

```

1  (*)
2                                     CS51 Lab 3
3                                     Polymorphism and record types
4  *)
5
6
7  (=====
8  Readings:
9
10     This lab builds on material from Chapters 7.4 and 9-9.5 of the
11     textbook <http://book.cs51.io>, which should be read before the lab
12     session.
13
14  Objective:
15
16     In this lab, you'll exercise your understanding of polymorphism and
17     record types. Some of the problems extend those from Lab 2, but we'll
18     provide the necessary background code from that lab.
19  =====*)
20
21  (=====
22  Part 1: Records and tuples
23
24  Records and tuples provide two different ways to package together
25  data. They differ in whether their components are selected by *name*
26  or by *position*, respectively.
27
28  Consider a point in Cartesian (x-y) coordinates. A point is specified
29  by its x and y values, which we'll take to be ints. We can package
30  these together as a pair (a 2-tuple), as in the following data type
31  definition: *)
32
33  type point_pair = int * int ;;
34
35  (* Then, we can add two points (summing their x and y coordinates
36  separately) with the following function:
37
38      let add_point_pair (p1 : point_pair) (p2 : point_pair) : point_pair =
39          let x1, y1 = p1 in
40          let x2, y2 = p2 in
41          (x1 + x2, y1 + y2) ;;
42
43  .....
44  Exercise 1:

```

```

45
46 It might be nicer to deconstruct the points in a single match, rather
47 than the two separate matches in the two `let` expressions. Reimplement
48 `add_point_pair` to use a single pattern match in a single `let`
49 expression.
50 .....*)
51
52 let add_point_pair (p1 : point_pair) (p2 : point_pair) : point_pair =
53   failwith "add_point_pair not implemented" ;;
54
55 (* Analogously, we can define a point by using a record to package up
56 the x and y coordinates. *)
57
58 type point_recd = {x : int; y : int} ;;
59
60 let example_point_recd = {x = 1; y = 3} ;;
61
62 (*.....
63 Exercise 2A:
64
65 Replace the two lines below with a single top-level `let` expression
66 that extracts the x and y coordinate values from `example_point_recd`
67 above into global variables `x1` and `y1`, respectively.
68 .....*)
69
70 let x1 = 0 ;;
71 let y1 = 0 ;;
72
73 (*.....
74 Exercise 2B:
75
76 Implement a function `add_point_recd` to add two points of type
77 `point_recd` and returning a `point_recd` as well.
78 .....*)
79
80 let add_point_recd =
81   fun _ -> failwith "add_point_recd not implemented" ;;
82
83 (* Recall the dot product from Lab 2. The dot product of two points
84 `x1, y1` and `x2, y2` is the sum of the products of their x and y
85 coordinates.
86
87 .....
88 Exercise 3: Write a function `dot_product_pair` to compute the dot
89 product for points encoded as the `point_pair` type.
90 .....*)

```

```

91
92 let dot_product_pair (p1 : point_pair) (p2 : point_pair) : int =
93     failwith "dot_product_pair not implemented" ;;
94
95 (*.....*)
96 Exercise 4: Write a function `dot_product_recd` to compute the dot
97 product for points encoded as the `point_recd` type.
98 .....*)
99
100 let dot_product_recd (p1 : point_recd) (p2 : point_recd) : int =
101     failwith "dot_product_recd not implemented" ;;
102
103 (* Converting between the pair and record representations of points
104
105 You might imagine that the two representations have
106 different advantages, such that two libraries, say, might use
107 differing types for points. In that case, we may want to have
108 functions to convert between the two representations.
109
110 .....*)
111 Exercise 5: Write a function `point_pair_to_recd` that converts a
112 `point_pair` to a `point_recd`.
113 .....*)
114
115 let point_pair_to_recd =
116     fun _ -> failwith "point_pair_to_recd not implemented" ;;
117
118 (*.....*)
119 Exercise 6: Write a function `point_recd_to_pair` that converts a
120 `point_recd` to a `point_pair`.
121 .....*)
122
123 let point_recd_to_pair =
124     fun _ -> failwith "point_recd_to_pair not implemented" ;;
125
126 (*=====*)
127 Part 2: A simple database of records
128
129 A college wants to store student records in a simple database,
130 implemented as a list of individual course enrollments. The enrollments
131 themselves are implemented as a record type, called `enrollment`, with
132 `string` fields labeled `name` and `course` and an integer student ID
133 number labeled `id`. The appropriate type definition is: *)
134
135 type enrollment = { name : string;
136                     id : int;

```

```

137         course : string } ;;
138
139 (* Here's an example of a list of enrollments. *)
140
141 let college =
142   [ { name = "Pat";    id = 603858772; course = "cs51" };
143     { name = "Pat";    id = 603858772; course = "expos20" };
144     { name = "Kim";    id = 482958285; course = "expos20" };
145     { name = "Kim";    id = 482958285; course = "cs20" };
146     { name = "Sandy";  id = 993855891; course = "ls1b" };
147     { name = "Pat";    id = 603858772; course = "ec10b" };
148     { name = "Sandy";  id = 993855891; course = "cs51" };
149     { name = "Sandy";  id = 482958285; course = "ec10b" }
150   ] ;;
151
152 (* In the following exercises, you'll want to avail yourself of the
153 'List' module functions, writing the requested functions in
154 higher-order style rather than handling the recursion yourself.
155
156 .....
157 Exercise 7: Define a function called 'transcript' that takes an
158 'enrollment list' and returns a list of all the enrollments for a given
159 student as specified by the student's ID.
160
161 For example:
162
163     # transcript college 993855891 ;;
164     - : enrollment list =
165       [{name = "Sandy"; id = 993855891; course = "ls1b"};
166        {name = "Sandy"; id = 993855891; course = "cs51"}]
167     .....*)
168
169 let transcript (enrollments : enrollment list)
170   (student : int)
171   : enrollment list =
172   failwith "transcript not implemented" ;;
173
174 (*.....
175 Exercise 8: Define a function called 'ids' that takes an 'enrollment
176 list' and returns a list of all the ID numbers in that list,
177 eliminating any duplicates. Hint: The 'map' and 'sort_uniq' functions
178 from the 'List' module and the 'compare' function from the 'Stdlib'
179 module may be useful here.
180
181 For example:
182

```

```

183     # ids college ;;
184     - : int list = [482958285; 603858772; 993855891]
185     .....*)
186
187 let ids (enrollments: enrollment list) : int list =
188     failwith "ids not implemented" ;;
189
190 (* There's a big problem with this database design: nothing guarantees
191 that a given student ID is associated with a single name. The right
192 thing to do is to use a different database design where this kind of
193 thing can't happen; that would be an application of the *edict of
194 prevention*. But for the purpose of this lab, you'll just write a
195 function to verify that this problem doesn't occur.
196
197 .....
198 Exercise 9: Define a function `verify` that determines whether all the
199 entries in an enrollment list for each of the ids appearing in the
200 list have the same name associated. Hint: You may want to use
201 functions from the `List` module such as `map`, `for_all`,
202 `sort_uniq`.
203
204 For example:
205
206     # verify college ;;
207     - : bool = false
208
209 (Do you see why it's false?)
210 .....*)
211
212 let verify (enrollments : enrollment list) : bool =
213     failwith "verify not implemented" ;;
214
215 (*=====
216 Part 3: Polymorphism
217
218 .....
219 Exercise 10: In Lab 2, you implemented a function `zip` that takes two
220 lists and "zips" them together into a list of pairs. Here's a possible
221 implementation of `zip`:
222
223     let rec zip (x : int list) (y : int list) : (int * int) list =
224         match x, y with
225         | [], [] -> []
226         | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
227
228 As implemented, this function works only on integer lists. Revise your

```

```

229 solution to operate polymorphically on lists of any type. What is the
230 type of the result? Did you provide full typing information in the
231 first line of the definition? (As usual, for the time being, don't
232 worry about explicitly handling the anomalous case when the two lists
233 are of different lengths.)
234 .....*)
235
236 let zip =
237   fun _ -> failwith "zip not implemented" ;;
238
239 (*.....
240 Exercise 11: Partitioning a list -- Given a boolean function, say
241
242     fun x -> x mod 3 = 0
243
244 and a list of elements, say,
245
246     [3; 4; 5; 10; 11; 12; 1]
247
248 we can partition the list into two lists, the list of elements
249 satisfying the boolean function (`[3; 12]`) and the list of elements
250 that don't (`[4; 5; 10; 11; 1]`).
251
252 The library function `List.partition` partitions its list argument in
253 just this way, returning a pair of lists. Here's an example:
254
255     # List.partition (fun x -> x mod 3 = 0) [3; 4; 5; 10; 11; 12; 1] ;;
256     - : int list * int list = ([3; 12], [4; 5; 10; 11; 1])
257
258 What is the type of the `partition` function, keeping in mind that it
259 should be as polymorphic as possible?
260
261 Now implement the function yourself (without using `List.partition` of
262 course, though other `List` module functions may be useful).
263 .....*)
264
265 let partition =
266   fun _ -> failwith "partition not implemented" ;;
267
268 (*=====
269 Part 4: Implementing polymorphic application, currying, and uncurrying
270
271 .....
272 Exercise 12: We can think of function application itself as a
273 polymorphic higher-order function (:exploding_head:). It takes two
274 arguments -- a function and its argument -- and returns the value

```

```

275 obtained by applying the function to its argument. In this exercise,
276 you'll write this function, called `apply`. You might use it as in the
277 following examples:
278
279     # apply pred 42 ;;
280     - : int = 41
281     # apply (fun x -> x ** 2.) 3.14159 ;;
282     - : float = 9.86958772809999907
283
284     An aside: You may think such a function isn't useful, since we
285     already have an even more elegant notation for function
286     application, as in
287
288         # pred 42 ;;
289         - : int = 41
290         # (fun x -> x ** 2.) 3.14159 ;;
291         - : float = 9.86958772809999907
292
293     But we'll see a quite useful operator that works similarly --
294     the backwards application operator -- in Chapter 11 of the
295     textbook.
296
297 Start by thinking about the type of the function. We'll assume it
298 takes its two arguments curried, that is, one at a time.
299
300 1. What is the most general (polymorphic) type for its first argument
301    (the function to be applied)?
302
303 2. What is the most general type for its second argument (the argument
304    to apply it to)?
305
306 3. What is the type of its result?
307
308 4. Given the above, what should the type of the function `apply` be?
309
310 Now write the function.
311
312 Can you think of a reason that the `apply` function might in fact be
313 useful?
314 .....*)
315
316 let apply =
317   fun _ -> failwith "apply not implemented" ;;
318
319 (*.....
320 Exercise 13: In the next two exercises, you'll define polymorphic

```

```

321 higher-order functions `curry` and `uncurry` for currying and uncurrying
322 binary functions (functions of two arguments). The functions are named
323 after mathematician Haskell Curry '1920. (By way of reminder, a
324 curried function takes its arguments one at a time. An uncurried
325 function takes them all at once in a tuple.)
326
327 We start with the polymorphic higher-order function `curry`, which
328 takes as its argument an uncurried binary function and returns the
329 curried version of its argument function.
330
331 Before starting to code, pull out a sheet of paper and a pencil and
332 with your partner work out the answers to the following seven
333 questions.
334
335 *****
336             Do not skip this pencil and paper work.
337 *****
338
339 1. What is the type of the argument to the function `curry`? Write down
340   a type expression for the argument type.
341
342 2. What is an example of a function that `curry` could apply to?
343
344 3. What is the type of the result of the function `curry`? Write down a
345   type expression for the result type.
346
347 4. What should the result of applying the function `curry` to the
348   function from (2) be?
349
350 5. Given (1) and (2), write down a type expression for the type of the
351   `curry` function itself.
352
353 6. What would a good variable name for the argument to `curry` be?
354
355 7. Write down the header line for the definition of the `curry` function.
356
357 Call over a staff member to go over your answers to these
358 questions. Once you fully understand all this, its time to implement
359 the function `curry`.
360 .....*)
361
362 let curry = fun _ -> failwith "curry not implemented" ;;
363
364 (*.....
365 Exercise 14: Now implement the polymorphic higher-order function
366 `uncurry`, which takes as its argument a curried function and returns

```



```

367 the uncurried version of its argument function. You may want to go
368 through the same 7-step process to get started.
369 .....*)
370
371 let uncurry = fun _ -> failwith "uncurry not implemented" ;;
372
373 (*.....
374 Exercise 15: OCaml's built in binary operators, like '+' and '*' are
375 curried. You can tell from their types:
376
377     # ( + ) ;;
378     - : int -> int -> int = <fun>
379     # ( * ) ;;
380     - : int -> int -> int = <fun>
381
382 Using your 'uncurry' function, define uncurried versions of the '+' and
383 '*' functions. Call them 'plus' and 'times'.
384 .....*)
385
386 let plus =
387     fun _ -> failwith "plus not implemented"
388
389 let times =
390     fun _ -> failwith "times not implemented" ;;
391
392 (*.....
393 Exercise 16: Recall the 'prods' function from Lab 1:
394
395     let rec prods (lst : (int * int) list) : int list =
396         match lst with
397         | [] -> []
398         | (x, y) :: tail -> (x * y) :: (prods tail) ;;
399
400 Now reimplement 'prods' using 'map' and your uncurried 'times'
401 function. Why do you need the uncurried 'times' function?
402 .....*)
403
404 let prods =
405     fun _ -> failwith "prods not implemented" ;;

```