

```

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

```

```

45
46 .....  

47 Exercise 1:  

48  

49 It might be nicer to deconstruct the points in a single match, rather
50 than the two separate matches in the two 'let' expressions. Reimplement
51 'add_point_pair' to use a single pattern match in a single 'let'
52 expression.  

53 .....*)  

54  

55 let add_point_pair (p1 : point_pair) (p2 : point_pair) : point_pair =  

56   let (x1, y1), (x2, y2) = p1, p2 in  

57     x1 + x2, y1 + y2 ;;  

58  

59 (* Analogously, we can define a point by using a record to package up
60 the x and y coordinates. *)  

61  

62 type point_recd = {x : int; y : int} ;;  

63  

64 let example_point_recd = {x = 1; y = 3} ;;  

65  

66 (*.....  

67 Exercise 2A:  

68  

69 Replace the two lines below with a single top-level 'let' expression
70 that extracts the x and y coordinate values from 'example_point_recd'
71 above into global variables 'x1' and 'y1', respectively.  

72 .....*)  

73  

74 let {x = x1; y = y1} = example_point_recd ;;  

75  

76 (*.....  

77 Exercise 2B:  

78  

79 Implement a function 'add_point_recd' to add two points of type
80 'point_recd' and returning a 'point_recd' as well.  

81 .....*)  

82  

83 (* A direct reimplementation of 'add_point_pair' would be: *)  

84  

85 let add_point_recd (p1 : point_recd) (p2 : point_recd) : point_recd =  

86   let {x = x1; y = y1}, {x = x2; y = y2} = p1, p2 in  

87     {x = x1 + x2; y = y1 + y2} ;;  

88  

89 (* By making use of dot notation for selecting record elements, this
90 version may be a bit cleaner

```

```

91
92     let add_point_recd (p1 : point_recd) (p2 : point_recd) : point_recd =
93         {x = p1.x + p2.x; y = p1.y + p2.y} ;;
94     *)
95
96 (* Recall the dot product from Lab 2. The dot product of two points
97 'x1, y1' and 'x2, y2' is the sum of the products of their x and y
98 coordinates.
99
100 .....
101 Exercise 3: Write a function `dot_product_pair` to compute the dot
102 product for points encoded as the `point_pair` type.
103 .....
104
105 let dot_product_pair (x1, y1 : point_pair) (x2, y2 : point_pair) : int =
106     x1 * x2 + y1 * y2 ;;
107
108 (* In this example, we've gone even further, performing the match
109 directly in the `let` definition of the function itself. This is
110 actually the stylistically preferred way of implementing this in
111 OCaml, as discussed in the Style Guide. Can you adjust the solution
112 to Exercises 1 and 2B to use this syntactic sugar? *)
113
114 (*.....
115 Exercise 4: Write a function `dot_product_recd` to compute the dot
116 product for points encoded as the `point_recd` type.
117 .....
118
119 let dot_product_recd (p1 : point_recd) (p2 : point_recd) : int =
120     p1.x * p2.x + p1.y * p2.y ;;
121
122 (* Converting between the pair and record representations of points
123
124 You might imagine that the two representations have
125 different advantages, such that two libraries, say, might use
126 differing types for points. In that case, we may want to have
127 functions to convert between the two representations.
128
129 .....
130 Exercise 5: Write a function `point_pair_to_recd` that converts a
131 `point_pair` to a `point_recd`.
132 .....
133
134 let point_pair_to_recd (x, y : point_pair) : point_recd =
135     {x; y} ;;
136

```

```

137 (* Note the use of pattern-matching for deconstruction directly in the
138 argument and of "field punning". Without those techniques, we'd
139 have the more cumbersome:
140
141 let point_pair_to_recd (p : point_pair) : point_recd =
142   let x, y = p in
143   {x = x; y = y} ;;
144   *)
145
146 (*.....
147 Exercise 6: Write a function `point_recd_to_pair` that converts a
148 `point_recd` to a `point_pair`.
149 .....*)
150
151 let point_recd_to_pair ({x; y} : point_recd) : point_pair =
152   x, y ;;
153
154 (*=====
155 Part 2: A simple database of records
156
157 A college wants to store student records in a simple database,
158 implemented as a list of individual course enrollments. The enrollments
159 themselves are implemented as a record type, called `enrollment`, with
160 `string` fields labeled `name` and `course` and an integer student ID
161 number labeled `id`. The appropriate type definition is: *)
162
163 type enrollment = { name : string;
164                     id : int;
165                     course : string } ;;
166
167 (* Here's an example of a list of enrollments. *)
168
169 let college =
170   [ { name = "Pat";   id = 603858772; course = "cs51" };
171     { name = "Pat";   id = 603858772; course = "expos20" };
172     { name = "Kim";   id = 482958285; course = "expos20" };
173     { name = "Kim";   id = 482958285; course = "cs20" };
174     { name = "Sandy"; id = 993855891; course = "ls1b" };
175     { name = "Pat";   id = 603858772; course = "ec10b" };
176     { name = "Sandy"; id = 993855891; course = "cs51" };
177     { name = "Sandy"; id = 482958285; course = "ec10b" }
178   ] ;;
179
180 (* In the following exercises, you'll want to avail yourself of the
181 `List` module functions, writing the requested functions in
182 higher-order style rather than handling the recursion yourself.

```

```

183
184 .....  

185 Exercise 7: Define a function called `transcript` that takes an  

186 `enrollment list` and returns a list of all the enrollments for a given  

187 student as specified by the student's ID.  

188  

189 For example:  

190  

191     # transcript college 993855891 ;;  

192     - : enrollment list =  

193     [{name = "Sandy"; id = 993855891; course = "ls1b"}];  

194     {name = "Sandy"; id = 993855891; course = "cs51"}]  

195 .....*)  

196  

197 let transcript (enrollments : enrollment list)  

198         (student : int)  

199         : enrollment list =  

200     List.filter (fun { id; _ } -> id = student) enrollments ;;  

201 (*           ^--- field punning!  

202  

203 Note the use of field punning, using the `id` variable to refer to  

204 the value of the `id` field.  

205  

206 An alternative approach is to use the dot notation to pick out the  

207 record field.  

208  

209     let transcript (enrollments : enrollment list)  

210             (student : int)  

211             : enrollment list =  

212         List.filter (fun studentrec -> studentrec.id = student)  

213             enrollments ;;  

214     *)  

215  

216 (*.....  

217 Exercise 8: Define a function called `ids` that takes an `enrollment  

218 list` and returns a list of all the ID numbers in that list,  

219 eliminating any duplicates. Hint: The `map` and `sort_uniq` functions  

220 from the `List` module and the `compare` function from the `Stdlib`  

221 module may be useful here.  

222  

223 For example:  

224  

225     # ids college ;;  

226     - : int list = [482958285; 603858772; 993855891]  

227 .....*)  

228

```

```

229 (* Making good use of the recommended library functions, we have the
230     following succinct implementation: *)
231
232 let ids (enrollments: enrollment list) : int list =
233     List.sort_uniq (compare)
234         (List.map (fun student -> student.id) enrollments) ;;
235
236 (* This time we used the alternative strategy of picking out the `id`
237     using dot notation.
238
239     By the way, the aggregation to eliminate duplicates can also be
240     done using a fold. We leave that strategy as an additional
241     exercise. *)
242
243 (* There's a big problem with this database design: nothing guarantees
244     that a given student ID is associated with a single name. The right
245     thing to do is to use a different database design where this kind of
246     thing can't happen; that would be an application of the *edict of
247     prevention*. But for the purpose of this lab, you'll just write a
248     function to verify that this problem doesn't occur.
249
250 .....
251 Exercise 9: Define a function `verify` that determines whether all the
252 entries in an enrollment list for each of the ids appearing in the
253 list have the same name associated. Hint: You may want to use
254 functions from the `List` module such as `map`, `for_all`,
255 `sort_uniq`.
256
257 For example:
258
259     # verify college ;;
260     - : bool = false
261
262 (Do you see why it's false?)
263 .....
264
265 (* We start with a function to extract all the names from the database. *)
266 let names (enrollments : enrollment list) : string list =
267     List.sort_uniq (compare)
268         (List.map (fun { name; _ } -> name) enrollments) ;;
269
270 (* Then we verify that for each id, the list of names associated with
271     the courses in that id's transcript has length 1. *)
272 let verify (enrollments : enrollment list) : bool =
273     List.for_all (fun l -> List.length l = 1)
274         (List.map

```

```

275         (fun student -> names (transcript enrollments student))
276             (ids enrollments)) ;;
277
278 (* By the way, the computed value in the example is false because
279     Sandy appears with Kim's ID number in one of the entries.) *)
280
281 (*=====
282 Part 3: Polymorphism
283
284 .....
285 Exercise 10: In Lab 2, you implemented a function `zip` that takes two
286 lists and "zips" them together into a list of pairs. Here's a possible
287 implementation of `zip`:
288
289 let rec zip (x : int list) (y : int list) : (int * int) list =
290   match x, y with
291   | [], [] -> []
292   | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
293
294 As implemented, this function works only on integer lists. Revise your
295 solution to operate polymorphically on lists of any type. What is the
296 type of the result? Did you provide full typing information in the
297 first line of the definition? (As usual, for the time being, don't
298 worry about explicitly handling the anomalous case when the two lists
299 are of different lengths.)
300 .....*)
301
302 [@@@warning "-8"]
303 let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
304   match x, y with
305   | [], [] -> []
306   | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
307
308 (* Notice how a polymorphic typing was provided in the first line, to
309   capture the intention of the polymorphic function.
310
311 You can ignore the non-exhaustive match warning, which occurs
312 because we have no match cases for when only one of the two
313 argument lists is empty. We'll have better tools to address that
314 issue later. *)
315
316 (*.....
317 Exercise 11: Partitioning a list -- Given a function returning a
318 boolean, for instance
319
320   fun x -> x mod 3 = 0

```

```

321 and a list of elements, for instance
323
324 [3; 4; 5; 10; 11; 12; 1]
325
326 we can partition the list into two lists, the list of elements
327 satisfying the boolean function `([3; 12])` and the list of elements
328 that don't `([4; 5; 10; 11; 1])`.
329
330 The library function `List.partition` partitions its list argument in
331 just this way, returning a pair of lists. Here's an example:
332
333 # List.partition (fun x -> x mod 3 = 0) [3; 4; 5; 10; 11; 12; 1] ;;
334 - : int list * int list = ([3; 12], [4; 5; 10; 11; 1])
335
336 What is the type of the `partition` function, keeping in mind that it
337 should be as polymorphic as possible?
338
339 Now implement the function yourself (without using `List.partition` of
340 course, though other `List` module functions may be useful).
341 .....*)
342
343 (* Let's start by working out the type. The `partition` function takes
344 two arguments, a boolean condition and a list of elements. The
345 boolean condition might apply to elements of any type, so it should
346 be a function of type `'a -> bool`. The list must contain elements
347 appropriate to apply the condition to, that is, elements of type
348 `'a`, so the list itself is of type `'a list`. The result is a pair
349 of lists, each of which contains elements of type `'a`, that is,
350 `'a list * 'a list`. The type of partition itself is then
351
352 ('a -> bool) -> 'a list -> 'a list * 'a list
353
354 The implementation is really straightforward if we just reuse the
355 filtering functionality of the `List.filter` function. *)
356
357 let partition (condition : 'a -> bool) (lst : 'a list)
358   : 'a list * 'a list =
359   let open List in
360   filter condition lst, filter (fun x -> not (condition x)) lst ;;
361
362 (* If, instead, we want to perform the walking of the list directly,
363 we might have
364
365 let rec partition (condition : 'a -> bool) (lst : 'a list)
366   : 'a list * 'a list =

```

```

367     match lst with
368     | [] -> [], []
369     | hd :: tl ->
370         let yeses, noes = partition condition tl in
371         if condition hd then (hd :: yeses), noes
372         else yeses, (hd :: noes) ;;
373
374     An implementation with a single fold is also possible.
375
376     let partition (condition : 'a -> bool) (lst : 'a list)
377         : 'a list * 'a list =
378         List.fold_right (fun elt (yeses, noes) ->
379             if condition elt then (elt :: yeses), noes
380             else yeses, (elt :: noes))
381         lst
382         ([] , []) ;;
383
384     To think about: Which of these do you like best? What are the
385     advantages and disadvantages of each?
386     *)
387
388 (*=====
389 Part 4: Implementing polymorphic application, currying, and uncurrying
390
391 .....
392 Exercise 12: We can think of function application itself as a
393 polymorphic higher-order function (:exploding_head:). It takes two
394 arguments -- a function and its argument -- and returns the value
395 obtained by applying the function to its argument. In this exercise,
396 you'll write this function, called `apply'. You might use it as in the
397 following examples:
398
399 # apply pred 42 ;;
400 - : int = 41
401 # apply (fun x -> x ** 2.) 3.14159 ;;
402 - : float = 9.86958772809999907
403
404     An aside: You may think such a function isn't useful, since we
405     already have an even more elegant notation for function
406     application, as in
407
408     # pred 42 ;;
409     - : int = 41
410     # (fun x -> x ** 2.) 3.14159 ;;
411     - : float = 9.86958772809999907
412

```

```

413     But we'll see a quite useful operator that works similarly --
414     the backwards application operator -- in Chapter 11 of the
415     textbook.
416
417     Start by thinking about the type of the function. We'll assume it
418     takes its two arguments curried, that is, one at a time.
419
420     1. What is the most general (polymorphic) type for its first argument
421     (the function to be applied)?
422
423     2. What is the most general type for its second argument (the argument
424     to apply it to)?
425
426     3. What is the type of its result?
427
428     4. Given the above, what should the type of the function 'apply' be?
429
430     Now write the function.
431
432     Can you think of a reason that the 'apply' function might in fact be
433     useful?
434     .....*)
```

(\* Thinking through the types of the 'apply' function:

1. Its first argument, the function to be applied, itself takes an argument of some generic type, call it ''arg''. (We're not restricted to type variables like ''a'', ''b'', ''c''. We might as well use a good mnemonic type variable name like ''arg''.) The result type for the function to be applied we'll call ''result''. So the type of the first argument is ''arg -> 'result''.
2. Its second argument is the argument to apply that function to, and must thus be of type ''arg''.
3. The type of the result of the application is, of course, ''result''.
4. So the type for apply is given by the typing:

```

453         apply : ('arg -> 'result) -> 'arg -> 'result
455
456     Types in hand, the apply function itself is truly trivial to
457     implement: *)
```

```

459 let apply (func : 'arg -> 'result) (arg : 'arg) : 'result =
460   func arg ;;
461
462 (* Something to think about: One reason the `apply` function might be
463    useful is that it might be handy as *an argument to another
464    higher-order function*. *)
465
466 (*.....*)
467 Exercise 13: In the next two exercises, you'll define polymorphic
468 higher-order functions `curry` and `uncurry` for currying and uncurrying
469 binary functions (functions of two arguments). The functions are named
470 after mathematician Haskell Curry '1920. (By way of reminder, a
471 curried function takes its arguments one at a time. An uncurried
472 function takes them all at once in a tuple.)
473
474 We start with the polymorphic higher-order function `curry`, which
475 takes as its argument an uncurried binary function and returns the
476 curried version of its argument function.
477
478 Before starting to code, pull out a sheet of paper and a pencil and
479 work out with your partner the answers to the following seven
480 questions.
481
482 ****
483      Do not skip this pencil and paper work.
484 ****
485
486 1. What is the type of the argument to the function `curry`? Write down
487    a type expression for the argument type.
488
489 2. What is an example of a function that `curry` could apply to?
490
491 3. What is the type of the result of the function `curry`? Write down a
492    type expression for the result type.
493
494 4. What should the result of applying the function `curry` to the
495    function from (2) be?
496
497 5. Given (1) and (2), write down a type expression for the type of the
498    `curry` function itself.
499
500 6. What would a good variable name for the argument to `curry` be?
501
502 7. Write down the header line for the definition of the `curry` function.
503
504 Call over a staff member to go over your answers to these

```

```

505 questions. Once you fully understand all this, its time to implement
506 the function `curry`.
507 .....*)
508
509 (* In order to think through this problem, it helps to start with the
510 types of the functions. The `curry` function is a *function*; it has
511 a function type, of the form  $_ \rightarrow _$ . It is intended to take an
512 uncurried binary function as its argument, and return the
513 corresponding curried function. An uncurried binary function is a
514 function that takes its two arguments both "at the same time", that
515 is, as a pair. Generically, the type of such a function is thus
516
517     ``a * 'b -> 'c''      (that's the answer to question (1) above)
518
519 An example (2) would be the function that adds the elements of an
520 'int' pair:
521
522     `fun (x, y) -> x + y`
523
524 A curried binary function takes its two arguments "one at a time".
525 Its type is
526
527     ``a -> ('b -> 'c)'
528
529 which is the appropriate result type for the curry function (3). For
530 instance, the curried version of the integer addition function is
531 just the `(+)` operator itself (4).
532
533 Putting these together, the type of curry should be (5)
534
535     ((`a * 'b) -> 'c) -> ('a -> ('b -> 'c)) .
536
537 Dropping extraneous parentheses since the `->` type operator is right
538 associative (and of lower precedence than `*`, we can also write this
539 as
540
541     ('a * 'b -> 'c) -> 'a -> 'b -> 'c .
542
543 A good name for the argument of the curry function is 'uncurried'
544 (6), to emphasize that it is an uncurried function.
545
546 This type information already gives us a big hint as to how to
547 write the curry function. We start with the first line giving the
548 argument structure (7):
549
550     let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c = ...

```

```

551
552     The return type is a function type, so we'll want to build a
553     function value to return. We use the `fun _ -> _` anonymous
554     function construction to do so, carefully labeling the type of the
555     function's argument as a reminder of what's going on:
556
557         let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
558             fun (x : 'a) -> ...
559
560     The type of the argument of this anonymous function is ``a`` because
561     its type as a whole -- the return type of `curry` itself -- is ``a
562     -> ('b -> 'c)``. This function should return a function of type ``b
563     -> 'c``. We'll construct that as an anonymous function as well:
564
565         let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
566             fun (x : 'a) ->
567                 fun (y : 'b) -> ...
568
569     Now, how should we construct the value (of type ``c``) that this
570     inner function should return? Remember that `curry` should return a
571     curried function whose value is the same as the uncurried function
572     would have delivered on arguments `x` and `y`. So we can simply
573     apply `uncurried` to `x` and `y` (in an uncurried fashion, of
574     course), to obtain the value of type ``c``:
575
576         let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
577             fun (x : 'a) ->
578                 fun (y : 'b) -> uncurried (x, y) ;;
579
580     You'll note that all of these anonymous functions are a bit
581     cumbersome, and we have a nicer notation for defining functions in
582     let expressions incorporating the arguments in the definition part
583     itself. We've already done so for the argument `uncurried`. Let's use
584     that notation for the `x` and `y` arguments as well.
585
586         let curry (uncurried : 'a * 'b -> 'c) (x : 'a) (y : 'b) : 'c =
587             uncurried (x, y) ;;
588
589     To make clearer what's going on, we can even drop the explicit
590     types to show the structure of the computation:
591
592         let curry uncurried x y = uncurried (x, y) ;;
593
594     Here, we see what's really going on: `curry uncurried` when applied
595     to `x` and `y` in curried fashion gives the same value that
596     `uncurried` gives when applied to `x` and `y` in uncurried fashion.

```

```

597     By a similar argument (which it might be useful to carry out
598     yourself), uncurry is implemented as
599
600         let uncurry curried (x, y) = curried x y ;;
601
602     Below, we use the version with explicit types, as we generally want
603     to do to make our typing intentions known to the
604     compiler/interpreter. *)
605
606
607 let curry (uncurried : 'a * 'b -> 'c) (x : 'a) (y : 'b) : 'c =
608     uncurried (x, y) ;;
609
610 (*.....
611 Exercise 14: Now implement the polymorphic higher-order function
612 `uncurry`, which takes as its argument a curried function and returns
613 the uncurried version of its argument function. You may want to go
614 through the same 7-step process to get started.
615 .....*)
616
617 let uncurry (curried : 'a -> 'b -> 'c) (x, y : 'a * 'b) : 'c =
618     curried x y ;;
619
620 (*.....
621 Exercise 15: OCaml's built in binary operators, like `+` and `*` are
622 curried. You can tell from their types:
623
624     # ( + ) ;;
625     - : int -> int -> int = <fun>
626     # ( * ) ;;
627     - : int -> int -> int = <fun>
628
629 Using your `uncurry` function, define uncurried versions of the `+` and
630 `*` functions. Call them `plus` and `times`.
631 .....*)
632
633 let plus = uncurry ( + ) ;;
634
635 let times = uncurry ( * ) ;;
636
637 (* Did you write something like this?
638
639     let plus x y =
640         ...more stuff here...
641
642 Remember, functions are first-class values in OCaml; they can be

```

