Solution to CS3110

Sky

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`.

```
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 7. Modular Programming

Problem 1: complex synonym

Here is a module type for complex numbers, which have a real and imaginary component:

```
module type ComplexSig = sig
  val zero : float * float
  val add : float * float -> float * float -> float * float
end
```

Improve that code by adding `type t = float * float`. Show how the signature can be written more tersely because of the type synonym.

```
module type ComplexSig = sig
  type t = float * float
  val zero : t
  val add : t -> t -> t
end
```

Problem 2: big list queue

Use the following code to create `ListQueue` of exponentially increasing length: 10, 100, 1000, etc. How big of a queue can you create before there is a noticeable delay? How big until there's a delay of at least 10 seconds? (Note: you can abort utop computations with Ctrl-C.)

```
(** Creates a ListQueue filled with [n] elements. *)
let fill_listqueue n =
  let rec loop n q =
    if n = 0 then q
    else loop (n - 1) (ListQueue.enqueue n q)
  in loop n ListQueue.empty
```

```
module ListQueue = struct
  type 'a queue = 'a list

  let empty = []

  let is_empty q = (q = [])

  let enqueue x q = q @ [x]

  let peek = function
    | []   -> failwith "Empty"
    | x::_ -> x

  let dequeue = function
    | []   -> failwith "Empty"
    | _::q -> q
end
```

Problem 3: big batched queue

Use the following function to create `BatchedQueue` of exponentially increasing length:

```
let fill_batchedqueue n =
  let rec loop n q =
    if n = 0 then q
    else loop (n - 1) (BatchedQueue.enqueue n q)
  in loop n BatchedQueue.empty
```

Now how big of a queue can you create before there's a delay of at least 10 seconds?

```
module BatchedQueue = struct
  type 'a t = {outbox:'a list; inbox:'a list}

  let empty = {outbox=[]; inbox=[]}

  let is_empty = function
    | {outbox=[]; inbox=[]} -> true
    | _ -> false
```

```
  let norm = function
    | {outbox=[]; inbox} -> {outbox=List.rev inbox; inbox=[]}
    | q -> q

  let enqueue x q = norm {q with inbox=x::q.inbox}

  let peek = function
    | {outbox=[]; _} -> None
    | {outbox=x::_; _} -> Some x

  let dequeue = function
    | {outbox=[]; _} -> None
    | {outbox=_::xs; inbox} -> Some (norm {outbox=xs; inbox})
end
```

Problem 4: binary search tree map

Write a module `BstMap` that implements the `Map` module type using a binary search tree type. Binary trees were covered earlier when we discussed algebraic data types. A binary search tree (BST) is a binary tree that obeys the following BST Invariant:

> For any node `n`, every node in the left subtree of `n` has a value less than `n`'s value, and every node in the right subtree of `n` has a value greater than `n`'s value.

Your nodes should store pairs of keys and values. The keys should be ordered by the BST Invariant. Based on that invariant, you will always know whether to look left or right in a tree to find a particular key.

```
module type Map = sig
  type ('k, 'v) t
  val empty  : ('k, 'v) t
  val insert : 'k -> 'v -> ('k,'v) t -> ('k,'v) t
  val lookup  : 'k -> ('k,'v) t -> 'v
end

module BstMap : Map = struct
  type 'a tree =
    | Leaf
    | Node of 'a * 'a tree * 'a tree

  type ('k, 'v) t = ('k * 'v) tree

  let empty =
    Leaf

  let rec insert k v = function
    | Leaf -> Node((k, v), Leaf, Leaf)
    | Node ((k',v'), l, r) ->
      if (k = k') then Node ((k, v), l, r)
      else if (k < k') then Node ((k',v'), insert k v l, r)
      else Node ((k',v'), l, insert k v r)

  let rec lookup k = function
    | Leaf -> failwith "Not_found"
    | Node ((k',v'), l, r) ->
      if (k = k') then v'
      else if (k < k') then lookup k l
      else lookup k r
end
```

Problem 5: fraction

Write a module that implements the `Fraction` module type below:

```
module type Fraction = sig
```

```
  (* A fraction is a rational number p/q, where q != 0. *)
  type t

  (** [make n d] represents n/d, a fraction with
      numerator [n] and denominator [d].
      Requires d <> 0. *)
  val make : int -> int -> t
  val numerator : t -> int
  val denominator : t -> int
  val to_string : t -> string
  val to_float : t -> float
  val add : t -> t -> t
  val mul : t -> t -> t
end
```

```
module PairFraction : Fraction = struct
  type t = {
    num : int;
    den : int;
  }

  exception Division_by_zero

  let rec gcd x y = if y = 0 then x else gcd y (x mod y)

  let make num den =
    if den = 0 then raise Division_by_zero
    else
      let g = gcd (abs num) (abs den) in
      let sign = if den < 0 then -1 else 1 in
      {num = sign * num / g; den = abs (den / g)}

  let numerator {num; _} = num
  let denominator {den; _} = den
  let to_string {num; den} = string_of_int num ^ "/" ^ string_of_int den
  let to_float {num; den} = float_of_int num /. float_of_int den

  let add {num = num1; den = den1} {num = num2; den = den2} =
    make ((num1 * den2) + (num2 * den1)) (den1 * den2)

  let mul {num = num1; den = den1} {num = num2; den = den2} =
    make (num1 * num2) (den1 * den2)
end
```

Problem 6: use char map

Using the `CharMap` you just made, create a map that contains the following bindings:

- `'A'` maps to `"Alpha"`
- `'E'` maps to `"Echo"`
- `'S'` maps to `"Sierra"`
- `'V'` maps to `"Victor"`

Use `CharMap.find` to find the binding for `'E'`.

Now remove the binding for `'A'`. Use `CharMap.mem` to find whether `'A'` is still bound.

Use the function `CharMap.bindings` to convert your map into an association list.

```
module CharMap = Map.Make(Char)
let map = CharMap.(
    empty
    |> add 'A' "Alpha"
    |> add 'E' "Echo"
    |> add 'S' "Sierra"
```

```
    |> add 'V' "Victor"
  )
let echo = CharMap.find 'E' map (* "Echo" *)
let map' = CharMap.remove 'A' map
let a_exists = CharMap.mem 'A' map' (* false *)
let bindings = CharMap.bindings map'
```

Problem 7: date order

Here is a type for dates:

```
type date = {month : int; day : int}
```

For example, March 31st would be represented as `{month = 3; day = 31}`. Our goal in the next few exercises is to implement a map whose keys have type `date`.

Obviously it's possible to represent invalid dates with type `date`—for example, `{ month=6; day=50 }` would be June 50th, which is not a real date. The behavior of your code in the exercises below is unspecified for invalid dates.

To create a map over dates, we need a module that we can pass as input to `Map.Make`. That module will need to match the `Map.OrderedType` signature. Create such a module. Here is some code to get you started:

```
module Date = struct
  type t = date
  let compare ...
end
```

Recall the specification of `compare` in `Map.OrderedType` as you write your `Date.compare` function.

```
type date = {
  month : int;
  day : int
}

module Date = struct
  type t = date

  let compare d1 d2 =
    if d1.month = d2.month then d1.day - d2.day
    else d1.month - d2.month
end
```

Problem 8: calendar

Use the `Map.Make` functor with your `Date` module to create a `DateMap` module. Then define a calendar type as follows:

```
type calendar = string DateMap.t
```

The idea is that `calendar` maps a date to the name of an event occurring on that date.

Using the functions in the `DateMap` module, create a calendar with a few entries in it, such as birthdays or anniversaries.

```
module DateMap = Map.Make (Date)

type calender = string DateMap.t

let my_calendar =
  DateMap.(
    empty
    |> add {month = 2; day = 7} "e day"
    |> add {month = 3; day = 14} "pi day"
    |> add {month = 6; day = 18} "phi day"
    |> add {month = 10; day = 23} "mole day"
    |> add {month = 11; day = 23} "fibonacci day")
```

Problem 9: print calendar

Write a function `print_calendar : calendar -> unit` that prints each entry in a calendar in a format similar to the inspiring examples in the previous exercise. Hint: use `DateMap.iter`, which is documented in the `Map.S` signature.

```
let print_calendar =
  DateMap.iter (fun date event ->
      Printf.printf "%d/%d: %s\n" date.month date.day event)
```

Problem 10: is for

Write a function `is_for : string CharMap.t -> string CharMap.t` that given an input map with bindings from k_1 to v_1, ..., k_n to v_n, produces an output map with the same keys, but where each key k_i is now bound to the string "k_i is for v_i". For example, if `m` maps `'a'` to `"apple"`, then `is_for m` would map `'a'` to `"a is for apple"`. Hint: there is a one-line solution that uses a function from the `Map.S` signature. To convert a character to a string, you could use `String.make`. An even fancier way would be to use `Printf.sprintf`.

```
let is_for m =
  CharMap.mapi (fun key v -> Printf.sprintf "%c is for %s" key v) m
```

Problem 11: first after

Write a function `first_after : calendar -> Date.t -> string` that returns the name of the first event that occurs strictly after the given date. If there is no such event, the function should raise `Not_found`, which is an exception already defined in the standard library. Hint: you can do this in one-line by using a function or two from the `Map.S` signature.

```
let first_after date calendar =
  DateMap.(
    split date calendar |> fun (_, _, after) ->
    after |> DateMap.min_binding |> snd)
```

Problem 12: sets

The standard library `Set` module is quite similar to the `Map` module. Use it to create a module that represents sets of case-insensitive strings. Strings that differ only in their case should be considered equal by the set. For example, the sets `grr", "argh""grr", "argh"` and `aRgh", "GRR""aRgh", "GRR"` should be considered the same, and adding `"gRr"` to either set should not change the set.

```
module CisSet = Set.Make(struct
    type t = string
    let compare s1 s2 =
      String.compare (String.lowercase_ascii s1) (String.lowercase_ascii s2)
  end)

let _ = CisSet.(equal (of_list ["grr"; "argh"]) (of_list ["GRR"; "aRgh"]))
```

Problem 13: ToString

Write a module type `ToString` that specifies a signature with an abstract type `t` and a function `to_string : t -> string`.

```
module type ToString = sig
  type t
  val to_string: t -> string
end
```

Problem 14: Print

Write a functor `Print` that takes as input a module named `M` of type `ToString`. The module returned by your functor should have exactly one value in it, `print`, which is a function that takes a value of type `M.t` and prints a string representation of that value.

```
module Print (M : ToString) = struct
  let print v = print_string (M.to_string v)
end
```

Problem 15: Print Int

Create a module named `PrintInt` that is the result of applying the functor `Print` to a new module `Int`. You will need to write `Int` yourself. The type `Int.t` should be `int`. Hint: do not seal `Int`.

```
module Int = struct
  type t = int

  let to_string = string_of_int
end
```

Problem 16: Print String

Create a module named `PrintString` that is the result of applying the functor `Print` to a new module `MyString`. You will need to write `MyString` yourself. Hint: do not seal `MyString`.

```
module MyString = struct
  type t = string

  let to_string s = s
end
```

Problem 17: Print String reuse revisited

The `PrintString` module you created above supports just one operation: `print`. It would be great to have a module that supports all the `String` module functions in addition to that `print` operation, and it would be super great to derive such a module without having to copy any code.

Define a module `StringWithPrint`. It should have all the values of the built-in `String` module. It should also have the `print` operation, which should be derived from the `Print` functor rather than being copied code. Hint: use two `include` statements.

```
module StringWithPrint = struct
  include String
  include Print(MyString)
end
```

Problem 18: date

```
(* date.mli *)
type date
val make_date : int -> int -> date
val get_month : date -> int
val get_day : date -> int
val to_string : date -> string
val format : Format.formatter -> date -> unit

(* date.ml *)
type date = {
  month : int;
  day : int;
}
```

```
let make_date month day = {month; day}
let get_month d = d.month
let get_day d = d.day
let to_string d = string_of_int d.month ^ "/" ^ string_of_int d.day

let format fmt d = Format.fprintf fmt "%s" (to_string d)
```

Problem 19: refactor arith

Download this file: `algebra.ml`. It contains these signatures and structures:

- `Ring` is a signature that describes the algebraic structure called a ring, which is an abstraction of the addition and multiplication operators.
- `Field` is a signature that describes the algebraic structure called a field, which is like a ring but also has an abstraction of the division operation.
- `IntRing` and `FloatRing` are structures that implement rings in terms of `int` and `float`.
- `IntField` and `FloatField` are structures that implement fields in terms of `int` and `float`.
- `IntRational` and `FloatRational` are structures that implement fields in terms of ratios (aka fractions) —that is, pairs of `int` and pairs of `float`.

  Note: Dear fans of abstract algebra: of course these representations don't necessarily obey all the axioms of rings and fields because of the limitations of machine arithmetic. Also, the division operation in `IntField` is ill-defined on zero. Try not to worry about that.

Refactor the code to improve the amount of code reuse it exhibits. To do that, use `include`, functors, and introduce additional structures and signatures as needed. There isn't necessarily a right answer here, but here's some advice:

- No name should be directly declared in more than one signature. For example, `( + )` should not be directly declared in `Field`; it should be reused from an earlier signature. By "directly declared" we mean a declaration of the form `val name : ...`. An indirect declaration would be one that results from an `include`.
- You need only three direct definitions of the algebraic operations and numbers (plus, minus, times, divide, zero, one): once for `int`, once for `float`, and once for ratios. For example, `IntField.( + )` should not be directly defined as `Stdlib.( + )`; rather, it should be reused from elsewhere. By "directly defined" we mean a definition of the form `let name = ...`. An indirect definition would be one that results from an `include` or a functor application.
- The rational structures can both be produced by a single functor that is applied once to `IntRing` and once to `FloatRing`.
- It's possible to eliminate all duplication of `of_int`, such that it is directly defined exactly once, and all structures reuse that definition; and such that it is directly declared in only one signature. This will require the use of functors. It will also require inventing an algorithm that can convert an integer to an arbitrary `Ring` representation, regardless of what the representation type of that `Ring` is.

When you're done, the types of all the modules should remain unchanged. You can easily see those types by running `ocamlc -i algebra.ml`.

```
module type PreRing = sig
  type t
  val zero   : t
  val one    : t
  val (+)    : t -> t -> t
  val (~-)   : t -> t
  val ( * ) : t -> t -> t
  val to_string : t -> string
end
```

```
module type OfInt = sig
  type t
  val of_int : int -> t
end

module type Ring = sig
  include PreRing
  include OfInt with type t := t
end

module type PreField = sig
  include PreRing
  val (/) : t -> t -> t
end

module type Field = sig
  include PreField
  include OfInt with type t := t
end

module RingOfPreRing (R:PreRing) = (struct
  include R
  let of_int n =
    let two = one + one in
    (* [loop n b x] is [nb + x] *)
    let rec loop n b x =
      if n=0 then x
      else loop Stdlib.(n/2) (b*two)
          (if n mod 2 = 0 then x else x+b)
    in
    let m = loop (abs n) one zero in
    if n<0 then -m else m
end : Ring with type t = R.t)

module FieldOfPreField (F:PreField) = (struct
  module R : (OfInt with type t := F.t) = RingOfPreRing(F)
  include F
  include R
end : Field)

module IntPreRing = struct
  type t = int
  let zero = 0
  let one = 1
  let (+) = (+)
  let (~-) = (~-)
  let ( * ) = ( * )
  let to_string = string_of_int
end
```

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:

$$\left(\frac{x_1}{|x|}, \ldots, \frac{x_n}{|x|}\right).$$

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))
```

Chapter 10. Data Structures

Problem 1: hashtbl usage

Create a hash table tab with `Hashtbl.create` whose initial size is 16. Add 31 bindings to it with `Hashtbl.add`. For example, you could add the numbers 1..31 as keys and the strings "1"..."31" as their values. Use

`Hashtbl.find` to look for keys that are in tab, as well as keys that are not.

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

let tab = Hashtbl.create 16

let ints = List.map (fun x -> (x, string_of_int x)) (1 -- 31)

let () = List.iter (fun (k, v) -> Hashtbl.add tab k v) ints

let () = assert (Hashtbl.find tab 1 = "1")
let () = assert ((try Hashtbl.find tab 0 with Not_found -> "") = "")
```

Problem 2: hashtbl stats

Use the `Hashtbl.stats` function to find out the statistics of `tab` (from an exercise above). How many buckets are in the table? How many buckets have a single binding in them?

```
let buckets h =
  (Hashtbl.stats h).num_buckets

let () = assert (buckets tab = 16)

let single_binding h =
  (Hashtbl.stats h).bucket_histogram.(1)

let () = assert (single_binding tab = 3)
```

Problem 3: hashtbl bindings

Define a function `bindings : ('a, 'b) Hashtbl.t -> ('a * 'b) list`, such that `bindings` h returns a list of all bindings in `h`. Use your function to see all the bindings in `tab` (from an exercise above). Hint: fold.

```
let bindings h =
  Hashtbl.fold (fun k v acc -> (k, v) :: acc) h []
```

Problem 4: hashtbl load factor

Define a function `load_factor : ('a,'b) Hashtbl.t -> float`, such that `load_factor h` is the load factor of `h`. What is the load factor of `tab`? Hint: stats. Add one more binding to `tab`. Do the stats or load factor change? Now add yet another binding. Now do the stats or load factor change? Hint: `Hashtbl` resizes when the load factor goes strictly above 2.

```
let (/..) x y =
 float_of_int x /. float_of_int y

let load_factor h =
 let stats = Hashtbl.stats h in
 stats.num_bindings /.. stats.num_buckets

let epsilon = 0.1

let close_to x y =
 abs_float (x -. y) <= epsilon

let () = Hashtbl.add tab 0 "0"; assert (not (close_to (load_factor tab) 1.0))
let () = Hashtbl.add tab ~-1 "-1"; assert (close_to (load_factor tab) 1.0)
```

Problem 5: functorial interface

Use the functorial interface (i.e., `Hashtbl.Make`) to create a hash table whose keys are strings that are case-insensitive. Be careful to obey the specification of `Hashtbl.HashedType.hash`:

> If two keys are equal according to `equal`, then they have identical hash values as computed by `hash`.

```
module CaseInsensitiveHashtbl =
  Hashtbl.Make (struct
    type t = string

    let equal s1 s2 =
      String.lowercase_ascii s1 = String.lowercase_ascii s2

    let hash s =
      Hashtbl.hash (String.lowercase_ascii s)
  end)
```

Problem 6: bad hash

Use the functorial interface to create a hash table with a really bad hash function (e.g., a constant function). Use the `stats` function to see how bad the bucket distribution becomes.

```
module BadHashtbl =
  Hashtbl.Make (struct
    type t = int
    let equal = (=)
    let hash _ = 0
  end)

let bad = BadHashtbl.create 16

let () =
  1--100
  |> List.map (fun x -> x, string_of_int x)
  |> List.iter (fun (k,v) -> BadHashtbl.add bad k v)

(* there is now one bucket that has 100 bindings *)
let () = assert ((BadHashtbl.stats bad).bucket_histogram.(100) = 1)
```

Problem 7: linear probing

We briefly mentioned *probing* as an alternative to chaining. Probing can be effectively used in hardware implementations of hash tables, as well as in databases. With probing, every bucket contains exactly one binding. In case of a collision, we search forward through the array, as described below.

Your task: Implement a hash table that uses linear probing. The details are below.

Find. Suppose we are trying to find a binding in the table. We hash the binding's key and look in the appropriate bucket. If there is already a different key in that bucket, we start searching forward through the array at the next bucket, then the next bucket, and so forth, wrapping back around to the beginning of the array if necessary. Eventually we will either

- find an empty bucket, in which case the key we're searching for is not bound in the table;
- find the key before we reach an empty bucket, in which case we can return the value; or
- never find the key or an empty bucket, instead wrapping back around to the original bucket, in which case all buckets are full and the key is not bound in the table. This case actually should never occur, because we won't allow the load factor to get high enough for all buckets to be filled.

Insert. Insertion follows the same algorithm as finding a key, except that whenever we first find an empty bucket, we can insert the binding there.

Remove. Removal is more difficult. Once the key is found, we can't just make the bucket empty, because that would affect future searches by causing them to stop early. Instead, we can introduce a special "deleted" value into that bucket to indicate that the bucket does not contain a binding but the searches should not stop at it.

Resizing. Since we never want the array to become completely full, we can keep the load factor near 1/4. When the load factor exceeds 1/2, we can double the array, bringing the load factor back to 1/4. When the load factor goes below 1/8, we can half the array, again bringing the load factor back to 1/4. "Deleted" bindings complicate the definition of load factor:

- When determining whether to double the table size, we calculate the load factor as (number of bindings + number of deleted bindings) / (number of buckets). That is, deleted bindings contribute toward increasing the load factor.
- When determining whether the half the table size, we calculate the load factor as (number of bindings) / (number of buckets). That is, deleted bindings do not count toward increasing the load factor.

When rehashing the table, deleted bindings are of course not re-inserted into the new table.

```
module type TableMap = sig
  type ('k, 'v) t

  val insert : 'k -> 'v -> ('k, 'v) t -> unit
  val find : 'k -> ('k, 'v) t -> 'v option
  val remove : 'k -> ('k, 'v) t -> unit
  val create : ('k -> int) -> int -> ('k, 'v) t
  val bindings : ('k, 'v) t -> ('k * 'v) list
  val of_list : ('k -> int) -> ('k * 'v) list -> ('k, 'v) t
end

module LinearProbingHashMap : TableMap = struct
  type ('k, 'v) content =
    | Empty
    | Deleted
    | Full of 'k * 'v

  type ('k, 'v) t = {
    hash : 'k -> int;
    mutable size : int;
    mutable buckets : ('k, 'v) content array;
  }

  let expand_factor = 0.5
  let shrink_factor = 0.125
  let capacity {buckets} = Array.length buckets
  let load_factor tab = float_of_int tab.size /. float_of_int (capacity tab)
  let create hash n = {hash; size = 0; buckets = Array.make n Empty}
  let index k tab = tab.hash k mod capacity tab

  let rec find_slot k tab i =
    let pos = (index k tab + i) mod capacity tab in
    match tab.buckets.(pos) with
    | Empty -> pos
    | Deleted -> find_slot k tab (i + 1)
    | Full (k', _) when k = k' -> pos
    | Full _ -> find_slot k tab (i + 1)

  let insert_no_resize k v tab =
    let pos = find_slot k tab 0 in
    match tab.buckets.(pos) with
    | Empty | Deleted ->
      tab.buckets.(pos) <- Full (k, v);
      tab.size <- tab.size + 1
    | Full (k', _) when k = k' -> tab.buckets.(pos) <- Full (k, v)
    | Full _ -> failwith "Invalid state"

  let resize tab new_cap =
```

```
    let old_buckets = tab.buckets in
    let new_tab =
      {hash = tab.hash; size = 0; buckets = Array.make new_cap Empty}
    in
    Array.iter
      (function
        | Full (k, v) -> insert_no_resize k v new_tab
        | _ -> ())
      old_buckets;
    tab.buckets <- new_tab.buckets;
    tab.size <- new_tab.size

  let insert k v tab =
    if load_factor tab > expand_factor then resize tab (2 * capacity tab);
    insert_no_resize k v tab

  let find k tab =
    let pos = find_slot k tab 0 in
    match tab.buckets.(pos) with
    | Full (_, v) -> Some v
    | _ -> None

  let remove k tab =
    let pos = find_slot k tab 0 in
    match tab.buckets.(pos) with
    | Full (k', _) when k = k' ->
      tab.buckets.(pos) <- Deleted;
      tab.size <- tab.size - 1;
      if load_factor tab < shrink_factor && capacity tab > 1 then
        resize tab (capacity tab / 2)
    | _ -> ()

  let bindings tab =
    Array.fold_left
      (fun acc content ->
        match content with
        | Full (k, v) -> (k, v) :: acc
        | _ -> acc)
      [] tab.buckets

  let of_list hash lst =
    let tab = create hash (List.length lst) in
    List.iter (fun (k, v) -> insert k v tab) lst;
    tab
end
```

Problem 8: functorized BST

Our implementation of BSTs assumed that it was okay to compare values using the built-in comparison operators $<$, $=$, and $>$. But what if the client wanted to use their own comparison operators? (e.g., to ignore case in strings, or to have sets of records where only a single field of the record was used for ordering.) Implement a `BstSet` abstraction as a functor parameterized on a structure that enables client-provided comparison operator(s), much like the standard library Set.

```
module type Set = sig
  type elt
  type t
  val empty    : t
  val insert   : elt -> t -> t
  val mem      : elt -> t -> bool
  val of_list  : elt list -> t
  val elements : t -> elt list
end

module type Ordered = sig
  type t
```

```
  val compare : t -> t -> int
end

module BstSet (Ord : Ordered) : Set with type elt = Ord.t = struct
  type elt = Ord.t

  type t = Leaf | Node of t * elt * t

  let empty = Leaf

  let rec mem x = function
    | Leaf -> false
    | Node (l, v, r) ->
      begin
        match compare x v with
        | ord when ord < 0 -> mem x l
        | ord when ord > 0 -> mem x r
        | _                -> true
      end

  let rec insert x = function
    | Leaf -> Node (Leaf, x, Leaf)
    | Node (l, v, r) ->
      begin
        match compare x v with
        | ord when ord < 0 -> Node(insert x l, v, r         )
        | ord when ord > 0 -> Node(l,          v, insert x r)
        | _                -> Node(l,          x, r         )
      end

  let of_list lst =
    List.fold_left (fun s x -> insert x s) empty lst

  let rec elements = function
    | Leaf -> []
    | Node (l, v, r) -> (elements l) @ [v] @ (elements r)
end

module IntSet = BstSet (Int)
let example_set = IntSet.(empty |> insert 1)
```

Problem 9: efficient traversal

Suppose you wanted to convert a tree to a list. You'd have to put the values stored in the tree in some order. Here are three ways of doing that:

- *preorder*: each node's value appears in the list before the values of its left then right subtrees.
- *inorder*: the values of the left subtree appear, then the value at the node, then the values of the right subtree.
- *postorder*: the values of a node's left then right subtrees appear, followed by the value at the node.

Here is code that implements those *traversals*, along with some example applications:

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

let rec preorder = function
  | Leaf -> []
  | Node (l,v,r) -> [v] @ preorder l @ preorder r

let rec inorder = function
  | Leaf -> []
  | Node (l,v,r) -> inorder l @ [v] @ inorder r

let rec postorder = function
  | Leaf -> []
  | Node (l,v,r) -> postorder l @ postorder r @ [v]
```

On unbalanced trees, the traversal functions above require quadratic worst-case time (in the number of nodes), because of the `@` operator. Re-implement the functions without `@`, and instead using `::`, such that they perform exactly one `cons` per `Node` in the tree. Thus, the worst-case execution time will be linear. You will need to add an additional accumulator argument to each function, much like with tail recursion. (But your implementations won't actually be tail recursive.)

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

let rec preorder t =
  (* [go acc t] is equivalent to [List.rev (preorder t) @ acc] *)
  let rec go acc = function
    | Leaf -> acc
    | Node (l,v,r) -> go (go (v :: acc) l) r
  in
  List.rev (go [] t)

let rec inorder t =
  (* [go acc t] is equivalent to [List.rev (inorder t) @ acc] *)
  let rec go acc = function
    | Leaf -> acc
    | Node (l,v,r) -> go (v :: go acc l) r
  in
  List.rev (go [] t)

let rec postorder t =
  (* [go acc t] is equivalent to [List.rev (postorder t) @ acc] *)
  let rec go acc = function
    | Leaf -> acc
    | Node (l,v,r) -> v :: go (go acc l) r
  in
  List.rev (go [] t)
```

Appendix. More OCaml

Problem 1: Acronym (easy)

Convert a phrase to its acronym. Techies love their TLA (Three Letter Acronyms)! Help generate some jargon by writing a program that converts a long name like Portable Network Graphics to its acronym (PNG). Punctuation is handled as follows: hyphens are word separators (like whitespace); all other punctuation can be removed from the input.

```
open Base

let acronym (phrase : string) : string =
  phrase
  |> String.tr ~target:'-' ~replacement:' '
  |> String.filter ~f:(fun c -> Char.is_alpha c || Char.is_whitespace c)
  |> String.split ~on:' '
  |> List.filter ~f:(fun s -> not (String.is_empty s))
  |> List.map ~f:(fun word -> String.get word 0)
  |> String.of_char_list
  |> String.uppercase
```

Problem 2: Allergies (easy)

Given a person's allergy score, determine whether or not they're allergic to a given item, and their full list of allergies. An allergy test produces a single numeric score which contains the information about all the allergies the person has (that they were tested for). The list of items (and their value) that were tested are:

- eggs (1)
- peanuts (2)

- shellfish (4)
- strawberries (8)
- tomatoes (16)
- chocolate (32)
- pollen (64)
- cats (128)

```
open Base

type allergen = Eggs
              | Peanuts
              | Shellfish
              | Strawberries
              | Tomatoes
              | Chocolate
              | Pollen
              | Cats

let allergen_scores = [(Eggs, 1); (Peanuts, 2); (Shellfish, 4); (Strawberries, 8); (Tomatoes
    , 16); (Chocolate, 32); (Pollen, 64); (Cats, 128)]

let allergic_to n allergen =
  let allergen_score = List.Assoc.find allergen_scores allergen ~equal:Poly.equal in
  match allergen_score with
  | Some score -> (n land score) <> 0
  | None -> false

let allergies n =
  List.filter_map allergen_scores ~f:(fun (allergen, score) ->
    if (n land score) <> 0 then Some allergen else None)
```

Problem 3: Anagram (easy)

Given a target word and one or more candidate words, your task is to find the candidates that are anagrams of the target. An anagram is a rearrangement of letters to form a new word: for example `"owns"` is an anagram of `"snow"`. A word is **not** its own anagram: for example, `"stop"` is not an anagram of `"stop"`. The target word and candidate words are made up of one or more ASCII alphabetic characters (`A-Z` and `a-z`). Lowercase and uppercase characters are equivalent: for example, `"PoTS"` is an anagram of `"sTOp"`, but `"StoP"` is not an anagram of `"sTOp"`. The words you need to find should be taken from the candidate words, using the same letter case. Given the target `"stone"` and the candidate words `"stone"`, `"tones"`, `"banana"`, `"tons"`, `"notes"`, and `"Seton"`, the anagram words you need to find are `"tones"`, `"notes"`, and `"Seton"`. You must return the anagrams in the same order as they are listed in the candidate words.

```
let normalize_word word =
  word
  |> String.to_seq
  |> List.of_seq
  |> List.sort Char.compare
  |> List.to_seq
  |> String.of_seq

let anagrams target candidates =
  let lower_target = String.lowercase_ascii target in
  let sorted_target = normalize_word lower_target in

  List.filter (fun candidate ->
    let lower_candidate = String.lowercase_ascii candidate in
    lower_candidate <> lower_target &&
    normalize_word lower_candidate = sorted_target
  ) candidates
```

Problem 4: Binary Search Tree (easy)

Implement a binary search tree.

```
type bst = Leaf | Node of int * bst * bst

let empty = Leaf

let value = function
  | Leaf -> Error "Value of empty tree"
  | Node (v, _, _) -> Ok v

let left = function
  | Leaf -> Error "Left of empty tree"
  | Node (_, l, _) -> Ok l

let right = function
  | Leaf -> Error "Right of empty tree"
  | Node (_, _, r) -> Ok r

let rec insert value tree =
  match tree with
  | Leaf -> Node (value, Leaf, Leaf)
  | Node (v, l, r) -> if value <= v then Node (v, insert value l, r) else Node (v, l, insert
      value r)

let rec to_list = function
  | Leaf -> []
  | Node (v, l, r) -> to_list l @ [v] @ to_list r
```

Problem 5: Bob (easy)

Your task is to determine what Bob will reply to someone when they say something to him or ask him a question. Bob only ever answers one of five things:

- **"Sure."** This is his response if you ask him a question, such as "How are you?" The convention used for questions is that it ends with a question mark.
- **"Whoa, chill out!"** This is his answer if you YELL AT HIM. The convention used for yelling is ALL CAPITAL LETTERS.
- **"Calm down, I know what I'm doing!"** This is what he says if you yell a question at him.
- **"Fine. Be that way!"** This is how he responds to silence. The convention used for silence is nothing, or various combinations of whitespace characters.
- **"Whatever."** This is what he answers to anything else.

```
let is_silence phrase =
  String.trim phrase = ""

let is_question phrase =
  let trimmed = String.trim phrase in
  if String.length trimmed = 0 then false
  else String.ends_with ~suffix:"?" trimmed

let has_letters phrase =
  String.exists (fun c -> (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) phrase

let is_yelling phrase =
  has_letters phrase && String.uppercase_ascii phrase = phrase

let response_for phrase =
  let question = is_question phrase in
  let yelling = is_yelling phrase in
  let silence = is_silence phrase in

  if yelling && question then
```

```
    "Calm down, I know what I'm doing!"
  else if yelling then
    "Whoa, chill out!"
  else if question then
    "Sure."
  else if silence then
    "Fine. Be that way!"
  else
    "Whatever."
```

Problem 6: Difference of Squares (easy)

Find the difference between the square of the sum and the sum of the squares of the first N natural numbers.

```
let square_of_sum n =
    let sum = List.fold_left (+) 0 (List.init n (fun x -> x + 1)) in
    sum * sum

let sum_of_squares n =
    List.fold_left (+) 0 (List.init n (fun x -> (x + 1) * (x + 1)))

let difference_of_squares n =
    abs (square_of_sum n - sum_of_squares n)
```

Problem 7: Eliud's Eggs (easy)

Your friend Eliud inherited a farm from her grandma Tigist. Her granny was an inventor and had a tendency to build things in an overly complicated manner. The chicken coop has a digital display showing an encoded number representing the positions of all eggs that could be picked up. Eliud is asking you to write a program that shows the actual number of eggs in the coop. The position information encoding is calculated as follows:

- Scan the potential egg-laying spots and mark down a `1` for an existing egg or a `0` for an empty spot.
- Convert the number from binary to decimal.
- Show the result on the display.

```
let egg_count number =
  let rec egg_count_helper n acc =
    if n = 0 then acc
    else egg_count_helper (n / 2) (acc + n mod 2)
  in
  egg_count_helper number 0
```

Problem 8: ETL (easy)

Your task is to change the data format of letters and their point values in the game. Currently, letters are stored in groups based on their score, in a one-to-many mapping.

- 1 point: "A", "E", "I", "O", "U", "L", "N", "R", "S", "T",
- 2 points: "D", "G",
- 3 points: "B", "C", "M", "P",
- 4 points: "F", "H", "V", "W", "Y",
- 5 points: "K",
- 8 points: "J", "X",
- 10 points: "Q", "Z",

This needs to be changed to store each individual letter with its score in a one-to-one mapping.

- "a" is worth 1 point.

- "b" is worth 3 points.
- "c" is worth 3 points.
- "d" is worth 2 points.
- etc.

```
let transform (legacy_data: (int * char list) list) : (char * int) list =
  let flat_list =
    List.concat_map (fun (score, letters) ->
      List.map (fun letter ->
        let lower_char = Char.lowercase_ascii letter in
        (lower_char, score)
      ) letters
    ) legacy_data
  in
  List.sort (fun (char1, _) (char2, _) ->
    Char.compare char1 char2
  ) flat_list
```

Problem 9: Grade School (easy)

Given students' names along with the grade they are in, create a roster for the school. In the end, you should be able to:

- Add a student's name to the roster for a grade:
  - "Add Jim to grade 2."
  - "OK."
- Get a list of all students enrolled in a grade:
  - "Which students are in grade 2?"
  - "We've only got Jim right now."
- Get a sorted list of all students in all grades. Grades should be sorted as 1, 2, 3, etc., and students within a grade should be sorted alphabetically by name.
  - "Who is enrolled in school right now?"
  - "Let me think. We have Anna, Barb, and Charlie in grade 1, Alex, Peter, and Zoe in grade 2, and Jim in grade 5. So the answer is: Anna, Barb, Charlie, Alex, Peter, Zoe, and Jim."

Note that all our students only have one name (it's a small town, what do you want?), and each student cannot be added more than once to a grade or the roster. If a test attempts to add the same student more than once, your implementation should indicate that this is incorrect.

```
open Base

module Int_map = Map.M(Int)
type school = string list Int_map.t

let empty_school = Map.empty (module Int)

let add name grade_num school =
  Map.change school grade_num ~f:(fun maybe_students ->
    let students = Option.value maybe_students ~default:[] in
    let new_students = name :: students in
    Some (List.sort new_students ~compare:String.compare)
  )

let grade g school =
  Map.find school g |> Option.value ~default:[]

let sorted school =
  Map.map school ~f:(List.sort ~compare:String.compare)
```

```
let roster school =
  List.concat (Map.data school)
```

Problem 10: Hamming (easy)

Calculate the Hamming distance between two DNA strands. We read DNA using the letters C, A, G and T. Two strands might look like this:

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
^ ^ ^  ^ ^    ^^
```

They have 7 differences, and therefore the Hamming distance is 7.

```
type nucleotide = A | C | G | T

let hamming_distance (seq1: nucleotide list) (seq2: nucleotide list) =
  if List.length seq1 == 0 && List.length seq2 == 0 then Ok 0
  else if List.length seq1 == 0 then Error "left strand must not be empty"
  else if List.length seq2 == 0 then Error "right strand must not be empty"
  else if List.length seq1 <> List.length seq2 then Error "left and right strands must be of
       equal length"
  else
  let rec hamming_distance_helper seq1 seq2 acc =
    match seq1, seq2 with
    | [], [] -> Ok acc
    | [], _ | _, [] -> Error "left and right strands must be of equal length"
    | x::xs, y::ys -> hamming_distance_helper xs ys (acc + if x = y then 0 else 1)
  in
  hamming_distance_helper seq1 seq2 0
```

Problem 11: Isogram (easy)

Determine if a word or phrase is an isogram. An isogram (also known as a "non-pattern word") is a word or phrase without a repeating letter, however spaces and hyphens are allowed to appear multiple times. Examples of isograms:

- lumberjacks
- background
- downstream
- six-year-old

The word `"isograms"`, however, is not an isogram, because the `s` repeats.

```
open Base

let is_isogram word =
    let rec is_isogram_helper set chars =
        match chars with
        | [] -> true
        | c :: rest when Char.is_alpha c ->
            let c_lower = Char.lowercase c in
            if Set.mem set c_lower then false
            else is_isogram_helper (Set.add set c_lower) rest
        | _ :: rest -> is_isogram_helper set rest
    in
    let chars = String.to_list word in
    is_isogram_helper (Set.empty (module Char)) chars
```

Problem 12: Leap (easy)

Determine whether a given year is a leap year.

```
let leap_year year =
    if year mod 4 = 0 then
        if year mod 100 = 0 then
            if year mod 400 = 0 then
                true
            else
                false
        else
            true
    else
        false
```

Problem 13: nucleotide Count (easy)

Given a string representing a DNA sequence, count how many of each nucleotide is present. If the string contains characters that aren't A, C, G, or T then it is invalid and you should signal an error. For example:

```
"GATTACA" -> 'A': 3, 'C': 1, 'G': 1, 'T': 2
"INVALID" -> error
```

```
open Base

let empty = Map.empty (module Char)

let dna_set = Set.of_list (module Char) ['A'; 'C'; 'G'; 'T']

let count_nucleotide s c =
  if Set.mem dna_set c then
    let invalid_char = String.to_list s |> List.find ~f:(fun ch -> not (Set.mem dna_set ch))
        in
    match invalid_char with
    | Some invalid -> Error invalid
    | None -> Ok (String.count s ~f:(Char.equal c))
  else
    Error c

let count_nucleotides s =
  let rec count_nucleotides_helper s acc =
    match s with
    | [] -> Ok acc
    | c::cs -> begin
      if Set.mem dna_set c then
        let current_count = Map.find acc c |> Option.value ~default:0 in
        count_nucleotides_helper cs (Map.set acc ~key:c ~data:(current_count + 1))
        else
          Error c
    end
  in
  count_nucleotides_helper (String.to_list s) empty
```

Problem 14: Pangram (easy)

Your task is to figure out if a sentence is a pangram. A pangram is a sentence using every letter of the alphabet at least once. It is case insensitive, so it doesn't matter if a letter is lower-case (e.g. `k`) or upper-case (e.g. `K`). For this exercise, a sentence is a pangram if it contains each of the 26 letters in the English alphabet.

```
open Base

let is_pangram sentence =
  sentence
```

```
  |> String.lowercase
  |> String.to_list
  |> List.filter ~f:Char.is_alpha
  |> List.dedup_and_sort ~compare:Char.compare
  |> List.length
  |> (=) 26
```

Problem 15: Raindrops (easy)

Your task is to convert a number into its corresponding raindrop sounds. If a given number:

- is divisible by 3, add "Pling" to the result.
- is divisible by 5, add "Plang" to the result.
- is divisible by 7, add "Plong" to the result.
- **is not** divisible by 3, 5, or 7, the result should be the number as a string.

```
let raindrop n =
  let res = ref "" in
  if n mod 3 = 0 then res := !res ^ "Pling";
  if n mod 5 = 0 then res := !res ^ "Plang";
  if n mod 7 = 0 then res := !res ^ "Plong";
  if !res = "" then res := !res ^ string_of_int n;
  !res
```

Problem 16: Reverse String (easy)

Reverse a given string.

```
open Base

let reverse_string s =
  String.to_list s
  |> List.rev
  |> String.of_char_list
```

Problem 17: RNA Transcription (easy)

Determine the RNA complement of a given DNA sequence.

```
type dna = [ `A | `C | `G | `T ]
type rna = [ `A | `C | `G | `U ]

let to_rna seq =
  List.map (function
    | `A -> `U
    | `C -> `G
    | `G -> `C
    | `T -> `A
  ) seq
```

Problem 18: Sieve (easy)

Your task is to create a program that implements the Sieve of Eratosthenes algorithm to find all prime numbers less than or equal to a given number.

```
let primes limit =
  if limit < 2 then []
  else
    let sieve = Array.make (limit + 1) true in
    sieve.(0) <- false;
    sieve.(1) <- false;
```

```
    for p = 2 to int_of_float (sqrt (float_of_int limit)) do
      if sieve.(p) then
        let i = ref (p * p) in
        while !i <= limit do
          sieve.(!i) <- false;
          i := !i + p
        done
    done;

    let result = ref [] in
    for i = limit downto 2 do
      if sieve.(i) then
        result := i :: !result
    done;
    !result
```

Problem 19: Space Age (easy)

Given an age in seconds, calculate how old someone would be on a planet in our Solar System. One Earth year equals 365.25 Earth days, or 31,557,600 seconds. If you were told someone was 1,000,000,000 seconds old, their age would be 31.69 Earth-years. For the other planets, you have to account for their orbital period in Earth Years:

Planet	Orbital period in Earth Years
Mercury	0.2408467
Venus	0.61519726
Earth	1.0
Mars	1.8808158
Jupiter	11.862615
Saturn	29.447498
Uranus	84.016846
Neptune	164.79132

```
type planet = Mercury | Venus | Earth | Mars
            | Jupiter | Saturn | Neptune | Uranus

let orbital_period_in_earth_years = function
  | Mercury -> 0.2408467
  | Venus   -> 0.61519726
  | Earth   -> 1.0
  | Mars    -> 1.8808158
  | Jupiter -> 11.862615
  | Saturn  -> 29.447498
  | Uranus  -> 84.016846
  | Neptune -> 164.79132

let age_on (p : planet) (seconds : int) : float =
  let earth_year_in_seconds = 31_557_600.0 in

  let age_in_earth_years = float_of_int seconds /. earth_year_in_seconds in

  let planet_orbital_period = orbital_period_in_earth_years p in

  age_in_earth_years /. planet_orbital_period
```

Problem 20: Triangle (easy)

Determine if a triangle is equilateral, isosceles, or scalene.

- An `equilateral` triangle has all three sides the same length.
- An `isosceles` triangle has at least two sides the same length.

- A `scalene` triangle has all sides of different lengths.

```
let is_triangle a b c =
  a + b > c && a + c > b && b + c > a

let is_equilateral a b c =
  is_triangle a b c && a = b && b = c

let is_isosceles a b c =
  is_triangle a b c && (a = b || b = c || a = c)

let is_scalene a b c =
  is_triangle a b c && a <> b && b <> c && a <> c
```