
363     | head :: tail ->
364       match (max_list_opt tail) with
365       | None -> Some head
366       | Some max_tail -> Some (max head max_tail) ;;

```

367
368 In this version, no special match case is needed for the case of a
369 singleton list. Instead, we can recur all the way to the empty list
370 case, and handle the singleton case in the first case in the
371 embedded match, where the ‘None’ from the recursive call becomes
372 ‘Some head’. However, when using the ‘maybe’ in the corresponding
373 case in ‘max_list_opt_2’, we can’t allow the recursion to proceed
374 all the way to the empty list, where ‘None’ would be returned,
375 because ‘maybe’ always preserves ‘None’s; we can’t “promote” the
376 ‘None’ to a ‘Some’. Consequently, we need to handle the singleton
377 case explicitly. *)