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

(*
                               SOLUTION
 *)

(*======================================================================
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) ;;
```

```ocaml

......................................................................
Exercise 1:

It might be nicer to deconstruct the points in a single match, rather
than the two separate matches in the two `let` expressions. Reimplement
`add_point_pair` to use a single pattern match in a single `let`
expression.
....................................................................*)

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

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

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

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

(*....................................................................
Exercise 2A:

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

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

(*....................................................................
Exercise 2B:

Implement a function `add_point_recd` to add two points of type
`point_recd` and returning a `point_recd` as well.
....................................................................*)

(* A direct reimplementation of `add_point_pair` would be: *)

let add_point_recd (p1 : point_recd) (p2 : point_recd) : point_recd =
  let {x = x1; y = y1}, {x = x2; y = y2} = p1, p2 in
  {x = x1 + x2; y = y1 + y2} ;;

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

```
   let add_point_recd (p1 : point_recd) (p2 : point_recd) : point_recd =
     {x = p1.x + p2.x; y = p1.y + p2.y} ;;
 *)

(* Recall the dot product from Lab 2. The dot product of two points
`x1, y1` and `x2, y2` is the sum of the products of their x and y
coordinates.

.......................................................................
Exercise 3: Write a function `dot_product_pair` to compute the dot
product for points encoded as the `point_pair` type.
.....................................................................*)

let dot_product_pair (x1, y1 : point_pair) (x2, y2 : point_pair) : int =
  x1 * x2 + y1 * y2 ;;

  (* In this example, we've gone even further, performing the match
   directly in the `let` definition of the function itself. This is
   actually the stylistically preferred way of implementing this in
   OCaml, as discussed in the Style Guide. Can you adjust the solution
   to Exercises 1 and 2B to use this syntactic sugar? *)

(*.....................................................................
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 =
  p1.x * p2.x + p1.y * p2.y ;;

(* 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 (x, y : point_pair) : point_recd =
  {x; y} ;;
```

```
(* Note the use of pattern-matching for deconstruction directly in the
   argument and of "field punning". Without those techniques, we'd
   have the more cumbersome:

let point_pair_to_recd (p : point_pair) : point_recd =
  let x, y = p in
  {x = x; y = y} ;;
   *)

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

let point_recd_to_pair ({x; y} : point_recd) : point_pair =
  x, y ;;

(*======================================================================
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;
                    course : string } ;;

(* Here's an example of a list of enrollments. *)

let college =
  [ { name = "Pat";   id = 603858772; course = "cs51" };
    { name = "Pat";   id = 603858772; course = "expos20" };
    { name = "Kim";   id = 482958285; course = "expos20" };
    { name = "Kim";   id = 482958285; course = "cs20" };
    { name = "Sandy"; id = 993855891; course = "ls1b" };
    { name = "Pat";   id = 603858772; course = "ec10b" };
    { name = "Sandy"; id = 993855891; course = "cs51" };
    { name = "Sandy"; id = 482958285; course = "ec10b" }
  ] ;;

(* In the following exercises, you'll want to avail yourself of the
`List` module functions, writing the requested functions in
higher-order style rather than handling the recursion yourself.
```

```

......................................................................
Exercise 7: Define a function called `transcript` that takes an
`enrollment list` and returns a list of all the enrollments for a given
student as specified by the student's ID.

For example:

    # transcript college 993855891 ;;
    - : enrollment list =
    [{name = "Sandy"; id = 993855891; course = "ls1b"};
     {name = "Sandy"; id = 993855891; course = "cs51"}]
....................................................................*)

let transcript (enrollments : enrollment list)
               (student : int)
             : enrollment list =
  List.filter (fun { id; _ } -> id = student) enrollments ;;
(*                  ^^--- field punning!

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

An alternative approach is to use the dot notation to pick out the
record field.

    let transcript (enrollments : enrollment list)
                   (student : int)
                 : enrollment list =
      List.filter (fun studentrec -> studentrec.id = student)
                  enrollments ;;
 *)

(*....................................................................
Exercise 8: Define a function called `ids` that takes an `enrollment
list` and returns a list of all the ID numbers in that list,
eliminating any duplicates. Hint: The `map` and `sort_uniq` functions
from the `List` module and the `compare` function from the `Stdlib`
module may be useful here.

For example:

    # ids college ;;
    - : int list = [482958285; 603858772; 993855891]
....................................................................*)

```

```
(* Making good use of the recommended library functions, we have the
   following succinct implementation: *)

let ids (enrollments: enrollment list) : int list =
  List.sort_uniq (compare)
                 (List.map (fun student -> student.id) enrollments) ;;

(* This time we used the alternative strategy of picking out the `id`
   using dot notation.

   By the way, the aggregation to eliminate duplicates can also be
   done using a fold. We leave that strategy as an additional
   exercise. *)

(* There's a big problem with this database design: nothing guarantees
that a given student ID is associated with a single name. The right
thing to do is to use a different database design where this kind of
thing can't happen; that would be an application of the *edict of
prevention*. But for the purpose of this lab, you'll just write a
function to verify that this problem doesn't occur.

......................................................................
Exercise 9: Define a function `verify` that determines whether all the
entries in an enrollment list for each of the ids appearing in the
list have the same name associated. Hint: You may want to use
functions from the `List` module such as `map`, `for_all`,
`sort_uniq`.

For example:

    # verify college ;;
    - : bool = false

(Do you see why it's false?)
....................................................................*)

(* We start with a function to extract all the names from the database. *)
let names (enrollments : enrollment list) : string list =
  List.sort_uniq (compare)
                 (List.map (fun { name; _ } -> name) enrollments) ;;

(* Then we verify that for each id, the list of names associated with
   the courses in that id's transcript has length 1. *)
let verify (enrollments : enrollment list) : bool =
  List.for_all (fun l -> List.length l = 1)
               (List.map
```

```
                    (fun student -> names (transcript enrollments student))
                    (ids enrollments)) ;;

(* By the way, the computed value in the example is false because
   Sandy appears with Kim's ID number in one of the entries.) *)

(*======================================================================
Part 3: Polymorphism

........................................................................
Exercise 10: In Lab 2, you implemented a function `zip` that takes two
lists and "zips" them together into a list of pairs. Here's a possible
implementation of `zip`:

  let rec zip (x : int list) (y : int list) : (int * int) list =
    match x, y with
    | [], [] -> []
    | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;

As implemented, this function works only on integer lists. Revise your
solution to operate polymorphically on lists of any type. What is the
type of the result? Did you provide full typing information in the
first line of the definition? (As usual, for the time being, don't
worry about explicitly handling the anomalous case when the two lists
are of different lengths.)
......................................................................*)

[@@@warning "-8"]
let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
  match x, y with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;

(* Notice how a polymorphic typing was provided in the first line, to
   capture the intention of the polymorphic function.

   You can ignore the non-exhaustive match warning, which occurs
   because we have no match cases for when only one of the two
   argument lists is empty. We'll have better tools to address that
   issue later. *)

(*......................................................................
Exercise 11: Partitioning a list -- Given a function returning a
boolean, for instance

    fun x -> x mod 3 = 0
```

```
and a list of elements, for instance

    [3; 4; 5; 10; 11; 12; 1]

we can partition the list into two lists, the list of elements
satisfying the boolean function (`[3; 12]`) and the list of elements
that don't (`[4; 5; 10; 11; 1]`).

The library function `List.partition` partitions its list argument in
just this way, returning a pair of lists. Here's an example:

    # List.partition (fun x -> x mod 3 = 0) [3; 4; 5; 10; 11; 12; 1] ;;
    - : int list * int list = ([3; 12], [4; 5; 10; 11; 1])

What is the type of the `partition` function, keeping in mind that it
should be as polymorphic as possible?

Now implement the function yourself (without using `List.partition` of
course, though other `List` module functions may be useful).
......................................................................*)

(* Let's start by working out the type. The `partition` function takes
   two arguments, a boolen condition and a list of elements. The
   boolean condition might apply to elements of any type, so it should
   be a function of type `'a -> bool`. The list must contain elements
   appropriate to apply the condition to, that is, elements of type
   `'a`, so the list itself is of type `'a list`. The result is a pair
   of lists, each of which contains elements of type `'a`, that is,
   `'a list * 'a list`. The type of partition itself is then

      ('a -> bool) -> 'a list -> 'a list * 'a list

   The implementation is really straightforward if we just reuse the
   filtering functionality of the `List.filter` function.  *)

let partition (condition : 'a -> bool) (lst : 'a list)
            : 'a list * 'a list =
  let open List in
  filter condition lst, filter (fun x -> not (condition x)) lst ;;

(* If, instead, we want to perform the walking of the list directly,
   we might have

    let rec partition (condition : 'a -> bool) (lst : 'a list)
                    : 'a list * 'a list =
```

```
      match lst with
      | [] -> [], []
      | hd :: tl ->
          let yeses, noes = partition condition tl in
          if condition hd then (hd :: yeses), noes
          else yeses, (hd :: noes) ;;

   An implementation with a single fold is also possible.

    let partition (condition : 'a -> bool) (lst : 'a list)
                : 'a list * 'a list =
      List.fold_right (fun elt (yeses, noes) ->
                         if condition elt then (elt :: yeses), noes
                         else yeses, (elt :: noes))
                      lst
                      ([], []) ;;

   To think about: Which of these do you like best? What are the
   advantages and disadvantages of each?
 *)

(*======================================================================
Part 4: Implementing polymorphic application, currying, and uncurrying

......................................................................
Exercise 12: We can think of function application itself as a
polymorphic higher-order function (:exploding_head:). It takes two
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

        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

```

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

(* 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:

         apply : ('arg -> 'result) -> 'arg -> 'result

   Types in hand, the apply function itself is truly trivial to
   implement: *)
```

```
let apply (func : 'arg -> 'result) (arg : 'arg) : 'result =
  func arg ;;

(* Something to think about: One reason the `apply` function might be
   useful is that it might be handy as *an argument to another
   higher-order function*. *)

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

(* In order to think through this problem, it helps to start with the
   types of the functions. The `curry` function is a *function*; it has
   a function type, of the form _ -> _. It is intended to take an
   uncurried binary function as its argument, and return the
   corresponding curried function. An uncurried binary function is a
   function that takes its two arguments both "at the same time", that
   is, as a pair. Generically, the type of such a function is thus

           `'a * 'b -> 'c`    (that's the answer to question (1) above)

   An example (2) would be the function that adds the elements of an
   `int` pair:

           `fun (x, y) -> x + y`

   A curried binary function takes its two arguments "one at a time".
   Its type is

           `'a -> ('b -> 'c)`

   which is the appropriate result type for the curry function (3). For
   instance, the curried version of the integer addition function is
   just the `(+)` operator itself (4).

   Putting these together, the type of curry should be (5)

        (('a * 'b) -> 'c) -> ('a -> ('b -> 'c))      .

   Dropping extraneous parentheses since the `->` type operator is right
   associative (and of lower precedence than `*`, we can also write this
   as

        ('a * 'b -> 'c) -> 'a -> 'b -> 'c      .

   A good name for the argument of the curry function is `uncurried`
   (6), to emphasize that it is an uncurried function.

   This type information already gives us a big hint as to how to
   write the curry function. We start with the first line giving the
   argument structure (7):

        let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c = ...
```

The return type is a function type, so we'll want to build a function value to return. We use the `fun _ -> _` anonymous function construction to do so, carefully labeling the type of the function's argument as a reminder of what's going on:

```
let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
  fun (x : 'a) -> ...
```

The type of the argument of this anonymous function is `'a` because its type as a whole -- the return type of `curry` itself -- is `'a -> ('b -> 'c)`. This function should return a function of type `'b -> 'c`. We'll construct that as an anonymous function as well:

```
let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
  fun (x : 'a) ->
    fun (y : 'b) -> ...
```

Now, how should we construct the value (of type `'c`) that this inner function should return? Remember that curry should return a curried function whose value is the same as the uncurried function would have delivered on arguments `x` and `y`. So we can simply apply `uncurried` to `x` and `y` (in an uncurried fashion, of course), to obtain the value of type `'c`:

```
let curry (uncurried : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
  fun (x : 'a) ->
    fun (y : 'b) -> uncurried (x, y) ;;
```

You'll note that all of these anonymous functions are a bit cumbersome, and we have a nicer notation for defining functions in let expressions incorporating the arguments in the definition part itself. We've already done so for the argument uncurried. Let's use that notation for the `x` and `y` arguments as well.

```
let curry (uncurried : 'a * 'b -> 'c) (x : 'a) (y : 'b) : 'c =
  uncurried (x, y) ;;
```

To make clearer what's going on, we can even drop the explicit types to show the structure of the computation:

```
let curry uncurried x y = uncurried (x, y) ;;
```

Here, we see what's really going on: `curry uncurried` when applied to `x` and `y` in curried fashion gives the same value that `uncurried` gives when applied to `x` and `y` in uncurried fashion.

```
   By a similar argument (which it might be useful to carry out
   yourself), uncurry is implemented as

       let uncurry curried (x, y) = curried x y ;;

   Below, we use the version with explicit types, as we generally want
   to do to make our typing intentions known to the
   compiler/interpreter. *)

let curry (uncurried : 'a * 'b -> 'c) (x : 'a) (y : 'b) : 'c =
  uncurried (x, y) ;;

(*......................................................................
Exercise 14: Now implement the polymorphic higher-order function
`uncurry`, which takes as its argument a curried function and returns
the uncurried version of its argument function.  You may want to go
through the same 7-step process to get started.
......................................................................*)

let uncurry (curried : 'a -> 'b -> 'c) (x, y : 'a * 'b) : 'c =
  curried x y ;;

(*......................................................................
Exercise 15: OCaml's built in binary operators, like `+` and `*` are
curried. You can tell from their types:

    # ( + ) ;;
    - : int -> int -> int = <fun>
    # ( * ) ;;
    - : int -> int -> int = <fun>

Using your `uncurry` function, define uncurried versions of the `+` and
`*` functions. Call them `plus` and `times`.
......................................................................*)

let plus = uncurry ( + ) ;;

let times = uncurry ( * ) ;;

(* Did you write something like this?

    let plus x y =
      ...more stuff here...

   Remember, functions are first-class values in OCaml; they can be
```

```
   returned by other functions. So you don't always need to give the
   arguments explicitly in a function definition.  *)

(*......................................................................
Exercise 16: Recall the `prods` function from Lab 1:

    let rec prods (lst : (int * int) list) : int list =
      match lst with
      | [] -> []
      | (x, y) :: tail -> (x * y) :: (prods tail) ;;

Now reimplement `prods` using `map` and your uncurried `times`
function. Why do you need the uncurried `times` function?
......................................................................*)

let prods = List.map times ;;

(* Elegant, no? *)
```