```
(*
                            CS51 Lab 3
                   Polymorphism and record types
 *)


(*======================================================================
Readings:

    This lab builds on material from Chapters 7.4 and 9-9.5 of the
    textbook <http://book.cs51.io>, which should be read before the lab
    session.

Objective:

    In this lab, you'll exercise your understanding of polymorphism and
    record types. Some of the problems extend those from Lab 2, but we'll
    provide the necessary background code from that lab.
======================================================================*)

(*======================================================================
Part 1: Records and tuples

Records and tuples provide two different ways to package together
data. They differ in whether their components are selected by *name*
or by *position*, respectively.

Consider a point in Cartesian (x-y) coordinates. A point is specified
by its x and y values, which we'll take to be ints. We can package
these together as a pair (a 2-tuple), as in the following data type
definition: *)

type point_pair = int * int ;;

(* Then, we can add two points (summing their x and y coordinates
separately) with the following function:

    let add_point_pair (p1 : point_pair) (p2 : point_pair) : point_pair =
      let x1, y1 = p1 in
      let x2, y2 = p2 in
      (x1 + x2, y1 + y2) ;;


.......................................................................
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   ...................................................................*)
```

```ocaml
let dot_product_pair (p1 : point_pair) (p2 : point_pair) : int =
  failwith "dot_product_pair not implemented" ;;

(*...............................................................
Exercise 4: Write a function `dot_product_recd` to compute the dot
product for points encoded as the `point_recd` type.
..............................................................*)

let dot_product_recd (p1 : point_recd) (p2 : point_recd) : int =
  failwith "dot_product_recd not implemented" ;;

(* Converting between the pair and record representations of points

You might imagine that the two representations have
different advantages, such that two libraries, say, might use
differing types for points. In that case, we may want to have
functions to convert between the two representations.

...............................................................
Exercise 5: Write a function `point_pair_to_recd` that converts a
`point_pair` to a `point_recd`.
..............................................................*)

let point_pair_to_recd =
  fun _ -> failwith "point_pair_to_recd not implemented" ;;

(*...............................................................
Exercise 6: Write a function `point_recd_to_pair` that converts a
`point_recd` to a `point_pair`.
..............................................................*)

let point_recd_to_pair =
  fun _ -> failwith "point_recd_to_pair not implemented" ;;

(*====================================================================
Part 2: A simple database of records

A college wants to store student records in a simple database,
implemented as a list of individual course enrollments. The enrollments
themselves are implemented as a record type, called `enrollment`, with
`string` fields labeled `name` and `course` and an integer student ID
number labeled `id`. The appropriate type definition is: *)

type enrollment = { name : string;
                    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 function returning a
241   boolean, for instance
242
243       fun x -> x mod 3 = 0
244
245   and a list of elements, for instance
246
247       [3; 4; 5; 10; 11; 12; 1]
248
249   we can partition the list into two lists, the list of elements
250   satisfying the boolean function (`[3; 12]`) and the list of elements
251   that don't (`[4; 5; 10; 11; 1]`).
252
253   The library function `List.partition` partitions its list argument in
254   just this way, returning a pair of lists. Here's an example:
255
256       # List.partition (fun x -> x mod 3 = 0) [3; 4; 5; 10; 11; 12; 1] ;;
257       - : int list * int list = ([3; 12], [4; 5; 10; 11; 1])
258
259   What is the type of the `partition` function, keeping in mind that it
260   should be as polymorphic as possible?
261
262   Now implement the function yourself (without using `List.partition` of
263   course, though other `List` module functions may be useful).
264   ...................................................................*)
265
266   let partition =
267     fun _ -> failwith "partition not implemented" ;;
268
269   (*==================================================================
270   Part 4: Implementing polymorphic application, currying, and uncurrying
271
272   ...................................................................
273   Exercise 12: We can think of function application itself as a
274   polymorphic higher-order function (:exploding_head:). It takes two
```

6

arguments -- a function and its argument -- and returns the value
obtained by applying the function to its argument. In this exercise,
you'll write this function, called `apply`. You might use it as in the
following examples:

```
# apply pred 42 ;;
- : int = 41
# apply (fun x -> x ** 2.) 3.14159 ;;
- : float = 9.86958772809999907
# apply int_of_float 3.14159 ;;
- : int = 3
```

An aside: You may think such a function isn't useful, since we
already have an even more elegant notation for function
application, as in

```
# pred 42 ;;
- : int = 41
# (fun x -> x ** 2.) 3.14159 ;;
- : float = 9.86958772809999907
# int_of_float 3.14159 ;;
- : int = 3
```

But we'll see a quite useful operator that works similarly --
the backwards application operator -- in Chapter 11 of the
textbook.

Start by thinking about the type of the function. We'll assume it
takes its two arguments curried, that is, one at a time.

1. What is the most general (polymorphic) type for its first argument
   (the function to be applied)?

2. What is the most general type for its second argument (the argument
   to apply it to)?

3. What is the type of its result?

4. Given the above, what should the type of the function `apply` be?

Now write the function.

Can you think of a reason that the `apply` function might in fact be
useful?
....*)

```ocaml
let apply =
  fun _ -> failwith "apply not implemented" ;;
```

```
(*...............................................................
Exercise 13: In the next two exercises, you'll define polymorphic
higher-order functions `curry` and `uncurry` for currying and uncurrying
binary functions (functions of two arguments). The functions are named
after mathematician Haskell Curry '1920. (By way of reminder, a
curried function takes its arguments one at a time. An uncurried
function takes them all at once in a tuple.)

We start with the polymorphic higher-order function `curry`, which
takes as its argument an uncurried binary function and returns the
curried version of its argument function.

Before starting to code, pull out a sheet of paper and a pencil and
work out with your partner the answers to the following seven
questions.

        ************************************************************
                   Do not skip this pencil and paper work.
        ************************************************************

1. What is the type of the argument to the function `curry`? Write down
   a type expression for the argument type.

2. What is an example of a function that `curry` could apply to?

3. What is the type of the result of the function `curry`? Write down a
   type expression for the result type.

4. What should the result of applying the function `curry` to the
   function from (2) be?

5. Given (1) and (2), write down a type expression for the type of the
   `curry` function itself.

6. What would a good variable name for the argument to `curry` be?

7. Write down the header line for the definition of the `curry` function.

Call over a staff member to go over your answers to these
questions. Once you fully understand all this, its time to implement
the function `curry`.
.................................................................*)
```

8

```
367    let curry = fun _ -> failwith "curry not implemented" ;;
368
369    (*.........................................................................
370    Exercise 14: Now implement the polymorphic higher-order function
371    `uncurry`, which takes as its argument a curried function and returns
372    the uncurried version of its argument function.  You may want to go
373    through the same 7-step process to get started.
374    .........................................................................*)
375
376    let uncurry = fun _ -> failwith "uncurry not implemented" ;;
377
378    (*.........................................................................
379    Exercise 15: OCaml's built in binary operators, like `+` and `*` are
380    curried. You can tell from their types:
381
382        # ( + ) ;;
383        - : int -> int -> int = <fun>
384        # ( * ) ;;
385        - : int -> int -> int = <fun>
386
387    Using your `uncurry` function, define uncurried versions of the `+` and
388    `*` functions. Call them `plus` and `times`.
389    .........................................................................*)
390
391    let plus =
392      fun _ -> failwith "plus not implemented"
393
394    let times =
395      fun _ -> failwith "times not implemented" ;;
396
397    (*.........................................................................
398    Exercise 16: Recall the `prods` function from Lab 1:
399
400        let rec prods (lst : (int * int) list) : int list =
401          match lst with
402          | [] -> []
403          | (x, y) :: tail -> (x * y) :: (prods tail) ;;
404
405    Now reimplement `prods` using `map` and your uncurried `times`
406    function. Why do you need the uncurried `times` function?
407    .........................................................................*)
408
409    let prods =
410      fun _ -> failwith "prods not implemented" ;;
```