Solution to CS3110

Sky

June 15, 2025

Contents

Chapter 4. The Basics of OCaml

Problem 1: date fun

Define a function that takes an integer **d** and string **m** as input and returns **true** just when **d** and **m** form a valid date. Here, a valid date has a month that is one of the following abbreviations: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec. The day must be a number that is between 1 and the minimum number of days in that month, inclusive. For example, if the month is Jan, then the day is between 1 and 31, inclusive, whereas if the month is Feb, then the day is between 1 and 28, inclusive. How terse (i.e., few and short lines of code) can you make your function? You can definitely do this in fewer than 12 lines.

```
let valid_date (d : int) (m : string) =
  let max_days =
    match m with
      | "Jan" | "Mar" | "May" | "Jul" | "Aug" | "Oct" | "Dec" -> 31
      | "Apr" | "Jun" | "Sept" | "Nov" -> 30
      | "Feb" -> 28
      | _ -> 0 (* Invalid month abbreviation *)
  in
  d >= 1 && d <= max_days
```

Problem 2: fib

Define a recursive function `fib : int -> int`, such that `fib n` is the nth number in the Fibonacci sequence, which is $1, 1, 2, 3, 5, 8, 13, \ldots$ That is:

- `fib 1 = 1`
- `fib 2 = 1`
- `fib n = fib (n - 1) + fib (n - 2)` for any $n > 2$

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n - 1) + fib (n - 2)
```

Problem 3: fib fast

Write a tail-recursive version of `fib`.

```
let fib n =
  let rec fib_tail n a b =
    if n = 0 then a
    else fib_tail (n - 1) b (a + b)
  in
  if n <= 0 then invalid_arg "fib: negative input"
  else fib_tail n 0 1
```

Problem 4: divide

Write a function `divide : numerator:float -> denominator:float -> float`. Apply your function to 1.0 and 2.0.

```
let divide ~numerator:x ~denominator:y = x /. y
```

Problem 5: average

Define an infix operator `+/.` to compute the average of two floating-point numbers. For example,

- $1.0 +/. 2.0 = 1.5$
- $0.0 +/. 0.0 = 0.0$

```
let (+/.) x y = (x +. y) /. 2.0
```

Chapter 5. Data and Types

Problem 1: list expressions

- Construct a list that has the integers 1 through 5 in it. Use the square bracket notation for lists.
- Construct the same list, but do not use the square bracket notation. Instead, use `::` and `[]`.
- Construct the same list again. This time, the following expression must appear in your answer: `[2; 3; 4]`. Use the `@` operator, and do not use `::`.

```
let lst1 = [1; 2; 3; 4; 5]
let lst2 = 1 :: 2 :: 3 :: 4 :: 5 :: []
let lst3 = [1] @ [2; 3; 4] @ [5]
```

Problem 2: product

Write a function `product` that returns the product of all the elements in a list. The product of all the elements of an empty list is 1.

```
let rec product = function
  | [] -> 1
  | h :: t -> h * product t
```

Problem 3: concat

Write a function that concatenates all the strings in a list. The concatenation of all the strings in an empty list is the empty string `""`.

```
let rec concat = function
  | [] -> ""
  | h :: t -> h ^ concat t
```

Problem 4: patterns

Using pattern matching, write three functions, one for each of the following properties. Your functions should return **true** if the input list has the property and **false** otherwise.

- the list's first element is `"bigred"`
- the list has exactly two or four elements; do not use the **length** function
- the first two elements of the list are equal

```
let is_bigred = function
  | "bigred" :: _ -> true
  | _ -> false

let two_or_four_elements = function
  | _ :: _ :: [] -> true
  | _ :: _ :: _ :: _ :: [] -> true
  | _ -> false

let eq_first_two = function
  | a :: b :: _ -> a = b
  | _ -> false
```

Problem 5: library

Consult the `List` standard library to solve these exercises:

- Write a function that takes an `int list` and returns the fifth element of that list, if such an element exists. If the list has fewer than five elements, return 0. Hint: `List.length` and `List.nth`.
- Write a function that takes an `int list` and returns the list sorted in descending order. Hint: `List.sort` with `Stdlib.compare` as its first argument, and `List.rev`.

```
let fifth_element lst =
  if List.length lst >= 5 then List.nth lst 4 else 0

let sort_list_descending lst =
  lst |> List.sort Stdlib.compare |> List.rev
```

Problem 6: library puzzle

- Write a function that returns the last element of a list. Your function may assume that the list is non-empty. Hint: Use two library functions, and do not write any pattern matching code of your own.
- Write a function `any_zeros : int list -> bool` that returns **true** if and only if the input list contains at least one 0. Hint: use one library function, and do not write any pattern matching code of your own.

Your solutions will be only one or two lines of code each.

```
let last_element lst =
  List.nth lst (List.length lst - 1)

let last_element' lst =
  lst |> List.rev |> List.hd

let any_zeros lst =
  List.exists (fun x -> x = 0) lst
```

Problem 7: take drop

- Write a function `take : int -> 'a list -> 'a list` such that `take n lst` returns the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return all of them.
- Write a function `drop : int -> 'a list -> 'a list` such that `drop n lst` returns all but the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return the empty list.

```
let rec take n lst =
  if n = 0 then [] else match lst with
    | [] -> []
    | x :: xs -> x :: take (n - 1) xs

let rec drop n lst =
  if n = 0 then lst else match lst with
    | [] -> []
    | x :: xs -> drop (n - 1) xs
```

Problem 8: take drop tail

Revise your solutions for `take` and `drop` to be tail recursive, if they aren't already. Test them on long lists with large values of `n` to see whether they run out of stack space. To construct long lists, use the `--` operator from the lists section.

```
let take_tr n lst =
  let rec take' n lst acc =
    match (n, lst) with
    | 0, _ -> List.rev acc
    | _, [] -> List.rev acc
    | n, hd :: tl -> take' (n - 1) tl (hd :: acc)
  in
  take' n lst []

let drop_tr = drop (* which is already tail recursive *)

let ( -- ) i j =
  let rec from i j acc = if i > j then acc else from i (j - 1) (j :: acc) in
  from i j []
```

Problem 9: unimodal

Write a function `is_unimodal : int list -> bool` that takes an integer list and returns whether that list is unimodal. A unimodal list is a list that monotonically increases to some maximum value then monotonically decreases after that value. Either or both segments (increasing or decreasing) may be empty. A constant list is unimodal, as is the empty list.

```
let is_unimodal lst =
  let rec is_decreasing = function
    | [] | [_] -> true
    | x :: y :: rest -> x >= y && is_decreasing (y :: rest)
  in
  let rec find_peak = function
    | [] | [_] -> true
    | x :: y :: rest ->
      if x <= y then find_peak (y :: rest) else is_decreasing (x :: y :: rest)
  in
  find_peak lst
```

Problem 10: powerset

Write a function `powerset : int list -> int list list` that takes a set `S` represented as a list and returns the set of all subsets of `S`. The order of subsets in the powerset and the order of elements in the subsets do not matter. Hint: Consider the recursive structure of this problem. Suppose you already have `p`, such that `p = powerset s`. How could you use `p` to compute `powerset (x :: s)`?

```
let rec powerset = function
  | [] -> [ [] ]
  | x :: s -> let p = powerset s in
    List.map (List.cons x) p @ p
```

Problem 11: print int list rec

Write a function `print_int_list : int list -> unit` that prints its input list, one number per line.

```
let rec print_int_list = function
  | [] -> ()
  | hd :: tl -> print_endline (string_of_int hd); print_int_list tl
```

Problem 12: print int list iter

Write a function `print_int_list' : int list -> unit` whose specification is the same as `print_int_list`. Do not use the keyword `rec` in your solution, but instead to use the List module function `List.iter`.

```
let print_int_list' lst =
  List.iter (fun x -> print_endline (string_of_int x)) lst
```

Problem 13: student

Assume the following type definition: `type student = {first_name : string; last_name : string; gpa : float}`
Give OCaml expressions that have the following types:

- `student`
- `student -> string * string` (a function that extracts the student's name)
- `string -> string -> float -> student` (a function that creates a student record)

```
type student = { first_name : string ; last_name : string ; gpa : float }

(* expression with type [student] *)
let s =
  { first_name = "Ezra"; last_name = "Cornell"; gpa = 4.3 }

(* expression with type [student -> string * string] *)
let get_full_name student =
  student.first_name, student.last_name

(* expression with type [string -> string -> float -> student] *)
let make_stud first last g =
  { first_name = first; last_name = last; gpa=g }
```

Problem 14: pokerecord

Here is a variant that represents a few Pokémon types: `type poketype = Normal` Fire | Water|

- Define the type `pokemon` to be a record with fields `name` (a string), `hp` (an integer), and `ptype` (a `poketype`).
- Create a record named `charizard` of type `pokemon` that represents a Pokémon with 78 HP and Fire type.
- Create a record named `squirtle` of type `pokemon` that represents a Pokémon with 44 HP and Water type.

```
type poketype = Normal | Fire | Water

(* define the type pokemon record *)
type pokemon = { name : string ; hp : int ; ptype : poketype }

let charizard = { name = "charizard"; hp = 78; ptype = Fire }

let squirtle = { name = "squirtle"; hp = 44; ptype = Water }
```

Problem 15: safe hd and tl

Write a function `safe_hd : 'a list -> 'a option` that returns `Some x` if the head of the input list is `x`, and `None` if the input list is empty. Also write a function `safe_tl : 'a list -> 'a list option` that returns the tail of the list, or `None` if the list is empty.

```
let safe_hd = function
  | [] -> None
  | h::_ -> Some h

let safe_tl = function
  | [] -> None
  | _::t -> Some t
```

Problem 16: pokefun

Write a function `max_hp : pokemon list -> pokemon option` that, given a list of `pokemon`, finds the Pokémon with the highest HP.

```
let max_hp = function
  | [] -> None
  | h :: t ->
    Some (List.fold_left (fun acc p -> if p.hp > acc.hp then p else acc) h t)
```

Problem 17: date before

Define a date-like triple to be a value of type `int * int * int`. A date is a date-like triple whose first part is a positive year, second part is a month between 1 and 12, and third part is a day between 1 and 31 (or 30, 29, or 28, depending on the month and year). You may ignore leap years. Write a function `is_before` that takes two dates as input and evaluates to **true** if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is **false**.)

```
type date = int * int * int

let is_before date1 date2 =
  let (y1, m1, d1) = date1 in
  let (y2, m2, d2) = date2 in
  y1 < y2 || (y1 = y2 && m1 < m2) || (y1 = y2 && m1 = m2 && d1 < d2)
```

Problem 18: earliest date

Write a function `earliest : (int * int * int) list -> (int * int * int) option`. It evaluates to `None` if the input list is empty, and to `Some d` if date `d` is the earliest date in the list. Hint: use `is_before`.

```
let rec earliest = function
  | [] -> None
  | d1::t -> begin
      match earliest t with
      | None -> Some d1
      | Some d2 -> Some (if is_before d1 d2 then d1 else d2)
    end
```

Problem 19: assoc list

Use the functions `insert` and `lookup` from the section on association lists to construct an association list that maps the integer 1 to the string "one", 2 to "two", and 3 to "three". Lookup the key 2. Lookup the key 4.

```
let insert k v d = (k,v)::d

(* find the value v to which key k is bound, if any, in the assocation list *)
```

```
let rec lookup k = function
  | [] -> None
  | (k',v)::t -> if k=k' then Some v else lookup k t

let dict = insert 3 "three" (insert 2 "two" (insert 1 "one" []))
let some_two = lookup 2 dict
let none = lookup 4 dict
```

Problem 20: cards

- Define a variant type `suit` that represents the four suits, `Clubs`, `Diamonds`, `Hearts`, and `Spades`, in a standard 52-card deck. All the constructors of your type should be constant.
- Define a type `rank` that represents the possible ranks of a card: 2, 3, ..., 10, Jack, Queen, King, or Ace.
- Define a type `card` that represents the suit and rank of a single card. Make it a record with two fields.
- Define a few values of type card: the Ace of Clubs, the Queen of Hearts, the Two of Diamonds, the Seven of Spades.

```
type suit = Hearts | Spades | Clubs | Diamonds

type rank = Number of int | Ace | Jack | Queen | King

type card = { suit: suit; rank: rank }

let ace_of_clubs : card = { suit = Clubs; rank = Ace }
let queen_of_hearts : card = { suit = Hearts; rank = Queen }
let two_of_diamonds : card = { suit = Diamonds; rank = Number 2 }
let seven_of_spades : card = { suit = Spades; rank = Number 7 }
```

Problem 21: quadrant

Complete the quadrant function below, which should return the quadrant of the given x, y point. Points that lie on an axis do not belong to any quadrant. Hints: (a) define a helper function for the sign of an integer, (b) match against a pair.

```
type quad = I | II | III | IV
type sign = Neg | Zero | Pos

(** [sign x] is [Zero] if [x = 0]; [Pos] if [x > 0]; and [Neg] if [x < 0]. *)
let sign x =
  if x = 0 then Zero
  else if x > 0 then Pos
  else Neg

(** [quadrant (x,y)] is [Some q] if [(x, y)] lies in quadrant [q], or [None] if
    it lies on an axis. *)
let quadrant (x,y) =
  match sign x, sign y with
  | Pos, Pos -> Some I
  | Neg, Pos -> Some II
  | Neg, Neg -> Some III
  | Pos, Neg -> Some IV
  | _ -> None
```

Problem 22: quadrant when

Rewrite the quadrant function to use the `when` syntax. You won't need your helper function from before.

```
let quadrant_when = function
  | x,y when x > 0 && y > 0 -> Some I
  | x,y when x < 0 && y > 0 -> Some II
  | x,y when x < 0 && y < 0 -> Some III
  | x,y when x > 0 && y < 0 -> Some IV
  | _ -> None
```

Problem 23: depth

Write a function `depth : 'a tree -> int` that returns the number of nodes in any longest path from the root to a leaf. For example, the depth of an empty tree (simply `Leaf`) is 0. Hint: there is a library function `max : 'a -> 'a -> 'a`.

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

let rec depth = function
  | Leaf -> 0
  | Node (_, left, right) -> 1 + max (depth left) (depth right)
```

Problem 24: shape

Write a function `same_shape : 'a tree -> 'b tree -> bool` that determines whether two trees have the same shape, regardless of whether the values they carry at each node are the same. Hint: use a pattern match with three branches, where the expression being matched is a pair of trees.

```
let rec same_shape t1 t2 =
  match t1, t2 with
  | Leaf, Leaf -> true
  | Node (_,l1,r1), Node (_,l2,r2) -> (same_shape l1 l2) && (same_shape r1 r2)
  | _ -> false
```

Problem 25: list max exn

Write a function `list_max : int list -> int` that returns the maximum integer in a list, or raises `Failure "list_max"` if the list is empty.

```
let list_max = function
  | [] -> failwith "list_max"
  | h :: t -> List.fold_left max h t
```

Problem 26: list max exn string

Write a function `list_max_string : int list -> string` that returns a string containing the maximum integer in a list, or the string `"empty"` if the list is empty.

```
let list_max_string lst =
  try string_of_int (list_max lst) with
  | Failure _ -> "empty"
```

Problem 27: `is_bst`

Write a function `is_bst : ('a * 'b) tree -> bool` that returns **true** if and only if the given tree satisfies the binary search tree invariant. An efficient version of this function that visits each node at most once is somewhat tricky to write. Hint: write a recursive helper function that takes a tree and either gives you (i) the minimum and maximum value in the tree, or (ii) tells you that the tree is empty, or (iii) tells you that the tree does not satisfy the invariant. You will need to define a new variant type for the return type of your helper function.

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

type 'a bst_check =
  | Empty
  | Invalid
  | Valid of 'a * 'a (* min, max *)

let rec is_bst_helper (t : 'a tree) : 'a bst_check =
  match t with
  | Leaf -> Empty
  | Node (value, left, right) -> (
    match (is_bst_helper left, is_bst_helper right) with
    | Empty, Empty -> Valid (value, value)
    | Valid (lmin, lmax), Empty ->
      if lmax <= value then Valid (lmin, value) else Invalid
    | Empty, Valid (rmin, rmax) ->
      if value <= rmin then Valid (value, rmax) else Invalid
    | Valid (lmin, lmax), Valid (rmin, rmax) ->
      if lmax <= value && value <= rmin then Valid (lmin, rmax) else Invalid
    | _ -> Invalid)

let is_bst (t : 'a tree) : bool =
  match is_bst_helper t with
  | Invalid -> false
  | _ -> true
```

Problem 28: quadrant poly

Modify your definition of quadrant to use polymorphic variants. The types of your functions should become these:

```
val sign : int -> [> `Neg | `Pos | `Zero ]
val quadrant : int * int -> [> `I | `II | `III | `IV ] option
```

```
let sign_poly x : [> `Neg | `Pos | `Zero] =
  if x < 0 then `Neg
  else if x = 0 then `Zero
  else `Pos

let quadrant (x,y) : [> `I | `II | `III | `IV ] option =
  match sign_poly x, sign_poly y with
  | `Pos, `Pos -> Some `I
  | `Neg, `Pos -> Some `II
  | `Neg, `Neg -> Some `III
  | `Pos, `Neg -> Some `IV
  | _ -> None
```

Chapter 6. Higher-Order Programming

Problem 1: repeat

Generalize `twice` to a function `repeat`, such that `repeat f n x` applies `f` to `x` a total of `n` times. That is,

- `repeat f 0 x` yields `x`
- `repeat f 1 x` yields `f x`
- `repeat f 2 x` yields `f (f x)` (which is the same as `twice f x`)
- `repeat f 3 x` yields `f (f (f x))`
- ...

```
let rec repeat f n x =
  if n = 0 then x else repeat f (n - 1) (f x)
```

Problem 2: product

Use `fold_left` to write a function `product_left` that computes the product of a list of floats. The product of the empty list is `1.0`. Hint: recall how we implemented sum in just one line of code in lecture. Use `fold_right` to write a function `product_right` that computes the product of a list of floats. Same hint applies.

```
let product_left lst = List.fold_left ( *. ) 1.0 lst

let product_right lst = List.fold_right ( *. ) lst 1.0
```

Problem 3: terse product

How terse can you make your solutions to the product exercise? Hints: you need only one line of code for each, and you do not need the `fun` keyword. For `fold_left`, your function definition does not even need to explicitly take a list argument. If you use `ListLabels`, the same is true for `fold_right`.

```
let terse_product_left = List.fold_left ( *. ) 1.0

let terse_product_right = ListLabels.fold_right ~f:( *. ) ~init:1.0
```

Problem 4: `sum_cube_odd`

Write a function `sum_cube_odd n` that computes the sum of the cubes of all the odd numbers between 0 and `n` inclusive. Do not write any new recursive functions. Instead, use the functionals `map`, `fold`, and `filter`, and the `( -- )` operator (defined in the discussion of pipelining).

```
let rec ( -- ) i j = if i > j then [] else i :: (i + 1 -- j)

let sum_cube_odd n =
  let odd x = x mod 2 = 1 in
  let cube x = x * x * x in
  List.filter odd (0 -- n) |> List.map cube |> List.fold_left (+) 0
```

Problem 5: `sum_cube_odd` pipeline

Rewrite the function `sum_cube_odd` to use the pipeline operator >|.

```
let sum_cube_odd_p n =
  0 -- n
  |> List.filter (fun i -> i mod 2 = 1)
  |> List.map (fun i -> i * i * i)
  |> List.fold_left (+) 0
```

Problem 6: exists

Consider writing a function `exists: ('a -> bool) -> 'a list -> bool`, such that `exists p [a1; ...; an]` returns whether at least one element of the list satisfies the predicate `p`. That is, it evaluates the same as `(p a1) || (p a2) || ... || (p an)`. When applied to an empty list, it evaluates to `false`. Write three solutions to this problem, as we did above:

- `exists_rec`, which must be a recursive function that does not use the `List` module,
- `exists_fold`, which uses either `List.fold_left` or `List.fold_right`, but not any other `List` module functions nor the `rec` keyword, and

- `exists_lib`, which uses any `List` module functions except `fold_left` or `fold_right`, and does not use `rec`.

```
let rec exists_rec p = function
  | [] -> false
  | hd :: tl -> p hd || exists_rec p tl

let exists_fold p lst = List.fold_left (fun acc x -> acc || p x) false lst

let exists_lib = List.exists
```

Problem 7: account balance

Write a function which, given a list of numbers representing debits, deducts them from an account balance, and finally returns the remaining amount in the balance. Write three versions: `fold_left`, `fold_right`, and a direct recursive implementation.

```
let balance_left balance debits =
  List.fold_left ( -. ) balance debits

let balance_right balance debits =
  List.fold_right (fun d b -> b -. d) debits balance

let rec balance_rec balance = function
  | [] -> balance
  | h :: t -> balance_rec (balance -. h) t
```

Problem 8: library uncurried

Here is an uncurried version of `List.nth`:

```
let uncurried_nth (lst, n) = List.nth lst n
```

In a similar way, write uncurried versions of these library functions:

- `List.append`
- `Char.compare`
- `Stdlib.max`

```
let uncurried_append (lst,e) = List.append lst e

let uncurried_compare (c1,c2) = Char.compare c1 c2

let uncurried_max (n1,n2) = Stdlib.max n1 n2
```

Problem 9: map composition

Show how to replace any expression of the form `List.map f (List.map g lst)` with an equivalent expression that calls `List.map` only once.

```
List.map (fun elt -> f (g elt)) lst
List.map (f @@ g) lst
```

Problem 10: more list fun

Write functions that perform the following computations. Each function that you write should use one of `List.fold`, `List.map` or `List.filter`. To choose which of those to use, think about what the computation is doing: combining, transforming, or filtering elements.

- Find those elements of a list of strings whose length is strictly greater than 3.
- Add `1.0` to every element of a list of floats.
- Given a list of strings `strs` and another string `sep`, produce the string that contains every element of `strs` separated by `sep`. For example, given inputs `["hi";"bye"]` and `","`, produce `"hi,bye"`, being sure not to produce an extra comma either at the beginning or end of the result string.

```
let at_least_three lst =
  List.filter (fun s -> String.length s > 3) lst

let add_one lst =
  List.map (fun x -> x +. 1.0) lst

let join_with strs sep =
  match strs with
  | [] -> ""
  | x :: xs ->
    List.fold_left (fun combined s -> combined ^ sep ^ s) x xs
```

Problem 11: association list keys

Recall that an association list is an implementation of a dictionary in terms of a list of pairs, in which we treat the first component of each pair as a key and the second component as a value. Write a function `keys: ('a * 'b) list -> 'a list` that returns a list of the unique keys in an association list. Since they must be unique, no value should appear more than once in the output list. The order of values output does not matter. How compact and efficient can you make your solution? Can you do it in one line and linearithmic space and time? Hint: `List.sort_uniq`.

```
(* here are a few solutions of varying efficiency *)

(* returns: a list of the unique keys in [lst] in no particular order.
 * efficiency:  O(n^2) time, where n is the number of elements in [lst],
 * and O(n) stack space.
*)
let keys1 lst =
  List.fold_right
    (fun (k, _) acc -> k :: List.filter (fun k2 -> k <> k2) acc)
    lst
    []

(* returns: a list of the unique keys in [lst] in no particular order.
 * efficiency:  O(n^2) time, where n is the number of elements in [lst],
 * and O(1) stack space.
*)
let keys2 lst =
  List.fold_left
    (fun acc (k, _) -> if List.exists ((=) k) acc then acc else k::acc)
    []
    lst

(* returns: a list of the unique keys in [lst] in no particular order.
 * efficiency:  O(n log n) time, where n is the number of elements in [lst],
 * and O(log n) stack space.
*)
let keys3 lst =
  lst
  |> List.rev_map fst
  |> List.sort_uniq Stdlib.compare
```

Problem 12: valid matrix

A mathematical matrix can be represented with lists. In row-major representation, this matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 9 & 8 & 7 \end{bmatrix}$$

would be represented as the list `[[1; 1; 1]; [9; 8; 7]]`. Let's represent a row vector as an `int list`. For example, `[9; 8; 7]` is a row vector. A valid matrix is an `int list list` that has at least one row, at least one column, and in which every column has the same number of rows. There are many values of type `int list list` that are invalid, for example,

- `[]`
- `[[1; 2]; [3]]`

Implement a function `is_valid_matrix: int list list -> bool` that returns whether the input matrix is valid.

```
let is_valid_matrix = function
  | [] -> false
  | r :: rows ->
    let m = List.length r in
    m > 0 && List.for_all (fun r' -> m = List.length r') rows
```

Problem 13: row vector add

Implement a function `add_row_vectors: int list -> int list -> int list` for the element-wise addition of two row vectors. For example, the addition of `[1; 1; 1]` and `[9; 8; 7]` is `[10; 9; 8]`. If the two vectors do not have the same number of entries, the behavior of your function is unspecified–that is, it may do whatever you like. Hint: there is an elegant one-line solution using `List.map2`.

```
let add_row_vectors v1 v2 = List.map2 ( + ) v1 v2
```

Problem 14: matrix add

Implement a function `add_matrices: int list list -> int list list -> int list list` for matrix addition. If the two input matrices are not the same size, the behavior is unspecified. Hint: there is an elegant one-line solution using `List.map2` and `add_row_vectors`.

```
let add_matrices = List.map2 add_row_vectors
```

Problem 15: matrix multiply

Implement a function `multiply_matrices: int list list -> int list list -> int list list` for matrix multiplication. If the two input matrices are not of sizes that can be multiplied together, the behavior is unspecified. Unit test the function. Hint: define functions for matrix transposition and row vector dot product.

```
let dot_product = List.fold_left2 (fun acc x y -> acc + x * y) 0

let rec transpose = function
  | [] | [] :: _ -> []
  | rows ->
      let heads = List.map List.hd rows in
      let tails = List.map List.tl rows in
      heads :: transpose tails

(* Validation helper *)
let validate_matrices m1 m2 =
  match m1, m2 with
```

```
  | [], _ | _, [] -> false
  | _ :: _, [] :: _ -> false
  | rows1, rows2 ->
      let cols1 = List.length (List.hd rows1) in
      let cols2 = List.length (List.hd rows2) in
      cols1 = List.length rows2 &&
      List.for_all (fun row -> List.length row = cols1) rows1 &&
      List.for_all (fun row -> List.length row = cols2) rows2

let matrix_multiply m1 m2 =
  if not (validate_matrices m1 m2) then
    failwith "Invalid matrix dimensions for multiplication"
  else
    let m2_t = transpose m2 in
    List.map (fun row1 ->
      List.map (fun col2 ->
        dot_product row1 col2
      ) m2_t
    ) m1
```

Chapter 9. Mutability

Problem 1: mutable fields

Define an OCaml record type to represent student names and GPAs. It should be possible to mutate the value of a student's GPA. Write an expression defining a student with name `"Alice"` and GPA 3.7. Then write an expression to mutate Alice's GPA to 4.0.

```
type student = {name: string; mutable gpa: float}

let alice = {name = "Alice"; gpa = 3.7}

let () = alice.gpa <- 4.0
```

Problem 2: refs

Give OCaml expressions that have the following types. Use utop to check your answers.

- `bool ref`
- `int list ref`
- `int ref list`

```
let (_ : bool ref) = ref true
let (_ : int list ref) = ref [5;3]
let (_ : int ref list) = [ref 5; ref 3]
```

Problem 3: inc fun

Define a reference to a function as follows:

```
let inc = ref (fun x -> x + 1)
```

Write code that uses `inc` to produce the value 3110.

```
let cs3110 =
  let inc = ref (fun x -> x + 1) in
  !inc 3109
```

Problem 4: addition assignment

The C language and many languages derived from it, such as Java, has an addition assignment operator written `a += b` and meaning `a = a + b`. Implement such an operator in OCaml; its type should be `int ref -> int -> unit`.

```
let (+:=) x y =
  x := !x + y
```

Problem 5: norm

The Euclidean norm of an n-dimensional vector $x = (x_1, \ldots, x_n)$ is written $|x|$ and is defined to be $\sqrt{x_1^2 + \cdots + x_n^2}$. Write a function `norm : vector -> float` that computes the Euclidean norm of a vector, where `vector` is defined as follows:

```
(* AF: the float array [| x1; ...; xn |] represents the
 * vector (x1, ..., xn)
 * RI: the array is non-empty *)
type vector = float array
```

Your function should not mutate the input array. Hint: try to use `Array.map` and `Array.fold_left` or `Array.fold_right`.

```
let norm v =
  sqrt (Array.fold_left (fun acc x -> acc +. x ** 2.) 0. v)
```

Problem 6: normalize

Every vector x can be normalized by dividing each component by $|x|$; this yields a vector with norm 1: $(\frac{x_1}{|x|}, \ldots, \frac{x_n}{|x|})$. Write a function `normalize : vector -> unit` that normalizes a vector "in place" by mutating the input array. Hint: `Array.iteri`.

```
let normalize v =
  let n = norm v in (* Must calculate norm before iteration *)
  Array.iteri (fun i x -> v.(i) <- x /. n) v

(* since OCaml 5.1 *)
let normalize' (v : vector) =
  let n = norm v in
  Array.map_inplace (fun x -> x /. n) v
```

Problem 7: norm loop

Modify your implementation of `norm` to use a loop.

```
let norm_loop v =
  let n = ref 0.0 in
  for i = 0 to Array.length v - 1 do
    n := !n +. (v.(i) ** 2.)
  done;
  sqrt !n
```

Problem 8: normalize loop

Modify your implementation of `normalize` to use a loop.

```
let normalize_loop v =
  let n = norm v in
  for i = 0 to Array.length v - 1 do
    v.(i) <- v.(i) /. n
  done
```

Problem 9: fact loop

Modify your implementation of `fact` to use a loop.

```
let fact_loop n =
  let ans = ref 1 in
  for i = 1 to n do
    ans := !ans * i
  done;
  !ans
```

Problem 10: init matrix

The `Array` module contains two functions for creating an array: `make` and `init`. `make` creates an array and fills it with a default value, while `init` creates an array and uses a provided function to fill it in. The library also contains a function `make_matrix` for creating a two-dimensional array, but it does not contain an analogous `init_matrix` to create a matrix using a function for initialization. Write a function `init_matrix : int -> int -> (int -> int -> 'a) -> 'a array array` such that `init_matrix n o f` creates and returns an `n` by `o` matrix `m` with `m.(i).(j) = f i j` for all `i` and `j` in bounds.

```
let init_matrix n o f =
  Array.init n (fun i -> Array.init o (fun j -> f i j))
```