

```

1  (*
2                                     CS51 Lab 2
3                                     More Functional Programming:
4                                     Simple Data Structures and Higher-Order Functions
5  *)
6  (*=====
7  Readings:
8
9      This lab builds on material from Chapters 7-8 of the textbook
10     <http://book.cs51.io>, which should be read before the lab session.
11
12 Objective:
13
14     This lab is intended to introduce you to staples of functional
15     programming in OCaml, including:
16
17         * simple data structures like lists and tuples
18         * higher-order functional programming (functions as first-class
19           values)
20     =====*)
21
22  (*=====
23  Part 1: Types and type inference beyond atomic types
24
25  Exercise 1: What are appropriate types to replace the ??? in the
26  expressions below? For each expression, replace the ??? with the
27  type. Test your solution by uncommenting the examples (removing the
28  `(*` and `*)` lines at start and end) and verifying that no typing
29  error is generated.
30  .....*)
31
32  (* <--- remove this start of comment line
33
34  let exercise1a : ??? =
35      (0.1, "hi") ;;
36
37  let exercise1b : ??? =
38      let add_to_hello_list x = ["Hello"; x]
39      in add_to_hello_list "World!";;
40
41  let exercise1c : ??? =
42      fun (x, y) -> x + int_of_float y ;;
43
44  let exercise1d : ??? =

```

```

45     fun lst ->
46         match lst with
47         | [] -> false
48         | hd :: _ -> hd < hd + 1 ;;
49
50 let exercise1e : ??? =
51     fun x -> if x then [x] else [] ;;
52
53 remove this end of comment line too ----> *)
54
55 (*.....
56 Exercise 2: Update each expression below by changing the 0 in the last
57 line to an integer literal so that the expression as a whole evaluates
58 to `true`.
59 .....*)
60
61 let exercise2a =
62     let lst = [1; 2; 3; 4] in
63     let value =
64         match lst with
65         | [] -> 0
66         | [h] -> h
67         | h1 :: h2 :: t -> h2 in
68     value = 0 ;;
69
70 let exercise2b =
71     let x, y, z = 4, [1; 3], true in
72     let value =
73         match y with
74         | [] -> 0
75         | h :: t -> h in
76     value = 0 ;;
77
78 let exercise2c =
79     let tuple_lst = [(1, 4); (5, 2)] in
80     let value =
81         match tuple_lst with
82         | [] -> 0
83         | (a, b) :: t -> a
84         | h1 :: (a, b) :: t -> a in
85     value = 0 ;;
86
87 let exercise2d =
88     let tuple_lst = [(1, 4); (5, 2)] in
89     let value =
90         match tuple_lst with

```

```

91     | [] -> 0
92     | h1 :: (a, b) :: t -> a
93     | (a, b) :: t -> a in
94     value = 0 ;;
95
96     (*.....
97     Exercise 3: Complete the following definition for a function
98     `third_element` that returns a `bool * int` pair, whose first element
99     represents whether or not its list argument has a third element, and
100    whose second element represents that element if it exists (or 0 if it
101    does not). Here are some examples of the intended behavior of this
102    function:
103
104        # third_element [1; 2; 3; 4; 5] ;;
105        - : bool * int = (true, 3)
106        # third_element [] ;;
107        - : bool * int = (false, 0)
108    .....*)
109
110    let third_element (lst : int list) : bool * int =
111        match lst with
112        | _ -> false, 0 ;;
113
114    (*=====
115    Part 2: First-order functional programming with lists
116
117    We'll continue with some "finger exercises" defining simple functions
118    before moving on to more complex problems. The intention in this part
119    of the lab is for you to implement these functions by *explicit
120    recursion*. Only later, in part 3 of this lab, will we make use of the
121    `map`/`fold`/`filter` higher-order functions.
122
123    As a reminder, here's the definition for the `length` function of type
124    `int list -> int` implemented in this explicit recursion style:
125
126        let rec length (lst : int list) : int =
127            match lst with
128            | [] -> 0
129            | _head :: tail -> 1 + length tail ;;
130
131    .....
132    Exercise 4: In lab 1, we defined a function that could square its
133    input. Now, define a function `square_all` that squares all of the
134    elements of an integer list. We've provided a bit of template code,
135    supplying the first line of the function definition, but the body of
136    the template code just causes a failure by forcing an error using the

```

```

137 built-in `failwith` function. Edit the code to implement `square_all`
138 properly.
139
140 Test out your implementation of `square_all` by modifying the template
141 code below to define `exercise4` to be the `square_all` function
142 applied to the list containing the elements `3`, `4`, and `5`. You'll
143 want to replace the `[]` with the correct function application.
144
145 Thorough testing is important in all your work, and we hope to impart
146 this view to you in CS51. Testing will help you find bugs, avoid
147 mistakes, and teach you the value of short, clear functions. In the
148 file `lab2_tests.ml`, we've put some prewritten tests for `square_all`
149 using the testing method of Section 6.7 in the book. Spend some time
150 understanding how the testing function works and why these tests are
151 comprehensive. Then test your code by compiling and running the test
152 suite:
153
154     % ocamlbuild -use-ocamlfind lab2_tests.byte
155     % ./lab2_tests.byte
156
157 You may want to add some tests for other functions in the lab to get
158 some practice with automated unit testing.
159 .....*)
160
161 let rec square_all (lst : int list) : int list =
162     failwith "square_all not implemented" ;;
163
164 let exercise4 = [] ;;
165
166 (*.....
167 Exercise 5: Define a recursive function `sum` that sums the values in
168 its integer list argument. (What's a sensible return value for the sum
169 of the empty list?)
170 .....*)
171
172 let rec sum (lst : int list) : int =
173     failwith "sum not implemented" ;;
174
175 (*.....
176 Exercise 6: Define a recursive function `max_list` that returns the
177 maximum element in a non-empty integer list. Don't worry about what
178 happens on an empty list. You may be warned by the compiler that "this
179 pattern-matching is not exhaustive." You may ignore this warning for
180 this lab.
181 .....*)
182

```

```

183 let rec max_list (lst : int list) : int =
184     failwith "max_list not implemented" ;;
185
186 (*.....*)
187 Exercise 7: Define a function `zip`, that takes two `int list`
188 arguments and returns a list of pairs of ints, one from each of the
189 two argument lists. Your function can assume the input lists will be
190 the same length. You can ignore what happens in the case the input
191 list lengths do not match. You may be warned by the compiler that
192 "this pattern-matching is not exhaustive." You may ignore this warning
193 for this lab.
194
195 For example,
196
197     fun lst -> zip [1; 2; 3] [4; 5; 6] ;;
198     - : (int * int) list = [(1, 4); (2, 5); (3, 6)]
199
200 To think about: Why wouldn't it be possible, in cases of mismatched
201 length lists, to just pad the shorter list with, say, `false` values, so
202 that, `zip [1] [2; 3; 4] = [(1, 2); (false, 3); (false, 4)]`?
203 .....*)
204
205 let rec zip (x : int list) (y : int list) : (int * int) list =
206     failwith "zip not implemented" ;;
207
208 (*.....*)
209 Exercise 8: Recall from Chapter 7 the definition of the function `prods`.
210 *)
211
212 let rec prods (lst : (int * int) list) : int list =
213     match lst with
214     | [] -> []
215     | (x, y) :: tail -> (x * y) :: (prods tail) ;;
216
217 (* Using `sum`, `prods`, and `zip`, define a function `dotprod` that
218 takes the dot product of two integer lists (that is, the sum of the
219 products of corresponding elements of the lists; see
220 https://en.wikipedia.org/wiki/Dot\_product if you want more
221 information, though it shouldn't be necessary). For example, you
222 should have:
223
224     # dotprod [1; 2; 3] [0; 1; 2] ;;
225     - : int = 8
226     # dotprod [1; 2] [5; 10] ;;
227     - : int = 25
228

```

```

229 Even without looking at the code for the functions, carefully looking
230 at the type signatures for `zip`, `prods`, and `sum` should give a
231 good idea of how you might combine these functions to implement
232 `dotproduct`.
233
234 If you've got the right idea, your implementation should be literally
235 a single short line of code. If it isn't, try it again, getting into
236 the functional programming zen mindset.
237 .....*)
238
239 let dotprod (a : int list) (b : int list) : int =
240   failwith "dotprod not implemented" ;;
241
242 (*=====
243 Part 3: Higher-order functional programming with map, filter, and fold
244
245 In these exercises, you should use the built-in functions `map`,
246 `filter`, and `fold_left` and `fold_right` provided in the OCaml List
247 module to implement these simple functions.
248
249 * IMPORTANT NOTE 1: When you make use of these functions, you'll
250 either need to prefix them with the module name, for example,
251 `List.map` or `List.fold_left`, or you'll need to open the `List`
252 module with the line
253
254     open List ;;
255
256 You can place that line at the top of this file if you'd like.
257
258 * IMPORTANT NOTE 2: In these labs, and in the problem sets as well,
259 we'll often supply some skeleton code that looks like this:
260
261     let somefunction (arg1 : type) (arg2 : type) : returntype =
262       failwith "somefunction not implemented"
263
264 We provide this to give you an idea of the function's intended
265 name, its arguments and their types, and the return type. But
266 there's no need to slavishly follow that particular way of
267 implementing code to those specifications. In particular, you may
268 want to modify the first line to introduce, say, a `rec` keyword
269 (if your function is to be recursive):
270
271     let rec somefunction (arg1 : type) (arg2 : type) : returntype =
272       ...your further code here...
273
274 Or you might want to define the function using anonymous function

```

```

275     syntax. (If you haven't seen this yet, come back to this comment
276     later when you have.)
277
278     let somefunction =
279         fun (arg1 : type) (arg2 : type) : returntype ->
280             ...your further code here...
281
282     This will be especially pertinent in this section, where functions
283     can be built just by applying other higher order functions
284     directly, without specifying the arguments explicitly, for
285     example, in this implementation of the 'double_all' function,
286     which doubles each element of a list:
287
288     let double_all : int list -> int list =
289         map (( * ) 2) ;;
290
291     * END IMPORTANT NOTES
292
293     .....
294     Exercise 9: Reimplement 'sum' using 'fold_left', naming it 'sum_ho'
295     (for "higher order").
296     .....*)
297
298     let sum_ho (lst : int list) : int =
299         failwith "sum_ho not implemented" ;;
300
301     (*.....
302     Exercise 10: Reimplement prods : '(int * int) list -> int list' using
303     the 'map' function. Call it 'prods_ho'.
304     .....*)
305
306     let prods_ho (lst : (int * int) list) : int list =
307         failwith "prods_ho not implemented" ;;
308
309     (*.....
310     Exercise 11: The OCaml List module provides -- in addition to the 'map',
311     'fold_left', and 'fold_right' higher-order functions -- several other
312     useful higher-order list manipulation functions. For instance, 'map2' is
313     like 'map', but takes two lists instead of one along with a function of
314     two arguments and applies the function to corresponding elements of the
315     two lists to form the result list. (You can read about it at
316     https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html#VALmap2.)
317     Use 'map2' to reimplement 'zip' and call it 'zip_ho'.
318     .....*)
319
320     let zip_ho (x : int list) (y : int list) : (int * int) list =

```

```

321     failwith "zip_ho not implemented" ;;
322
323     (*.....
324     Exercise 12: Define a function `evens`, using these higher-order
325     functional programming techniques, that returns a list of all of the
326     even numbers in its argument list in the same order. For instance,
327
328         # evens [1; 2; 3; 6; 5; 4] ;;
329         - : int list = [2; 6; 4]
330     .....*)
331
332     let evens (lst : int list) : int list =
333         failwith "evens not implemented" ;;

```