```ocaml
(*
                              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:
```

```

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 =
  failwith "add_point_pair not implemented" ;;

(* 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 x1 = 0 ;;
let y1 = 0 ;;

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

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

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

(* 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 (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;
```

```
                    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 =
  failwith "transcript not implemented" ;;

(*......................................................................
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]
..................................................................*)

let ids (enrollments: enrollment list) : int list =
  failwith "ids not implemented" ;;

(* 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?)
..................................................................*)

let verify (enrollments : enrollment list) : bool =
  failwith "verify not implemented" ;;

(*==================================================================
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.)
.......................................................................*)

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

(*......................................................................
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 partition =
  fun _ -> failwith "partition not implemented" ;;

(*======================================================================
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?
......................................................................*)

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

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

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

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

(*......................................................................
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 = fun _ -> failwith "uncurry not implemented" ;;

(*......................................................................
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 =
  fun _ -> failwith "plus not implemented"

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

(*......................................................................
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 =
  fun _ -> failwith "prods not implemented" ;;
```