```
(*
                              CS51 Lab 2
                     More Functional Programming:
           Simple Data Structures and Higher-Order Functions
 *)
(* 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? Test your solution by uncommenting the examples
(removing the `(*` and `*)` lines at start and end) and verifying that
no typing error is generated.
......................................................................*)

(*   <--- remove this start of comment line

let exercise1a : ??? =
  (0.1, "hi") ;;

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

let exercise1c : ???  =
  fun (x, y) -> x + int_of_float y ;;

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

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

remove this end of comment line too ----> *)

(*......................................................................
Exercise 2: Update each expression below by changing the 0 in the last
line so that it 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 = 0 ;;

let exercise2b =
  let x, y, z = 4, [1; 3], true in
  let value =
    match y with
    | [] -> 0
    | h :: t -> h in
  value = 0 ;;

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 = 0 ;;

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 = 0 ;;

(*......................................................................
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
  | _ -> false, 0 ;;

(*======================================================================
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

You may want to add some tests for other functions in the lab to get
some practice with automated unit testing.
......................................................................*)

let rec square_all (lst : int list) : int list =
  failwith "square_all not implemented" ;;

let exercise4 = [] ;;

(*......................................................................
Exercise 5: Define a recursive function `sum` that sums the values in
its integer list argument. (What's a sensible return value for the sum
of the empty list?)
......................................................................*)

let rec sum (lst : int list) : int =
  failwith "sum not implemented" ;;

(*......................................................................
Exercise 6: Define a recursive function `max_list` that returns the
maximum element in a non-empty integer list. Don't worry about what
happens on an empty list. You may be warned by the compiler that "this
pattern-matching is not exhaustive." You may ignore this warning for
this lab.
......................................................................*)

let rec max_list (lst : int list) : int =
  failwith "max_list not implemented" ;;

(*......................................................................
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,

   # 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 =
  failwith "zip not implemented" ;;

(*......................................................................
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 =
  failwith "dotprod not implemented" ;;

(*======================================================================
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 (lst : int list) : int =
  failwith "sum_ho not implemented" ;;

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

let prods_ho (lst : (int * int) list) : int list =
  failwith "prods_ho not implemented" ;;

(*....................................................................
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 (x : int list) (y : int list) : (int * int) list =
  failwith "zip_ho not implemented" ;;

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

let evens (lst : int list) : int list =
  failwith "evens not implemented" ;;
```