

```

1   (*
2           CS51 Lab 2
3           More Functional Programming:
4           Simple Data Structures and Higher-Order Functions
5   *)
6   (*
7           SOLUTION
8   *)
9   (*=====
10  Readings:
11
12     This lab builds on material from Chapters 7-8 of the textbook
13     <http://book.cs51.io>, which should be read before the lab session.
14
15  Objective:
16
17     This lab is intended to introduce you to staples of functional
18     programming in OCaml, including:
19
20         * simple data structures like lists and tuples
21         * higher-order functional programming (functions as first-class
22             values)
23 =====
24
25 =====
26 Part 1: Types and type inference beyond atomic types
27
28 Exercise 1: What are appropriate types to replace the ??? in the
29 expressions below? For each expression, replace the ??? with the
30 type. Test your solution by uncommenting the examples (removing the
31 `(*` and `*)` lines at start and end) and verifying that no typing
32 error is generated.
33 .....*)
34
35 let exercise1a : float * string =
36   (0.1, "hi") ;;
37
38 let exercise1b : string list =
39   let add_to_hello_list x = ["Hello"; x]
40   in add_to_hello_list "World!";;
41
42 let exercise1c : int * float -> int  =
43   fun (x, y) -> x + int_of_float y ;;
44

```

```

45 let exercise1d : int list -> bool =
46   fun lst ->
47     match lst with
48     | [] -> false
49     | hd :: _ -> hd < hd + 1;;
50
51 (* Here we deconstruct our argument as a list, so the argument `lst`
52 must be of type `list` (consistent with its name). In the
53 second match case, we compare the head of the list with itself plus
54 one. We use integer addition, so the head must be of type
55 `int`. All elements of a list must be of the same type, so the
56 argument is of type `int list`. To determine the type of the
57 result, we can look to the first match case. The literal `false` is
58 of type `bool`. All match cases must return the same type, so the
59 result must be of type `bool`. *)
60
61 let exercise1e : bool -> bool list =
62   fun x -> if x then [x] else [];;
63
64 (* The reasoning goes like this: The argument of the function is
65 `x`. Since `x` is used as the condition part of an `if` expression,
66 it must be of type `bool`. Thus the expression in the `then` clause
67 `[]` must be a `bool list`, and since the type of an `if` expression
68 is the type of its `then` and `else` expressions, the whole `if`
69 expression, the result of the function, must be a `bool list`. The
70 function itself is then of function type `bool -> bool list`, as is
71 the value named `exercise1e`. *)
72
73 (*.....*)
74 Exercise 2: Update each expression below by changing the 0 in the last
75 line to an integer literal so that the expression as a whole evaluates
76 to `true`.
77 .....*)
```

78

```

79 let exercise2a =
80   let lst = [1; 2; 3; 4] in
81   let value =
82     match lst with
83     | [] -> 0
84     | [h] -> h
85     | h1 :: h2 :: t -> h2 in
86   value = 2;;
87
88 let exercise2b =
89   let x, y, z = 4, [1; 3], true in
90   let value =
```



```

137     The order of the two match cases is crucial. If they are flipped,
138     the second match case will never be invoked, since the first
139     match case will always match. *)
140
141 (*=====
142 Part 2: First-order functional programming with lists
143
144 We'll continue with some "finger exercises" defining simple functions
145 before moving on to more complex problems. The intention in this part
146 of the lab is for you to implement these functions by *explicit
147 recursion*. Only later, in part 3 of this lab, will we make use of the
148 `map`/`fold`/`filter` higher-order functions.
149
150 As a reminder, here's the definition for the `length` function of type
151 `int list -> int` implemented in this explicit recursion style:
152
153     let rec length (lst : int list) : int =
154         match lst with
155             | [] -> 0
156             | _head :: tail -> 1 + length tail ;;
157
158 .....
159 Exercise 4: In lab 1, we defined a function that could square its
160 input. Now, define a function `square_all` that squares all of the
161 elements of an integer list. We've provided a bit of template code,
162 supplying the first line of the function definition, but the body of
163 the template code just causes a failure by forcing an error using the
164 built-in `failwith` function. Edit the code to implement `square_all`
165 properly.
166
167 Test out your implementation of `square_all` by modifying the template
168 code below to define `exercise4` to be the `square_all` function
169 applied to the list containing the elements `3`, `4`, and `5`. You'll
170 want to replace the `[]` with the correct function application.
171
172 Thorough testing is important in all your work, and we hope to impart
173 this view to you in CS51. Testing will help you find bugs, avoid
174 mistakes, and teach you the value of short, clear functions. In the
175 file `lab2_tests.ml`, we've put some prewritten tests for `square_all`
176 using the testing method of Section 6.7 in the book. Spend some time
177 understanding how the testing function works and why these tests are
178 comprehensive. Then test your code by compiling and running the test
179 suite:
180
181     % ocamlbuild -use-ocamlfind lab2_tests.byte
182     % ./lab2_tests.byte

```

```

183
184 You may want to add some tests for other functions in the lab to get
185 some practice with automated unit testing.
186 .....*)
187
188 let rec square_all (lst : int list) : int list =
189   match lst with
190   | [] -> []
191   | head :: tail -> (head * head) :: (square_all tail) ;;
192
193 let exercise4 =
194   square_all [3; 4; 5] ;;
195
196 (*.....
197 Exercise 5: Define a recursive function 'sum' that sums the values in
198 its integer list argument. (What's a sensible return value for the sum
199 of the empty list?)
200 ..)
201
202 let rec sum (lst : int list) : int =
203   match lst with
204   | [] -> 0
205   | head :: tail -> head + sum tail ;;
206
207 (*.....
208 Exercise 6: Define a recursive function 'max_list' that returns the
209 maximum element in a non-empty integer list. Don't worry about what
210 happens on an empty list. You may be warned by the compiler that "this
211 pattern-matching is not exhaustive." You may ignore this warning for
212 this lab.
213 ..)
214
215 (* Here's a first cut at a solution, using just the portion of OCaml
216 already introduced. Notice that there's no branch in the pattern
217 match that matches the empty list, because there is no maximum
218 element in the empty list! For that reason, the ocaml interpreter
219 warns us with an "inexhaustive pattern match" warning.
220
221 let rec max_list (lst : int list) : int =
222   match lst with
223   | [elt] -> elt
224   | head :: tail ->
225     let max_tail = max_list tail in
226       if head > max_tail then head else max_tail ;;
227
228 This is the solution we expected people to come up with. And it

```

```

229     seems to work.

230     # max_list [1; 3; 2] ;;
231     - : int = 3

233
234     What happens when we apply this function to the empty list?

235
236     # max_list [] ;;
237     Exception: Match_failure ("//toplevel//", 2, 2).

238
239     It generates a 'Match_failure' exception. (We'll talk more about
240     error handling and exceptions later in the course, and use them
241     starting in Lab 4.) This 'Match_failure' exception is a symptom of
242     a deeper underlying problem, namely, that the function 'max_list'
243     was called with an invalid argument. A better solution, then, and
244     one that not coincidentally eliminates the "inexhaustive pattern
245     match" warning, is to explicitly raise a more appropriate exception
246     like 'Invalid_argument', as we've done in the solution below. *)
247
248 let rec max_list (lst : int list) : int =
249   match lst with
250   | [] -> raise (Invalid_argument "max_list: empty list")
251   | [elt] -> elt
252   | head :: tail ->
253     let max_tail = max_list tail in
254     if head > max_tail then head else max_tail ;;

255
256 (* It turns out that the 'Stdlib' module has a 'max' function that
257    returns the larger of its two arguments. Using that function, we
258    can simplify a bit.
259
260 let rec max_list (lst : int list) : int =
261   match lst with
262   | [elt] -> elt
263   | head :: tail -> max head (max_list tail) ;;
264   *)
265
266 (*.....
267 Exercise 7: Define a function 'zip', that takes two 'int list'
268 arguments and returns a list of pairs of ints, one from each of the
269 two argument lists. Your function can assume the input lists will be
270 the same length. You can ignore what happens in the case the input
271 list lengths do not match. You may be warned by the compiler that
272 "this pattern-matching is not exhaustive." You may ignore this warning
273 for this lab.
274
```

```

275   For example,
276
277   fun lst -> zip [1; 2; 3] [4; 5; 6] ;;
278   - : (int * int) list = [(1, 4); (2, 5); (3, 6)]
279
280 To think about: Why wouldn't it be possible, in cases of mismatched
281 length lists, to just pad the shorter list with, say, 'false' values, so
282 that, `zip [1] [2; 3; 4] = [(1, 2); (false, 3); (false, 4)]`?
283 .....*)
284
285 let rec zip (x : int list) (y : int list) : (int * int) list =
286   match x, y with
287   | [], [] -> []
288   | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
289
290 (* This was the solution we expected people to come up with. It
291 generates a warning about the pattern match not being
292 exhaustive. As in 'max_list' above, the ramifications of this issue
293 and how best to address it are discussed at length in Chapter 10,
294 Section 10.2. That discussion is beyond the scope of lab 2, but feel
295 free to read ahead if you're interested. *)
296
297 (*.....
298 Exercise 8: Recall from Chapter 7 the definition of the function 'prods'.
299 *)
300
301 let rec prods (lst : (int * int) list) : int list =
302   match lst with
303   | [] -> []
304   | (x, y) :: tail -> (x * y) :: (prods tail) ;;
305
306 (* Using 'sum', 'prods', and 'zip', define a function 'dotprod' that
307 takes the dot product of two integer lists (that is, the sum of the
308 products of corresponding elements of the lists; see
309 https://en.wikipedia.org/wiki/Dot\_product if you want more
310 information, though it shouldn't be necessary). For example, you
311 should have:
312
313   # dotprod [1; 2; 3] [0; 1; 2] ;;
314   - : int = 8
315   # dotprod [1; 2] [5; 10] ;;
316   - : int = 25
317
318 Even without looking at the code for the functions, carefully looking
319 at the type signatures for 'zip', 'prods', and 'sum' should give a
320 good idea of how you might combine these functions to implement

```

```

321 `dotproduct`.
322
323 If you've got the right idea, your implementation should be literally
324 a single short line of code. If it isn't, try it again, getting into
325 the functional programming zen mindset.
326 .....*)
327
328 let dotprod (a : int list) (b : int list) : int =
329   sum (prods (zip a b)) ;;
330
331 (*=====
332 Part 3: Higher-order functional programming with map, filter, and fold
333
334 In these exercises, you should use the built-in functions `map`,
335 `filter`, and `fold_left` and `fold_right` provided in the OCaml List
336 module to implement these simple functions.
337
338 * IMPORTANT NOTE 1: When you make use of these functions, you'll
339 either need to prefix them with the module name, for example,
340 `List.map` or `List.fold_left`, or you'll need to open the `List`
341 module with the line
342
343     open List ;;
344
345 You can place that line at the top of this file if you'd like.
346
347 * IMPORTANT NOTE 2: In these labs, and in the problem sets as well,
348 we'll often supply some skeleton code that looks like this:
349
350     let somefunction (arg1 : type) (arg2 : type) : returntype =
351       failwith "somefunction not implemented"
352
353 We provide this to give you an idea of the function's intended
354 name, its arguments and their types, and the return type. But
355 there's no need to slavishly follow that particular way of
356 implementing code to those specifications. In particular, you may
357 want to modify the first line to introduce, say, a `rec` keyword
358 (if your function is to be recursive):
359
360     let rec somefunction (arg1 : type) (arg2 : type) : returntype =
361       ...your further code here...
362
363 Or you might want to define the function using anonymous function
364 syntax. (If you haven't seen this yet, come back to this comment
365 later when you have.)
366

```

```

367     let somefunction =
368         fun (arg1 : type) (arg2 : type) : returntype ->
369             ...your further code here...
370
371     This will be especially pertinent in this section, where functions
372     can be built just by applying other higher order functions
373     directly, without specifying the arguments explicitly, for
374     example, in this implementation of the `double_all` function,
375     which doubles each element of a list:
376
377     let double_all : int list -> int list =
378         map (( * ) 2) ;;
379
380     * END IMPORTANT NOTES
381
382     .....
383     Exercise 9: Reimplement `sum` using `fold_left`, naming it `sum_ho`
384     (for "higher order").
385     .....
386
387     let sum_ho : int list -> int =
388         List.fold_left (+) 0 ;;
389
390     (* One of the key advantages of curried functions (like `fold_left`)
391      is that they can be partially applied. (See Section 8.2 in the
392      textbook.) We've taken advantage of that in the definition above,
393      by defining `sum_ho` as a partially applied `fold_left`, rather
394      than as
395
396     let sum_ho (lst : int list) : int =
397         List.fold_left (+) 0 lst ;;
398
399     The latter will work, but lacks the elegance of the more idiomatic
400     approach here.
401
402     The same technique is used in the exercises below. It may be
403     useful, in order to understand what's going on, to try typing in
404     longer and longer prefixes of an expression like `List.fold_left
405     (+) 0 [1; 2; 3]` and watch the types closely.
406
407     # List.fold_left ;;
408     - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
409     # List.fold_left (+) ;;
410     - : int -> int list -> int = <fun>
411     # List.fold_left (+) 0 ;;
412     - : int list -> int = <fun>

```

```

413     # List.fold_left (+) 0 [1; 2; 3] ;;
414     - : int = 6
415
416 You may also note the use of parentheses in the expression
417 `(+)` . The `+` operator is an example of an "infix" operator, an
418 operator that goes in between its arguments rather than in front of
419 them. When using `+` as an argument to higher-order functions, we
420 generally need to remove that infix property, so that it will be
421 parsed as a prefix operator like most other functions. Wrapping the
422 operator in parentheses induces this behavior.
423
424     # 3 + 4 ;;
425     - : int = 7
426     # (+) 3 4
427     - : int = 7
428   *)
429
430 (*.....
431 Exercise 10: Reimplement prods : `(int * int) list -> int list` using
432 the `map` function. Call it `prods_ho`.
433 .....*)
434
435 let prods_ho : (int * int) list -> int list =
436   List.map (fun (x, y) -> x * y) ;;
437
438 (*.....
439 Exercise 11: The OCaml List module provides -- in addition to the `map` ,
440 `fold_left` , and `fold_right` higher-order functions -- several other
441 useful higher-order list manipulation functions. For instance, `map2` is
442 like `map` , but takes two lists instead of one along with a function of
443 two arguments and applies the function to corresponding elements of the
444 two lists to form the result list. (You can read about it at
445 https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html#VALmap2.)
446 Use `map2` to reimplement `zip` and call it `zip_ho`.
447 .....*)
448
449 let zip_ho : int list -> int list -> (int * int) list =
450   List.map2 (fun first second -> first, second) ;;
451
452 (* Note the rejigging of the first line to allow the function
453   `zip_ho` to take advantage of partial application, so that `zip_ho`
454   is the functional output of the higher-order function
455   `map2` . Without the rejigging, you'd probably implement it as:
456
457 let zip_ho (x : int list) (y : int list) : (int * int) list =
458   List.map2 (fun first second -> first, second) x y ;;

```

```
459 *)  
460  
461 (*.....  
462 Exercise 12: Define a function `evens`, using these higher-order  
463 functional programming techniques, that returns a list of all of the  
464 even numbers in its argument list in the same order. For instance,  
465  
466     # evens [1; 2; 3; 6; 5; 4] ;;  
467     - : int list = [2; 6; 4]  
468 .....*)  
469  
470 (* Again, without partial application:  
471  
472     let evens (lst : int list) : int list =  
473         List.filter (fun n -> n mod 2 = 0) lst;;  
474  
475 and with partial application: *)  
476  
477 let evens : int list -> int list =  
478     List.filter (fun n -> n mod 2 = 0) ;;
```