```
(*
                              CS51 Lab 2
                      More Functional Programming:
            Simple Data Structures and Higher-Order Functions
 *)
(*
                                SOLUTION
 *)
(*=====================================================================
Readings:

    This lab builds on material from Chapters 7-8 of the textbook
    <http://book.cs51.io>, which should be read before the lab session.

Objective:

    This lab is intended to introduce you to staples of functional
    programming in OCaml, including:

        * simple data structures like lists and tuples
        * higher-order functional programming (functions as first-class
          values)
=====================================================================*)

(*=====================================================================
Part 1: Types and type inference beyond atomic types

Exercise 1: What are appropriate types to replace the ??? in the
expressions below? For each expression, replace the ??? with the
type. Test your solution by uncommenting the examples (removing the
`(*` and `*)` lines at start and end) and verifying that no typing
error is generated.
.....................................................................*)

let exercise1a : float * string =
  (0.1, "hi") ;;

let exercise1b : string list =
  let add_to_hello_list x = ["Hello"; x]
  in add_to_hello_list "World!";;

let exercise1c : int * float -> int  =
  fun (x, y) -> x + int_of_float y ;;

```

```
let exercise1d : int list -> bool =
  fun lst ->
    match lst with
    | [] -> false
    | hd :: _ -> hd < hd + 1 ;;

(* Here we deconstruct our argument as a list, so the argument `lst`
   must be of of type `___ list` (consistent with its name). In the
   second match case, we compare the head of the list with itself plus
   one. We use integer addition, so the head must be of type
   `int`. All elements of a list must be of the same type, so the
   argument is of type `int list`. To determine the type of the
   result, we can look to the first match case. The literal `false` is
   of type `bool`. All match cases must return the same type, so the
   result must be of type `bool`. *)

let exercise1e : bool -> bool list =
  fun x -> if x then [x] else [] ;;

(* The reasoning goes like this: The argument of the function is
   `x`. Since `x` is used as the condition part of an `if` expression,
   it must be of type bool. Thus the expression in the `then` clause
   `[x]` must be a `bool list`, and since the type of an `if` expression
   is the type of its `then` and `else` expressions, the whole `if`
   expression, the result of the function, must be a `bool list`. The
   function itself is then of function type `bool -> bool list`, as is
   the value named `exercise1e`. *)

(*.....................................................................
Exercise 2: Update each expression below by changing the 0 in the last
line to an integer literal so that the expression as a whole evaluates
to `true`.
.....................................................................*)

let exercise2a =
  let lst = [1; 2; 3; 4] in
  let value =
    match lst with
    | [] -> 0
    | [h] -> h
    | h1 :: h2 :: t -> h2 in
  value = 2 ;;

let exercise2b =
  let x, y, z = 4, [1; 3], true in
  let value =
```

```
    match y with
    | [] -> 0
    | h :: t -> h in
  value = 1 ;;

let exercise2c =
  let tuple_lst = [(1, 4); (5, 2)] in
  let value =
    match tuple_lst with
    | [] -> 0
    | (a, b) :: t -> a
    | h1 :: (a, b) :: t -> a in
  value = 1 ;;

let exercise2d =
  let tuple_lst = [(1, 4); (5, 2)] in
  let value =
    match tuple_lst with
    | [] -> 0
    | h1 :: (a, b) :: t -> a
    | (a, b) :: t -> a in
  value = 5 ;;

(*..................................................................
Exercise 3: Complete the following definition for a function
`third_element` that returns a `bool * int` pair, whose first element
represents whether or not its list argument has a third element, and
whose second element represents that element if it exists (or 0 if it
does not). Here are some examples of the intended behavior of this
function:

    # third_element [1; 2; 3; 4; 5] ;;
    - : bool * int = (true, 3)
    # third_element [] ;;
    - : bool * int = (false, 0)
..................................................................*)

let third_element (lst : int list) : bool * int =
  match lst with
  | _ :: _ :: elt3 :: _ -> true, elt3
  | _ -> false, 0 ;;

(* Because we never use the first two elements or the tail of the
   list, naming the variables is unnecessary, so we can use anonymous
   variables (_) as shown above.
```

```
   The order of the two match cases is crucial. If they are flipped,
   the second match case will never be invoked, since the first
   match case will always match. *)

(*======================================================================
Part 2: First-order functional programming with lists

We'll continue with some "finger exercises" defining simple functions
before moving on to more complex problems. The intention in this part
of the lab is for you to implement these functions by *explicit
recursion*. Only later, in part 3 of this lab, will we make use of the
`map`/`fold`/`filter` higher-order functions.

As a reminder, here's the definition for the `length` function of type
`int list -> int` implemented in this explicit recursion style:

    let rec length (lst : int list) : int =
      match lst with
      | [] -> 0
      | _head :: tail -> 1 + length tail ;;

......................................................................
Exercise 4: In lab 1, we defined a function that could square its
input. Now, define a function `square_all` that squares all of the
elements of an integer list. We've provided a bit of template code,
supplying the first line of the function definition, but the body of
the template code just causes a failure by forcing an error using the
built-in `failwith` function. Edit the code to implement `square_all`
properly.

Test out your implementation of `square_all` by modifying the template
code below to define `exercise4` to be the `square_all` function
applied to the list containing the elements `3`, `4`, and `5`. You'll
want to replace the `[]` with the correct function application.

Thorough testing is important in all your work, and we hope to impart
this view to you in CS51. Testing will help you find bugs, avoid
mistakes, and teach you the value of short, clear functions. In the
file `lab2_tests.ml`, we've put some prewritten tests for `square_all`
using the testing method of Section 6.7 in the book. Spend some time
understanding how the testing function works and why these tests are
comprehensive. Then test your code by compiling and running the test
suite:

    % ocamlbuild -use-ocamlfind lab2_tests.byte
    % ./lab2_tests.byte
```

```
183
184 You may want to add some tests for other functions in the lab to get
185 some practice with automated unit testing.
186 ..........................................................................*)
187
188 let rec square_all (lst : int list) : int list =
189   match lst with
190   | [] -> []
191   | head :: tail -> (head * head) :: (square_all tail) ;;
192
193 let exercise4 =
194   square_all [3; 4; 5] ;;
195
196 (*..........................................................................
197 Exercise 5: Define a recursive function `sum` that sums the values in
198 its integer list argument. (What's a sensible return value for the sum
199 of the empty list?)
200 ..........................................................................*)
201
202 let rec sum (lst : int list) : int =
203   match lst with
204   | [] -> 0
205   | head :: tail -> head + sum tail ;;
206
207 (*..........................................................................
208 Exercise 6: Define a recursive function `max_list` that returns the
209 maximum element in a non-empty integer list. Don't worry about what
210 happens on an empty list. You may be warned by the compiler that "this
211 pattern-matching is not exhaustive." You may ignore this warning for
212 this lab.
213 ..........................................................................*)
214
215 (* Here's a first cut at a solution, using just the portion of OCaml
216    already introduced. Notice that there's no branch in the pattern
217    match that matches the empty list, because there is no maximum
218    element in the empty list! For that reason, the ocaml interpreter
219    warns us with an "inexhaustive pattern match" warning.
220
221       let rec max_list (lst : int list) : int =
222         match lst with
223         | [elt] -> elt
224         | head :: tail ->
225            let max_tail = max_list tail in
226            if head > max_tail then head else max_tail ;;
227
228    This is the solution we expected people to come up with. And it
```

```
   seems to work.

      # max_list [1; 3; 2] ;;
      - : int = 3

   What happens when we apply this function to the empty list?

      # max_list [] ;;
      Exception: Match_failure ("//toplevel//", 2, 2).

   It generates a `Match_failure` exception. (We'll talk more about
   error handling and exceptions later in the course, and use them
   starting in Lab 4.) This `Match_failure` exception is a symptom of
   a deeper underlying problem, namely, that the function `max_list`
   was called with an invalid argument. A better solution, then, and
   one that not coincidentally eliminates the "inexhaustive pattern
   match" warning, is to explicitly raise a more appropriate exception
   like `Invalid_argument`, as we've done in the solution below. *)

let rec max_list (lst : int list) : int =
  match lst with
  | [] -> raise (Invalid_argument "max_list: empty list")
  | [elt] -> elt
  | head :: tail ->
     let max_tail = max_list tail in
     if head > max_tail then head else max_tail ;;

(* It turns out that the `Stdlib` module has a `max` function that
   returns the larger of its two arguments. Using that function, we
   can simplify a bit.

    let rec max_list (lst : int list) : int =
      match lst with
      | [elt] -> elt
      | head :: tail -> max head (max_list tail) ;;
 *)

(*.....................................................................
Exercise 7: Define a function `zip`, that takes two `int list`
arguments and returns a list of pairs of ints, one from each of the
two argument lists. Your function can assume the input lists will be
the same length. You can ignore what happens in the case the input
list lengths do not match. You may be warned by the compiler that
"this pattern-matching is not exhaustive." You may ignore this warning
for this lab.
```

```
For example,

  fun lst -> zip [1; 2; 3] [4; 5; 6] ;;
  - : (int * int) list = [(1, 4); (2, 5); (3, 6)]

To think about: Why wouldn't it be possible, in cases of mismatched
length lists, to just pad the shorter list with, say, `false` values, so
that, `zip [1] [2; 3; 4] = [(1, 2); (false, 3); (false, 4)]`?
.....................................................................*)

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) ;;

(* This was the solution we expected people to come up with. It
   generates a warning about the pattern match not being
   exhaustive. As in `max_list` above, the ramifications of this issue
   and how best to address it are discussed at length in Chapter 10,
   Section 10.2.That discussion is beyond the scope of lab 2, but feel
   free to read ahead if you're interested. *)

(*.....................................................................
Exercise 8: Recall from Chapter 7 the definition of the function `prods`.
 *)

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

(* Using `sum`, `prods`, and `zip`, define a function `dotprod` that
takes the dot product of two integer lists (that is, the sum of the
products of corresponding elements of the lists; see
https://en.wikipedia.org/wiki/Dot_product if you want more
information, though it shouldn't be necessary). For example, you
should have:

    # dotprod [1; 2; 3] [0; 1; 2] ;;
    - : int = 8
    # dotprod [1; 2] [5; 10] ;;
    - : int = 25

Even without looking at the code for the functions, carefully looking
at the type signatures for `zip`, `prods`, and `sum` should give a
good idea of how you might combine these functions to implement
```

```
`dotproduct`.

If you've got the right idea, your implementation should be literally
a single short line of code. If it isn't, try it again, getting into
the functional programming zen mindset.
..........................................................................*)

let dotprod (a : int list) (b : int list) : int =
  sum (prods (zip a b)) ;;

(*======================================================================
Part 3: Higher-order functional programming with map, filter, and fold

In these exercises, you should use the built-in functions `map`,
`filter`, and `fold_left` and `fold_right` provided in the OCaml List
module to implement these simple functions.

  * IMPORTANT NOTE 1: When you make use of these functions, you'll
    either need to prefix them with the module name, for example,
    `List.map` or `List.fold_left`, or you'll need to open the `List`
    module with the line

        open List ;;

    You can place that line at the top of this file if you'd like.

  * IMPORTANT NOTE 2: In these labs, and in the problem sets as well,
    we'll often supply some skeleton code that looks like this:

        let somefunction (arg1 : type) (arg2 : type) : returntype =
          failwith "somefunction not implemented"

    We provide this to give you an idea of the function's intended
    name, its arguments and their types, and the return type. But
    there's no need to slavishly follow that particular way of
    implementing code to those specifications. In particular, you may
    want to modify the first line to introduce, say, a `rec` keyword
    (if your function is to be recursive):

        let rec somefunction (arg1 : type) (arg2 : type) : returntype =
          ...your further code here...

    Or you might want to define the function using anonymous function
    syntax. (If you haven't seen this yet, come back to this comment
    later when you have.)
```

```
        let somefunction =
          fun (arg1 : type) (arg2 : type) : returntype ->
            ...your further code here...

    This will be especially pertinent in this section, where functions
    can be built just by applying other higher order functions
    directly, without specifying the arguments explicitly, for
    example, in this implementation of the `double_all` function,
    which doubles each element of a list:

        let double_all : int list -> int list =
          map (( * ) 2) ;;

  * END IMPORTANT NOTES

.....................................................................
Exercise 9: Reimplement `sum` using `fold_left`, naming it `sum_ho`
(for "higher order").
.....................................................................*)

let sum_ho : int list -> int =
  List.fold_left (+) 0 ;;

(* One of the key advantages of curried functions (like `fold_left`)
   is that they can be partially applied. (See Section 8.2 in the
   textbook.) We've taken advantage of that in the definition above,
   by defining `sum_ho` as a partially applied `fold_left`, rather
   than as

     let sum_ho (lst : int list) : int =
       List.fold_left (+) 0 lst ;;

   The latter will work, but lacks the elegance of the more idiomatic
   approach here.

   The same technique is used in the exercises below. It may be
   useful, in order to understand what's going on, to try typing in
   longer and longer prefixes of an expression like `List.fold_left
   (+) 0 [1; 2; 3]` and watch the types closely.

     # List.fold_left ;;
     - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
     # List.fold_left (+) ;;
     - : int -> int list -> int = <fun>
     # List.fold_left (+) 0 ;;
     - : int list -> int = <fun>
```

```
      # List.fold_left (+) 0 [1; 2; 3] ;;
      - : int = 6

   You may also note the use of parentheses in the expression
   `(+)`. The `+` operator is an example of an "infix" operator, an
   operator that goes in between its arguments rather than in front of
   them. When using `+` as an argument to higher-order functions, we
   generally need to remove that infix property, so that it will be
   parsed as a prefix operator like most other functions. Wrapping the
   operator in parentheses induces this behavior.

      # 3 + 4 ;;
      - : int = 7
      # (+) 3 4
      - : int = 7
 *)

(*.....................................................................
Exercise 10: Reimplement prods : `(int * int) list -> int list` using
the `map` function. Call it `prods_ho`.
.....................................................................*)

let prods_ho : (int * int) list -> int list =
  List.map (fun (x, y) -> x * y) ;;

(*.....................................................................
Exercise 11: The OCaml List module provides -- in addition to the `map`,
`fold_left`, and `fold_right` higher-order functions -- several other
useful higher-order list manipulation functions. For instance, `map2` is
like `map`, but takes two lists instead of one along with a function of
two arguments and applies the function to corresponding elements of the
two lists to form the result list. (You can read about it at
https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html#VALmap2.)
Use `map2` to reimplement `zip` and call it `zip_ho`.
.....................................................................*)

let zip_ho : int list -> int list -> (int * int) list =
  List.map2 (fun first second -> first, second) ;;

(* Note the rejiggering of the first line to allow the function
   `zip_ho` to take advantage of partial application, so that `zip_ho`
   is the functional output of the higher-order function
   `map2`. Without the rejiggering, you'd probably implement it as:

    let zip_ho (x : int list) (y : int list) : (int * int) list =
      List.map2 (fun first second -> first, second) x y ;;
```

```
 *)

(*..................................................................
Exercise 12: Define a function `evens`, using these higher-order
functional programming techniques, that returns a list of all of the
even numbers in its argument list in the same order. For instance,

    # evens [1; 2; 3; 6; 5; 4] ;;
    - : int list = [2; 6; 4]
..................................................................*)

(* Again, without partial application:

    let evens (lst : int list) : int list =
      List.filter (fun n -> n mod 2 = 0) lst;;

   and with partial application: *)

let evens : int list -> int list =
  List.filter (fun n -> n mod 2 = 0) ;;
```