

```

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

```

```

45 Reimplement `max_list`, but this time, it should return an `int option`
46 instead of an `int`. Call it `max_list_opt`. The `None` return value
47 should be used when called on an empty list.
48
49 (Using the suffix `_opt` is a standard convention in OCaml for
50 functions that return an option type for this purpose. See, for
51 instance, the functions `nth` and `nth_opt` in the `List` module.)
52 .....*)
53
54 let max_list_opt (lst : int list) : int option =
55   failwith "max_list_opt not implemented" ;;
56
57 (*.....
58 Exercise 2: Alternatively, we could have `max_list` raise an exception
59 upon discovering the error condition. Reimplement `max_list` so that it
60 does so. What exception should it raise? (See Section 10.3 in the
61 textbook for some advice.)
62 .....*)
63
64 let max_list (lst : int list) : int =
65   failwith "max_list not implemented" ;;
66
67 (*.....
68 Exercise 3: Write a function `min_option` to return the smaller of its
69 two `int option` arguments, or `None` if both are `None`. If exactly one
70 argument is `None`, return the other. The built-in function `min` from
71 the Stdlib module may be useful. You'll want to make sure that all
72 possible cases are handled; no nonexhaustive match warnings!
73 .....*)
74
75 let min_option (x : int option) (y : int option) : int option =
76   failwith "min_option not implemented" ;;
77
78 (*.....
79 Exercise 4: Write a function `plus_option` to return the sum of its two
80 `int option` arguments, or `None` if both are `None`. If exactly one
81 argument is `None`, return the other.
82 .....*)
83
84 let plus_option (x : int option) (y : int option) : int option =
85   failwith "plus_option not implemented" ;;
86
87 (=====
88 Part 2: Polymorphism practice
89
90 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     Note: You might encounter inexplicable "weak type variable"
119     warnings. If you do, you should make sure to type the arguments of
120     the function. For more information read the detailed explanation
121     in the lab's solution comments or Section 9.6 of the textbook.
122 .....*)
123
124 let min_option_2 =
125   fun _ -> failwith "min_option_2 not implemented" ;;
126
127 let plus_option_2 =
128   fun _ -> failwith "plus_option_2 not implemented" ;;
129
130 (*.....
131 Exercise 7: Now that we have `lift_option`, we can use it in other
132 ways. Because `lift_option` is polymorphic, it can work on things other
133 than `int option`s. Define a function `and_option` to return the boolean
134 AND of two `bool option`s, or `None` if both are `None`. If exactly one
135 is `None`, return the other.
136 .....*)

```

```

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

```

```

183 `sum` to sum up all the integers in a list. The `sum` function is
184 simply *)
185
186 let sum : int list -> int =
187   List.fold_left (+) 0 ;;
188
189 (* and a version of `prods` is *)
190
191 let prods =
192   List.map (fun (x, y) -> x * y) ;;
193
194 (* Notice how in `dotprod_opt` and other option-manipulating functions
195 we frequently and annoyingly have to test if a value of option type is
196 `None`; this requires a separate match, and passing on the `None`
197 value in the "bad" branch and introducing a `Some` in the "good"
198 branch. This is something we're likely to be doing a lot of. Let's
199 factor that out to simplify the implementation.
200
201 .....
202 Exercise 9: Define a function called `maybe` that takes a first
203 argument, a function of type `'arg -> 'result`, and a second argument,
204 of type `'arg option`, and "maybe" applies the first (the function) to
205 the second (the argument), depending on whether its argument is a
206 `None` or a `Some`. The `maybe` function either passes on the `None`
207 if its second argument is `None`, or if its second argument is `Some
208 v`, it applies its first argument to that `v` and returns the result,
209 appropriately adjusted for the result type.
210
211 What should the type of the `maybe` function be?
212
213 Now implement the `maybe` function.
214 .....*)
215
216 let maybe (f : 'arg -> 'result) (x : 'arg option) : 'result option =
217   failwith "maybe not implemented" ;;
218
219 (*.....
220 Exercise 10: Now reimplement `dotprod_opt` to use the `maybe`
221 function. (The previous implementation makes use of functions `sum`
222 and `prods`, which we've provided for you above.) Your new solution
223 for `dotprod` should be much simpler than the version we provided
224 above at the top of Part 3.
225 .....*)
226
227 let dotprod_opt (a : int list) (b : int list) : int option =
228   failwith "dotprod_opt not implemented" ;;

```

```

229
230 (*.....
231 Exercise 11: Reimplement `zip_opt` using the `maybe` function, as
232 `zip_opt_2` below.
233 .....*)
234
235 let rec zip_opt_2 (x : 'a list) (y : 'b list) : (('a * 'b) list) option =
236   failwith "zip_opt_2 not implemented" ;;
237
238 (*.....
239 Exercise 12: [Optional] For the energetic, reimplement `max_list_opt`
240 as `max_list_opt_2` along the same lines. There's likely to be a
241 subtle issue here, since the `maybe` function always passes along the
242 `None`.
243 .....*)
244
245 let rec max_list_opt_2 (lst : int list) : int option =
246   failwith "max_list_opt_2 not implemented" ;;

```