

```

1  (*
2
3          CS51 Lab 2
4          More Functional Programming:
5          Simple Data Structures and Higher-Order Functions
6  *)
7  (*=====
8  Readings:
9
10     This lab builds on material from Chapters 7-8 of the textbook
11     <http://book.cs51.io>, which should be read before the lab session.
12
13     Objective:
14
15     This lab is intended to introduce you to staples of functional
16     programming in OCaml, including:
17
18         * simple data structures like lists and tuples
19         * higher-order functional programming (functions as first-class
20           values)
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 : ??? =

```



```

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

```



```

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 =

```

