Contents

1 Learning OCaml - Notes

My notes file for learning OCaml and for working through the CS3110 book.

A warning for readers of the pdf version of this document: Some code that runs fine in org-babel will suddenly produce errors when emacs re-runs the source blocks during the `org-export-to-latex` run. I think this is because during export, each block is run independently. So if one block defines `t` and another block calls a function with `t` as an argument, the second block will complain that `t` is undefined. These errors are not present in the notes.org document, they only appear in the pdf. At the moment it is not worth my time to troubleshoot this issue.

1.1 Goals for this document

- Learning OCaml
- Working through the CS3110 book.
- Learn something about literate programming in org-mode

1.2 Remarks on OCaml in org-mode

You can use `C-c C-,` to open the export block dialog, select `s` for source and type ocaml. lets you type ocaml code with syntax highlighting and indentation from `merlin` in the org buffer. In a source block, `C-c C-c` runs the code and puts the results of evaluation underneath the source block.

In the event these source blocks aren't adequate, `M-x tuareg-run-ocaml` opens the actual ocaml toplevel that emacs is running in the background.

The following variables control something about the way code gets passed to and retrieved from the ocaml toplevel that emacs runs in the background. If you end up actually opening and using the toplevel directly in order to check or debug code, it can end up looking a little cluttered with this expression repeated all over the place, so it might be worth it to change it to a smaller or simpler expression in that case. I haven't dound this to be necessary.

```
(setq org-babel-ocaml-eoe-output "org-babel-ocaml-eoe")
(setq org-babel-ocaml-eoe-indicator "\"org-babel-ocaml-eoe\";;")
```

The default behavior of source blocks may not be adequate for printing results. For example the following source block shows its result, but does not show that the result has type `int`

```
1 + 2;;
```

```
- : int = 3
```

But if we pass the header argument to insist that the result is displayed verbatim, then we see the type as well.

```
let x = 42;;
x
```

```
- : int = 42
```

This can also be an issue with multi-line output. The default seems to be that only the last line of toplevel output makes it back to the org buffer, so this source block would juss show `unit = ()`. Again, the verbatim tag fixes this. `:results output` is another option, similar to `verbatim`.

```
print_string "hello\n"
```

```
hello
- : unit = ()
```

For the purpose of exporting the entire org file to pdf by way of latex, we need to do a bit of extra work. We want to wrap source blocks in the `minted` latex environment so they can be colored and syntax-highlighted by `pygments`. For visual clarity in the final pdf, we also want the output to be ocaml source blocks. But these output blocks should not be evaluated.

So we want source blocks with `export` set to `both`, with `results` set to `verbatim` and `wrapped` in an ocaml source block, and where the results blocks have their `export` set to just `code`. The best way I've found to do this is by setting

```
#+PROPERTY: header-args:ocaml :exports both :results verbatim :wrap "src ocaml :exports code"
```

at the top of the org file. With this option set, ordinary `ocaml` source blocks with no other header args will work the way I want them to, both for literate programming in org and for my export configuration. Additionally, since so many ocaml functions have underscores in their names, we want to set `#+OPTIONS: ^:nil` at the top of the file. Otherwise tex will interpret these as subscripts.

1.2.1 TODO org latex export

These is still something strange going on with org's export. If I export from notes.org in an empty directory, source blocks don't make it to the final pdf document, though they look right in the latex source. Then if I run `pdflatex -shell-escape notes.tex` on my own, a pdf is produced that does show the source (and output) blocks correctly. After that, org export will start working as expected. I don't know why this is happening but for now it's not worth the headache to fix it. Exporting the pdf shouldn't happen often enough for it to be a real problem.

Also, need to change this variable to `nil`

```
(setq org-confirm-babel-evaluate nil)
```

before exporting in order to avoid needing to type "yes" for every single source block. Probably should try to incorporate this into emacs configuration, but in a safe way.

The following line should also be run before exporting to tex. Without it, all the source blocks will be re-evaluated before export. But it seems as if they're each re-evaluated in a fresh toplevel session. This means if one source block defines `t` and another source block calls a function with `t` as an argument, the second source block will throw an error, since it doesn't know what `t` is. Setting this variable to `nil` means source blocks don't get re-evaluated: they are exported as-is. If there's a corresponding export block from running `C-c C-c` manually, it will be exported as-is.

```
(setq org-export-use-babel nil)
```

What I would like is a way to re-run all the source blocks in order and have the results blocks written (or re-written) to the buffer. But I don't know how to do this while not running the results blocks themselves. The results blocks don't contain valid ocaml expressions, so they should not be run.

1.3 CS3110 - Notes

1.4 CS3110 Exercises [80/204] [39%]

1.4.1 2.9 Basics - Exercises [16/16]

1. **DONE** Values [⋆]

   What is the type and value of each of the following OCaml expressions:

   - `7 * (1 + 2 + 3)`

   This is `42`, an `int`.

   ```
   7 * (1 + 2 + 3)
   ```

   ```
   - : int = 42
   ```

   - `"CS " ^ string_of_int 3110`

   This is `"CS 3110"` , a `string`.

   ```
   "CS " ^ string_of_int 3110
   ```

   ```
   - : string = "CS 3110"
   ```

2. **DONE** Operators [⋆⋆]

   - Write an expression that multiplies 42 by 10

   ```
   42 * 10
   ```

```
- : int = 420
```

- Write an expression that divides 3.14 by 2.0

```
3.14 /. 2.0
```

```
- : float = 1.57
```

- Write an expression that computes 4.2 raised to the 7th power

```
let rec pow a b = match b with
  | 0 -> 1.0
  | b -> a *. pow a (b-1) in
    pow 4.2 7
```

```
- : float = 23053.9333248000075
```

3. **DONE** Equality [⋆]

- Write an expression that compares `42~` to `42` using structural equality

Structural equality is compared with `=` (or `<>` for inequality)

```
42 = 42
```

```
- : bool = true
```

- Write an expression that compares `"hi"` to `"hi"` using structural equality. What is the result?

```
"hi" = "hi"
```

```
- : bool = true
```

- Write an expression that compares `"hi"` to `"hi"` using physical equality. What is the result?

Physical equality is compared with `==` and `!=`.

```
"hi" == "hi"
```

```
- : bool = false
```

structural equality is closer to the mathematical notion of equality, but physical equality is closer to "are these the same object in memory?". Seems like for my purposes it's usually correct to use `=`.

4. **DONE** Assertions [⋆]

- Enter `assert true;;` into utop and see what happens.

`assert true;;` seems to do "nothing" with type unit.

- Enter `assert false;;` into utop and see what happens.

Assert false throws an exception, `Assert_failure`

- Write an expression that asserts 2110 is not (structurally) equal to 3110.

```
assert (2110 <> 3110);;
```

```
- : unit = ()
```

5. **DONE** If [⋆]

   Write an if expression that evaluates to 42 if 2 is greater than 1 and otherwise evaluates to 7.

```
if 2 > 1 then 42 else 7;;
```

```
- : int = 42
```

6. **DONE** Double fun [⋆]

   Using the increment function from above as a guide, define a function double that multiplies its input by 2. For example, double 7 would be 14. Test your function by applying it to a few inputs. Turn those test cases into assertions.

```
let double x = 2 * x;;
```

```
val double : int -> int = <fun>
```

   To test it, double some small integers.

```
List.map double [-1;0;1;2;3]
```

```
- : int list = [-2; 0; 2; 4; 6]
```

   Using assertions:

```
assert (double 0 = 0);;
assert (double 10 = 20);;
assert (double 50 = 100);;
assert (double 2 = 4);;
assert (double 3 <> 5);;
```

```
- : unit = ()
```

7. **DONE** More fun [⋆⋆]

- Define a function that computes the cube of a floating-point number. Test your function by applying it to a few inputs.

```
let cube x = x *. x *. x;;
```

```
val cube : float -> float = <fun>
```

Test on some small floats

```
List.map cube [-1.; 0.0; 1.; 1.5; 2.]
```

```
- : float list = [-1.; 0.; 1.; 3.375; 8.]
```

- Define a function that computes the sign (1, 0, or -1) of an integer. Use a nested if expression. Test your function by applying it to a few inputs.

As much as I'd prefer to use a `match` expression, they said use nested `if` expressions:

```
let sign x = if x < 0
             then -1
             else (if x > 0
                   then 1
                   else 0)
```

```
val sign : int -> int = <fun>
```

Test a little:

```
List.map sign [-2;-1;0;1;2;3]
```

```
- : int list = [-1; -1; 0; 1; 1; 1]
```

- Define a function that computes the area of a circle given its radius. Test your function with assert.

```
let area r =
  let pi = Float.pi in
  pi *. r *. r;;
```

```
val area : float -> float = <fun>
```

Quick `assert` test. Could do more.

```
assert (area 1.0 -. Float.pi < 1e-5)
```

```
- : unit = ()
```

8. **DONE** RMS [⋆⋆]

Define a function that computes the root mean square of two numbers—i.e.

$$\sqrt{x^2 + y^2}$$

Test your function with assert.

```
let rms x y = Float.sqrt(x *. x +. y *. y);;
```

```
val rms : float -> float -> float = <fun>
```

Testing it by generating Pythagorean triples:

```
let rmstest s t =
  let a = 2. *. s *. t in
  let b = s *. s -. t *. t in
  let c = s *. s +. t *. t in
  assert (rms a b -. c < 1e-8);;
```

```
val rmstest : float -> float -> unit = <fun>
```

```
[rmstest 10. 21.; rmstest 1000. 3201.;]
```

```
- : unit list = [(); ()]
```

9. **DONE** 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`. And 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.

(it's not clear to me why this is a "three star" exercise. Am I supposed to to this with a hash table or something? Six lines is fewer than 12, but is this not terse enough?)

```
let valid_date d m =
  match d with
  | "Feb" -> m <= 28
  | "Sept" | "Apr" | "Jun" | "Nov" -> m <= 30
  | "Jan" | "Mar" | "May" | "Jul" | "Aug" | "Oct" | "Dec" -> m <= 31
  | _ -> false;;
```

```
val valid_date : string -> int -> bool = <fun>
```

Little test

```
valid_date "Apr" 20
```

```
- : bool = true
```

10. **DONE** 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`, … That is

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

    ```
    let rec fib n = match n with
      | 1 | 2 -> 1
      | n -> fib (n-1) + fib (n-2);;
    ```

    ```
    val fib : int -> int = <fun>
    ```

    Test small values:

    ```
    List.map fib [1;2;3;4;5;6;7;8;9;10]
    ```

    ```
    - : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55]
    ```

    Looks right to me.

11. **DONE** fib fast [⋆ ⋆ ⋆]

    How quickly does your implementation of fib compute the 50th Fibonacci number? If it computes nearly instantaneously, congratulations! But the recursive solution most people come up with at first will seem to hang indefinitely. The problem is that the obvious solution computes subproblems repeatedly. For example, computing fib 5 requires computing both fib 3 and fib 4, and if those are computed separately, a lot of work (an exponential amount, in fact) is being redone.

    Here's my code to time the computation of `fib n`.

    ```
    let fibtimer n =
      let t1 = Sys.time() in
      let fn = fib n in
      let t2 = Sys.time() in
      let output = "found fib "
                   ^ (string_of_int n)
                   ^ " = "
                   ^ (string_of_int fn)
                   ^ " in "
                   ^ (string_of_float (t2 -. t1))
                   ^ " seconds." in
      print_endline output;;
    ```

```
val fibtimer : int -> unit = <fun>
```

Running `fibtimer 50;;` will print `found fib 50 = 12586269025 in 257.446328 seconds`. So indeed, it's Slow.

Here's a faster version (can probably do slighly better by writing the linear recurrence as the product of a power of a 2×2 matrix times a vector, but thats' a lot of work for minimal gain.

```
let fib_fast n =
  let rec fib_aux a b n = match n with
    | 1 -> a
    | n -> fib_aux b (a+b) (n-1) in
  fib_aux 1 1 n;;
```

```
val fib_fast : int -> int = <fun>
```

Again, here's a time:

```
let fibfasttimer n =
  let t1 = Sys.time() in
  let fn = fib_fast n in
  let t2 = Sys.time() in
  let output = "found fib_fast "
               ^ (string_of_int n)
               ^ " = "
               ^ (string_of_int fn)
               ^ " in "
               ^ (string_of_float (t2 -. t1))
               ^ " seconds." in
  print_endline output;;
```

```
val fibfasttimer : int -> unit = <fun>
```

Now, running `fibfasttimer 50` will print `found fib_fast 50 = 12586269025 in 4.99999998738e-06 seconds.`, which is much faster.

What is the first value of n for which `fib_fast n` is negative, indicating that integer overflow occurred?

```
let first_overflow =
  let rec next_neg_fib n =
    if (fib_fast n < 0) then (n) else (next_neg_fib (n+1)) in
  next_neg_fib 1
```

```
val first_overflow : int = 91
```

12. **DONE** poly types [⋆⋆⋆]

    What is the type of each of the functions below? You can ask the toplevel to check your answers

- `let f x = if x then x else x`

Since `x` is being passed as the first argument to the ternary if-then-else, `x` has to have type `bool`. Since the output is always `x`, the output of `f` will have type `bool`. So `f` is a function `bool -> bool`.

```
let f x = if x then x else x
```

```
val f : bool -> bool = <fun>
```

- `let g x y = if y then x else x`

Here, `y` needs to have type `bool`. But `x` can have arbitrary type `T`. The output of the function will have the same type as `x` (in fact, the output will be `x`), so `g` is a function that takes an argument of type `T` and an argument of type bool and returns an output of type `T`. i.e. `g: T -> bool -> T`. Ocaml uses `'a` for this type variable.

```
let g x y = if y then x else x
```

```
val g : 'a -> bool -> 'a = <fun>
```

- `let h x y z = if x then y else z`

Again, `x` needs to have type `bool`. Since the `then` and `else` branches needs to have the same output type, `y` and `z` need to have the same arbitrary type `T`. So `h : bool -> T -> T -> T`

```
let h x y z = if x then y else z
```

```
val h : bool -> 'a -> 'a -> 'a = <fun>
```

- `let i x y z = if x then y else y`

`let i x y z = if x then y else y`: Here, `x` need to have type `bool`. `y` can have arbitrary type `T1`, and `z` can have arbitrary type `T2`. The output is always `y`, which will have type `T1`. So `i: bool -> T1 -> T2 -> T1`. OCaml will use `'a` and `'b` to represent these two arbitrary types.

```
let i x y z = if x then y else y
```

```
val i : bool -> 'a -> 'b -> 'a = <fun>
```

13. **DONE** Divide [⋆⋆]

Write a function `divide : numerator:float -> denominator:float ->float`. Apply your function.

```
let divide num denom =
  let q = num /. denom in
  match q with
  | q when q = infinity -> raise Division_by_zero
  | q when q = neg_infinity -> raise Division_by_zero
  | q when compare q nan = 0 -> raise Division_by_zero
  | q -> q;;
```

```
val divide : float -> float -> float = <fun>
```

(weirdly, `nan = nan` is false, so you need to use `compare` in that case)

```
[divide 1.0 2.0; divide 1.0 4.0; divide 10.0 5.0]
```

```
- : float list = [0.5; 0.25; 2.]
```

14. **DONE** Associativity [★★]

Suppose that we have defined `let add x y = x + y`. Which of the following produces an integer, which produces a function, and which produces an error? Decide on an answer, then check your answer in the toplevel.

```
let add x y = x + y
```

```
val add : int -> int -> int = <fun>
```

- `add 5 1`

This is `add` applied to two arguments. It evaluates to `~5+1 = 6`.

```
add 5 1
```

```
- : int = 6
```

- `add 5`

This is `add` applied to one argument. It is the "add five" function, with type `int -> int`.

```
add 5
```

```
- : int -> int = <fun>
```

- `(add 5) 1`

This is the "add five" function, appled to `1`. It evaluates to `5+1 = 6`.

```
(add 5) 1
```

```
- : int = 6
```

- `add (5 1)`

This is a syntax error. `add` is expecting a space-delimited list of two or fewer integers. The token `(5 1)` doesn't fit the bill. In fact, just `(5 1)` by itself will produce an error, since `5` is not a function, so it can't be applied to `1`.

15. **DONE** 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.`

```
let (+/.) a b = (a +. b) /. 2.;;
```

```
val ( +/. ) : float -> float -> float = <fun>
```

```
[1.0 +/. 2.0; 0. +/. 0.; 100. +/. 50.]
```

```
- : float list = [1.5; 0.; 75.]
```

16. **DONE** Hello World [⋆]

    Type the following in utop, and notice the difference in output from each:

    - `print_endline "Hello world!";;`

    This prints the given string, with a carriage return at the end. It has type `unit`.

```
Hello world!
- : unit = ()
```

    - `print_string "Hello world!";;`

    Prints the string with no newline. Also has type unit. The output looks like this:

```
Hello world!- : unit = ()
```

1.4.2 3.14 Data and Types - Exercises [30/32]

1. **DONE** List Expressions [⋆]

    - Construct a list that has the integers 1 through 5 in it. Use the square bracket notation for lists.

```
let l1 = [1;2;3;4;5];;
```

```
val l1 : int list = [1; 2; 3; 4; 5]
```

    - Construct the same list, but do not use the square bracket notation. Instead use `::` and `[]`.

```
let l2 = 1::2::3::4::5::[];;
```

```
val l2 : int list = [1; 2; 3; 4; 5]
```

    - 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 l3 = [1] @ [2;3;4] @ [5];;
```

```
val l3 : int list = [1; 2; 3; 4; 5]
```

2. **DONE** Product [★★]

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

```
let list_product l =
  let rec list_product_acc p l = match l with
    | [] -> p
    | x :: xs -> list_product_acc (p*x) xs in
  list_product_acc 1 l;;
```

```
val list_product : int list -> int = <fun>
```

   Small test

```
list_product [1;2;3;4;5;6]
```

```
- : int = 720
```

3. **DONE** 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 list_concat l =
  let rec list_concat_acc s l = match l with
    | [] -> s
    | x :: xs -> list_concat_acc (s^x) xs in
  list_concat_acc "" l;;
```

```
val list_concat : string list -> string = <fun>
```

   Small test

```
list_concat ["Hel"; "lo"; ","; " ";"world";"!"]
```

```
- : string = "Hello, world!"
```

4. **DONE** product test [★★]

   I had trouble following the instructions in the CS3110 book. Following section 3.3.1, In a new directory, I created a file `sum.ml` containing

```
let rec sum = function
  | [] -> 0
  | x :: xs -> x + sum xs
```

A file `test.ml` containing

```
open OUnit2
open Sum

let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]

let _ = run_test_tt_main tests
```

and a file `dune` containing

```
(executable
 (name test)
 (libraries ounit2))
```

Now, running `dune build test.exe` throws an error: "Error: I cannot find the root of the current workspace/project." There was also a lot of complaining about the lack of a `dune-project` file. I followed dune's suggestion to create one via `dune init proj sum`, but the complaints about the root continued. Doing `dune build test.exe --root .` seemed to work. It complained about not finding `ounit2`, but after doing `opam install ounit2`, that complaint went away. Still, my feeling is that I'm not doing this right. Probably the best thing to do is learn how to start the whole project through dune, put the code to be tested and the tests in the correct locations, and do things that way.

But at this point it does seem like `dune build test.exe --root .` succeeds (with a persistent warning about the lack of a `dune-project` file), and then `dune exec ./test.exe --root .` runs the tests. Dune says:

```
Ran: 3 tests in: 0.11 seconds.
OK
```

I'd like to know how to start from an empty directory, and do `dune init proj <name>` to create an entire new dune project. Then fill that project with the relevant code to be tested, the relevant tests, and run those tests all within dune. But I can't seem to make that work. Dune's documentation is just a little too sparse for me to figure it out on my own.

I think the lack of a `dune-project` file can also be fixed by creating an approrpiate `dune-project file`. I seem to have a workflow that works and "fixes" (suppresses) the above errors and warnings, and for purposes of reproducibility, I'll try to make it clear what I did for this problem.

In a new directory (`/standalone/product test` directory), create the following files:

The `product` function to be tested is in the file `product.ml`

```
let product lst =
  let rec product_acc p l = match l with
    | [] -> p
    | x :: xs -> product_acc (x * p) xs in
  product_acc 1 lst
```

The test suite is in `test.ml`

```
open OUnit2
open Product

let tests = "test suite for product" >::: [
  "empty" >:: (fun _ -> assert_equal 1 (product []));
  "singleton one" >:: (fun _ -> assert_equal 1 (product [1]));
  "singleton five" >:: (fun _ -> assert_equal 5 (product [5]));
  "two_elements_both_one" >:: (fun _ -> assert_equal 1 (product [1; 1]));
  "two_elements_one_one" >:: (fun _ -> assert_equal 3 (product [1; 3]));
  "two_elements_neither_one" >:: (fun _ -> assert_equal 10 (product [5; 2]));
  "three_elements" >:: (fun _ -> assert_equal 30 (product [2; 3; 5]));
  "six_elements" >:: (fun _ -> assert_equal 720 (product [1;2;3;4;5;6]));
]

let _ = run_test_tt_main tests
```

There's also a `dune` file:

```
(executable
 (name test)
 (libraries ounit2))
```

And a `dune-project` file, containing:

```
(lang dune 1.1)
(name product)
```

(Is this what `dune` needs in order to know where the root of the current project is? It seems like this is the change that got rid of that error / warning).

Now, we can run `dune build test.exe`, followed by `dune exec test.exe`. This gives:

```
........
Ran: 8 tests in: 0.11 seconds.
OK
```

It is still not clear to me that this is the "right" way to do this. But it's close enough to the process outlined in section 3.3.1 in the book that I think I'll stick with this for now. I'd still like to learn how to use `dune` properly, but I'll postpone that until later.

5. **DONE** 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"

```
let bigred l = match l with
  | "bigred" :: xs -> true
  | _ -> false;;
```

```
val bigred : string list -> bool = <fun>
```

```
[bigred ["smallred"];
 bigred ["bigred";"x";"y";"z"]]
```

```
- : bool list = [false; true]
```

(I'm not sure how to make this polymorphic: if the first element is an integer, I get a type error. But it's not clear from the phrasing of the problem if that's necessary)

- the list has exactly two or four elements; do not use the length function

```
let two_or_four l = match l with
  | x::y::[] -> true
  | x::y::z::w::[] -> true
  | _ -> false;;
```

```
val two_or_four : 'a list -> bool = <fun>
```

A few tests:

```
[two_or_four [1;2;3;4];
 two_or_four ["a";"b"];
 two_or_four [1];
 two_or_four []]
```

```
- : bool list = [true; true; false; false]
```

- the first two elements of the list are equal

```
let first_two_equal l = match l with
  | x::y::xs when x = y -> true
  | _ -> false;;
```

```
val first_two_equal : 'a list -> bool = <fun>
```

```
[first_two_equal [1;2;3];
 first_two_equal [[1];[1];[1;2]];
 first_two_equal [[];[];[1;2]];
 first_two_equal ([[]]::[[]]::[]);
 first_two_equal ["a"]]
```

```
- : bool list = [false; true; true; true; false]
```

6. **DONE** 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`.

```
let fifth_element l =
  if (List.length l >= 5) then (List.nth l 4) else (0);;
```

```
val fifth_element : int list -> int = <fun>
```

   - 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 descending_sort lst =
  lst
  |> List.sort Stdlib.compare
  |> List.rev;;
```

```
val descending_sort : 'a list -> 'a list = <fun>
```

```
descending_sort [9;3;8;2;7;6;1;2;5;5]
```

```
- : int list = [9; 8; 7; 6; 5; 5; 3; 2; 2; 1]
```

```
descending_sort ["mercury";
                 "venus";
                 "earth";
                 "mars";
                 "jupiter";
                 "saturn";
                 "uranus";
                 "neptune";
                 "pluto"]
```

```
- : string list =
["venus"; "uranus"; "saturn"; "pluto"; "neptune"; "mercury"; "mars";
 "jupiter"; "earth"]
```

7. **DONE** Library Test [⋆⋆⋆]

   Write a couple OUnit unit tests for each of the functions you wrote in the previous exercise

   Again, code is in the standalone directory.

   The functions to be tested are in `library.ml`, which contains

```
let fifth_element l =
  if (List.length l >= 5) then (List.nth l 4) else (0)

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

   Then we also need a dune file

```
(executable
 (name test)
 (libraries ounit2))
```

   as well as a dune-project file, it seems

```
(lang dune 1.1)
(name library)
```

   Finally, the test file, which contains:

```
open OUnit2
open Library

let tests = "test suite for these two functions" >::: [
  "empty list" >:: (fun _ -> assert_equal 0 (fifth_element []));
  "short list" >:: (fun _ -> assert_equal 0 (fifth_element [1;2;3]));
  "five elts" >:: (fun _ -> assert_equal 5 (fifth_element [1;2;3;4;5]));
  "repeat elts" >:: (fun _ -> assert_equal 4 (fifth_element [4;4;4;4;4;4;4]));
  "fifth zero" >:: (fun _ -> assert_equal 0 (fifth_element [1;2;3;4;0]));

  "empty sort" >:: (fun _ -> assert_equal [] (descending_sort []));
  "singleton sort" >:: (fun _ -> assert_equal [10] (descending_sort [10]));
  "pre-sorted" >:: (fun _ -> assert_equal [3;2;1] (descending_sort [3;2;1]));
  "reverse sort" >:: (fun _ -> assert_equal [5;4;3;2;1] (descending_sort
  ↪  [1;2;3;4;5]));
  "bigger sort" >:: (fun _ -> assert_equal [10;9;8;7;6;6;6;5] (descending_sort
  ↪  [5;6;10;9;6;6;7;8]));
]

let _ = run_test_tt_main tests
```

   Now doing `dune build test.exe` followed by `dune exec ./test.exe` gives

```
..........
Ran: 10 tests in: 0.11 seconds.
OK
```

8. **DONE** 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.

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

```
val last_element : 'a list -> 'a = <fun>
```

Small test:

```
last_element [1;4;3;2;3;7];;
```

```
- : int = 7
```

   - Write a function `any_zeroes : 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.

```
let any_zeroes l = List.exists ((=) 0) l;;
```

```
val any_zeroes : int list -> bool = <fun>
```

A few tests

```
[any_zeroes [1;2;3;4;10];
 any_zeroes [1;2;3;-1;-2;-10];
 any_zeroes [];
 any_zeroes [1;1;1;1;0;1;1;2;2;3;3;4];
 any_zeroes [0]]
```

```
- : bool list = [false; false; false; true; true]
```

9. **DONE** 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.

```
let rec take n l = match n with
  | 0 -> []
  | n -> (match l with
          | x :: xs -> x::(take (n-1) xs)
          | [] -> []);;
```

```
val take : int -> 'a list -> 'a list = <fun>
```

Small tests:

```
[take 2 [5;4;3;2;1];
 take 3 [1;2];
 take 0 [1;2];
 take 0 [];
 take 4 [3;2;1;2;3]]
```

```
- : int list list = [[5; 4]; [1; 2]; []; []; [3; 2; 1; 2]]
```

- 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 drop n l = match n with
| 0 -> l
| n -> (match l with
        | x :: xs -> drop (n-1) xs
        | [] -> []);;
```

```
val drop : int -> 'a list -> 'a list = <fun>
```

Small tests:

```
[drop 3 [1;2;3;4;5;6;7;8];
 drop 2 [1];
 drop 3 [5;4;4];
 drop 0 [1;2;3]]
```

```
- : int list list = [[4; 5; 6; 7; 8]; []; []; [1; 2; 3]]
```

10. **DONE** 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.

Here's the `--` operator:

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

let ( -- ) i j = from i j [];;
```

```
val ( -- ) : int -> int -> int list = <fun>
```

Here's a long list (output suppressed)

```
let long_list = 0 -- 1_000_000;;
```

Here's a tail-recursive `take` function:

```
let take n l =
  let rec take_tr n l h = match n with
    | 0 -> h
    | n -> (match l with
            | [] -> h
            | x :: xs -> take_tr (n-1) (xs) (x :: h)) in
  List.rev (take_tr n l []);;
```

```
val take : int -> 'a list -> 'a list = <fun>
```

I am not sure whether I absolutely needed to use `List.rev` here. That seems like a cost that should be avoided, if possible. It also means I'm not 100% sure this is tail recursive unless I check whether or not `List.rev` is tail recursive. The documentation doesn't say whether it is or isn't. In any case, here's the kind of call that would probably stack overflow if the function weren't tail-recursive:

```
List.length (take 2000000 (6 -- 4000000))
```

```
- : int = 2000000
```

Now for a tail-recursive drop function:

```
let rec drop n l =
  match n with
  | 0 -> l
  | n -> (match l with
          | [] -> []
          | x :: xs -> drop (n-1) xs);;
```

```
val drop : int -> 'a list -> 'a list = <fun>
```

And a call that would likely overflow the stack if it isn't tail recursive:

```
drop 999999 (1 -- 1000000);;
```

```
- : int list = [1000000]
```

It's not completely clear how to check whether or not something is tail recursive. It seems like the givaway is when the recursive call is part of a bigger expression instead of just the recursive function being called on its own with modified arguments. The alternative is just to test the kind of input that would probably overflow for a non-tail-recursive function, though that seems iffy.

11. **DONE** 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 rec is_unimodal l =
  let rec is_nonincreasing l = match l with
  | [] -> true
  | x :: [] -> true
  | a :: b :: tail -> if (a < b)
                      then (false)
                      else (is_nonincreasing (b :: tail)) in
  match l with
  | [] -> true
  | x :: [] -> true
  | a :: b :: [] -> true
  | a :: b :: tail -> if (a <= b)
                      then (is_unimodal (b :: tail))
                      else (is_nonincreasing (b :: tail));;
```

```
val is_unimodal : 'a list -> bool = <fun>
```

Some tests, with comments on the expected `false` outputs. Note the polymorphism.

```
[is_unimodal [1;2;2;2;3;3;2;2];
is_unimodal [1;2;3;4;4;4;5];
is_unimodal [6;5;4;3;2;1];
is_unimodal [1;2;3;3;2;1;2]; (* false *)
is_unimodal [1;1;1;1;1];
is_unimodal [0;0;0;0;0;0;0;0;1];
is_unimodal [1;0;0;0;0;0;0;0;0;0];
is_unimodal [4];
is_unimodal [2;1;2]; (* false *)
is_unimodal ['a';'b';'c';'b';'a'];
is_unimodal ['b';'a';'a';'b']] (* false*)
```

```
- : bool list =
[true; true; true; false; true; true; true; true; false; true; false]
```

12. **DONE** Power set [⋆⋆⋆]

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 lst = match lst with
  | [] -> [[]]
  | x :: xs -> let p = powerset xs in
              (List.map (fun s -> x::s) p) @ p;;
```

```
val powerset : 'a list -> 'a list list = <fun>
```

One small test

```
powerset [1;2;3]
```

```
- : int list list = [[1; 2; 3]; [1; 2]; [1; 3]; [1]; [2; 3]; [2]; [3]; []]
```

A slightly larger, though less precise test

```
List.length (powerset [1;2;3;4;5;6;7])
```

```
- : int = 128
```

13. **DONE** Print int list rec [⋆⋆]

    Write a function `print_int_list : int list -> unit` that prints its input list, one number per line. For example, `print_int_list [1; 2; 3]` should result in this output:

    ```
    1
    2
    3
    ```

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

    ```
    val print_int_list : int list -> unit = <fun>
    ```

    As expected:

    ```
    print_int_list [1;2;3]
    ```

    ```
    1
    2
    3
    - : unit = ()
    ```

14. **DONE** 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 e -> e |> string_of_int |> print_endline) lst;;
```

```
val print_int_list : int list -> unit = <fun>
```

Once again, as expected:

```
print_int_list [1;2;3];;
```

```
1
2
3
- : unit = ()
```

15. **DONE** Student [⋆⋆]

    Assume the following type definition:

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

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

Give OCaml expressions that have the following types:

- `student`

```
let s = {first_name = "John";
         last_name = "Smith";
         gpa = 3.9}
```

```
val s : student = {first_name = "John"; last_name = "Smith"; gpa = 3.9}
```

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

```
let name_of_student s = (s.last_name, s.first_name);;
```

```
val name_of_student : student -> string * string = <fun>
```

- `string -> string -> float -> student` (a function that creates a student record)

(using the syntactic sugar mentioned in the chapter)

```
let student first_name last_name gpa = {first_name; last_name; gpa};;
```

```
val student : string -> string -> float -> student = <fun>
```

16. **DONE** Pokerecord [★★]

Here is a variant that represents a few Pokémon types:

```
type poketype = Normal | Fire | Water
```


```
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).

```
type pokemon = {name:string; hp:int; ptype:poketype}
```

```
type pokemon = { name : string; hp : int; ptype : poketype; }
```

- Create a record named `charizard` of type `pokemon` that represents a Pokémon with 78 HP and Fire type.

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

```
val charizard : pokemon = {name = "charizard"; hp = 78; ptype = Fire}
```

- Create a record named `squirtle` of type `pokemon` that represents a Pokémon with 44 HP and Water type.

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

```
val squirtle : pokemon = {name = "squirtle"; hp = 44; ptype = Water}
```

17. **DONE** 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.

Safe `hd` function:

```
let safe_hd = function
  | [] -> None
  | x :: xs -> Some x;;
```

```
val safe_hd : 'a list -> 'a option = <fun>
```

And a couple of tests:

```
[safe_hd [4;2;3];
 safe_hd [1];
 safe_hd []]
```

```
- : int option list = [Some 4; Some 1; None]
```

Safe `tl` function:

```
let safe_tl = function
  | [] -> None
  | x :: xs -> Some xs;;
```

```
val safe_tl : 'a list -> 'a list option = <fun>
```

And a few tests:

```
[safe_tl [4;2;3];
 safe_tl [1];
 safe_tl []]
```

```
- : int list option list = [Some [2; 3]; Some []; None]
```

18. **DONE** 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 lst =
  let rec max_hp_acc p lst = match lst with
  | [] -> p
  | x :: xs -> if (x.hp > p.hp)
               then (max_hp_acc x xs)
               else (max_hp_acc p xs) in
  match lst with
  | [] -> None
  | x :: xs -> Some (max_hp_acc x xs);;
```

```
val max_hp : pokemon list -> pokemon option = <fun>
```

```
[max_hp [charizard; squirtle];
 max_hp [squirtle];
 max_hp []]
```

```
- : pokemon option list =
[Some {name = "charizard"; hp = 78; ptype = Fire};
 Some {name = "squirtle"; hp = 44; ptype = Water}; None]
```

19. **DONE** Date before [⋆⋆]

    Define a date-like triple to be a value of type `int * int * int`. Examples of date-like triples include `(2013, 2, 1)` and `(0, 0, 1000)`. A date is a date-like triple whose first part is a positive year (i.e., a year in the common era), 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). (2013, 2, 1) is a date; (0, 0, 1000) is not.

    Write a function `is_before` that takes two dates as input and evaluates to `true` or `false`. It 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.)

    Your function needs to work correctly only for dates, not for arbitrary date-like triples. However, you will probably find it easier to write your solution if you think about making it work for arbitrary date-like triples. For example, it's easier to forget about whether the input is truly a date, and simply write a function that claims (for example) that January 100, 2013 comes before February 34, 2013—because any date in January comes before any date in February, but a function that says that January 100, 2013 comes after February 34, 2013 is also valid. You may ignore leap years.

    (I'm not convinced this is the "right" way to do this. Need to go back through the chapter and see if I missed anything.

    ```
    type date_like_triple = {year : int;
                             month : int;
                             day : int};;

    let is_before d1 d2 =
      let (y1, m1, d1, y2, m2, d2) = (d1.year,
                                      d1.month,
                                      d1.day,
                                      d2.year,
                                      d2.month,
                                      d2.day) in
      if y1 < y2 then true
      else if y1 > y2 then false
      else if m1 < m2 then true
      else if m1 > m2 then false
      else if d1 < d2 then true
      else if d1 >= d2 then false
      else false;;
    ```

    ```
    val is_before : date_like_triple -> date_like_triple -> bool = <fun>
    ```

    A trivial test:

    ```
    let date1 = {year=1988;month=6;day=22};;
    let date2 = {year=1986;month=7;day=14};;
    [is_before date1 date2; is_before date2 date1]
    ```

    ```
    - : bool list = [false; true]
    ```

20. **DONE** 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`.

As in the previous exercise, your function needs to work correctly only for dates, not for arbitrary date-like triples

```
let earliest lst =
  let rec earliest_carry d lst = match lst with
    | [] -> d
    | x :: xs -> if (is_before x d)
                 then (earliest_carry x xs)
                 else (earliest_carry d xs) in
  match lst with
  | [] -> None
  | x :: xs -> Some (earliest_carry x xs);;
```

```
val earliest : date_like_triple list -> date_like_triple option = <fun>
```

Small test using the two values defined in the previous problem:

```
earliest [date1; date2]
```

```
- : date_like_triple option = Some {year = 1986; month = 7; day = 14}
```

21. **DONE** Assoc list [$\star$]

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`.

Here are `insert` and `lookup` from the section in question:

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

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

```
val insert : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
val lookup : 'a -> ('a * 'b) list -> 'b option = <fun>
```

Here we build the specified association list:

```
let assoc_list =
  []
  |> insert 1 "one"
  |> insert 2 "two"
  |> insert 3 "three";;
```

```
val assoc_list : (int * string) list = [(3, "three"); (2, "two"); (1, "one")]
```

When we lookup `2` we get the expected `string`:

```
lookup 2 assoc_list;;
```

```
- : string option = Some "two"
```

But when we look up `4`, we find `None`:

```
lookup 4 assoc_list;;
```

```
- : string option = None
```

22. **DONE** Cards [★★]

- Define a variant type suit that represents the four suits, (hearts, clubs, diamonds and spades), in a standard 52-card deck. All the constructors of your type should be constant.

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

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

- Define a type rank that represents the possible ranks of a card: 2, 3, . . . , 10, Jack, Queen, King, or Ace. There are many possible solutions; you are free to choose whatever works for you. One is to make rank be a synonym of int, and to assume that Jack=11, Queen=12, King=13, and Ace=1 or 14. Another is to use variants.

```
type face =
  | King
  | Queen
  | Jack

type rank =
  | Number of int
  | Face of face
```

```
type face = King | Queen | Jack
type rank = Number of int | Face of face
```

- Define a type card that represents the suit and rank of a single card. Make it a record with two fields.

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

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

- Define a few values of type card: the Ace of Clubs, the Queen of Hearts, the Two of Diamonds, the Seven of Spades.

```
let ace_of_clubs = {rank = Number 1;
                    suit = Clubs};;

let queen_of_hearts = {rank = Face Queen;
                       suit = Hearts};;

let two_of_diamonds = {rank = Number 2;
                       suit = Diamonds};;

let seven_of_spades = {rank = Number 7;
                       suit = Spades};;
```

```
val seven_of_spades : card = {rank = Number 7; suit = Spades}
```

23. **DONE** Matching [⋆]

For each pattern in the list below, give a value of type `int option list` that does not match the pattern and is not the empty list, or explain why that's impossible.

(a) `Some x :: tl`

`[None]` does not match, since the head does not match

(a) `[Some 3110; None]`

`[None]` does not match, since the head does not match. Also, `[Some 3110; Some 3110]` will not match, since the second element is not `None`.

(a) `[Some x; _]`

Again, `[Some x; None; None]` does not match. It's too long.

(a) `h1 :: h2 :: tl`

Any list of length 2 or greater will match this pattern. But `[None]` does not match it.

(a) `h :: tl`

This pattern matches every list except the empty list, so we can't match it with a nonempty list.

24. **DONE** Quadrant [⋆⋆]

Complete the `quadrant` function. Points that lie on an axis do not belong to any quandrant. 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

let sign (x:int) : sign =
  match x with
  | x when x > 0 -> Pos
  | x when x < 0 -> Neg
  | _ -> Zero

let quadrant : int*int -> quad option = fun (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;;
```

```
type quad = I | II | III | IV
type sign = Neg | Zero | Pos
val sign : int -> sign = <fun>
val quadrant : int * int -> quad option = <fun>
```

A trivial test

```
quadrant (13,-58);;
```

```
- : quad option = Some IV
```

25. **DONE** Quadrant when [★★]

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

```
let quadrant_when : int*int -> quad option = 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;;
```

```
val quadrant_when : int * int -> quad option = <fun>
```

```
quadrant_when (13,-58)
```

```
- : quad option = Some IV
```

26. **DONE** 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`, and the depth of tree `t` above is `3`. Hint: there is a library function `max : 'a -> 'a -> 'a` that returns the maximum of any two values of the same type.

Here's the inductive definition of a tree:

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

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

Here's the tree from section 3.11.1:

```
the code below constructs this tree:

         4
       /   \
      2     5
     / \   / \
    1   3 6   7
```

```
let t =
  Node(4,
    Node(2,
      Node(1, Leaf, Leaf),
      Node(3, Leaf, Leaf)
    ),
    Node(5,
      Node(6, Leaf, Leaf),
      Node(7, Leaf, Leaf)
    )
  )
```

```
val t : int tree =
  Node (4, Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf)),
   Node (5, Node (6, Leaf, Leaf), Node (7, Leaf, Leaf)))
```

Finally, the depth function

```
let depth t =
  let rec depth_tr d t = match t with
  | Leaf -> d
  | Node (x, left, right) -> max (depth_tr (d+1) left) (depth_tr (d+1) right) in
  depth_tr 0 t;;
```

```
val depth : 'a tree -> int = <fun>
```

And a few tests:

```
[depth Leaf;
 depth (Node(1, Leaf, Node(1, Leaf, Leaf)));
 depth t]
```

```
- : int list = [0; 2; 3]
```

27. **DONE** 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(_, left1, right1), Node(_, left2, right2)) -> ((same_shape left1 left2)
  ↪  && (same_shape right1 right2))
  | _ -> false;;
```

```
val same_shape : 'a tree -> 'b tree -> bool = <fun>
```

Test using trees built out of the previous given tree `t`, but with different roots:

```
same_shape (Node(4,t,t)) (Node(1, t, t));;
```

```
- : bool = true
```

28. **DONE** 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 rec list_max_exn lst =
  let rec list_max_exn_acc m lst = match lst with
    | x :: xs -> if (x > m)
                 then (list_max_exn_acc x xs)
                 else (list_max_exn_acc m xs)
    | [] -> m in
  match List.hd lst with
  | exception (_) -> failwith "list_max"
  | m -> list_max_exn_acc m (List.tl lst);;
```

```
val list_max_exn : 'a list -> 'a = <fun>
```

It works as expected for a nonempty list:

```
list_max_exn [1;2;3;4;56;6;7;6;5;4;5;0;0;0;11;12;13];;
```

```
- : int = 56
```

But for an empty list, we get the exception we expected:

```
list_max_exn []
```

```
Exception: Failure "list_max".
```

There is something going on here that I don't understand. I thought that if you had a match expression, every possible match needs to evaluate to the same type. But in the second match expression in the above code, the first branch looks like it has type `exception` while the second has type `int` or maybe `'a`.

I also got a weird warning when I matched with `exception (Failure "hd")` ("fragile-literal-pattern) that went away when I changed to to `exception (_)`, though this seems like a less accurate expression to match against.

29. **DONE** 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"` (note, not the exception `Failure "empty"` but just the string `"empty"` if the list is empty.) Hint: `string_of_int` in the standard library will do what its name suggests.

```
let list_max_string lst =
  let rec list_max_string_acc m lst = match lst with
    | [] -> m
    | x :: xs -> if (x > m)
                 then (list_max_string_acc x xs)
                 else(list_max_string_acc m xs) in
  match lst with
  | [] -> "empty"
  | x :: xs -> list_max_string_acc x xs |> string_of_int;;
```

```
val list_max_string : int list -> string = <fun>
```

The usual tests:

```
[list_max_string [123;252435;12312;345435;123];
 list_max_string [99999;99998];
 list_max_string []]
```

```
- : string list = ["345435"; "99999"; "empty"]
```

30. **TODO** List max exn ounit [⋆]

31. **TODO** 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. Your `is_bst` function will not be recursive, but will call your helper function and pattern match on the result. You will need to define a new variant type for the return type of your helper function.

I don't really understand the signature of the specified function. Why do we need to be working with a tree of ordered pairs of type `('a*'b)` ? It would make sense to write a polymorphic `is_bst` for any `'a tree` where `'a` is a type that that admits a total ordering. But why a tree of pairs of two types?

Maybe just do it for `int tree` for now?

32. **DONE** 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
```

Here's the sign with polymorphic variants. We can see that it has the right signature:

```
let sign = function
  | p when p > 0 -> `Pos
  | n when n < 0 -> `Neg
  | _ -> `Zero
```

```
val sign : int -> [> `Neg | `Pos | `Zero ] = <fun>
```

And quadrant with polymorphic variants. Again, right signature.

```
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
```

```
val quadrant : int * int -> [> `I | `II | `III | `IV ] option = <fun>
```

1.4.3 4.9 Higher-Order Programming - Exercises [14/18]

1. **DONE** Twice, no arguments [⋆]

Consider the following definitions. Use the toplevel to determine what the types of quad and fourth are. Explain how it can be that quad is not syntactically written as a function that takes an argument, and yet its type shows that it is in fact a function.

The `double` function doubles its argument.

```
let double x = 2*x
```

```
val double : int -> int = <fun>
```

The `square` function squares its argument.

```
let square x = x*x
```

```
val square : int -> int = <fun>
```

The `twice` function takes a function `f` and an input `x` and applies `f` to `f x`. In other words it "applies `f` twice"

```
let twice f x = f (f x)
```

```
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

The `quad` function takes an input `x` and doubles it twice. So it should have signature `int -> int`

```
let quad = twice double
```

```
val quad : int -> int = <fun>
```

In other words, `double` is a function of type `int -> int`, while `twice` is (polymorphically) a function that takes a function of type `T -> T` and produces a new function of type `T -> T`. So when applied to `double`, it gives a new function `int -> int`.

Can also think of it in terms of currying: `twice f x` means `f (f x)`, so `twice f` is a function still waiting for its last argument, an integer. Its output will then be `double double` applied to that integer, so the output will also be an integer

```
let fourth = twice square
```

```
val fourth : int -> int = <fun>
```

The same description of `twice double` applies to `twice square` as well, since `double` and `square` have the same type. So this function will also have type `int -> int`, and for the same reason(s).

2. **DONE** Mystery Operator 1 [★★]

   What does the following operator do?

```
let ( $ ) f x = f x;;
```

```
val ( $ ) : ('a -> 'b) -> 'a -> 'b = <fun>
```

`$` is an infix operator that applies its left argument to its right argument. So `f $ x` evaluates to `f x`. But because of operator binding precedence, `double 3 + 1` is `(double 3) + 1`, which is 7. But `double $ 3 + 1` is `($) (double) (3 + 1)`, which is 8 as we see below

```
[double 3 + 1; double $ 3 + 1]
```

```
- : int list = [7; 8]
```

3. **DONE** Mystery Operator 2 [⋆⋆]

   What does the following operator do?

```
let ( @@ ) f g x = x |> g |> f;;
```

`@@` is an "infix" (sort of) operator, where `f @@ g` is a function that, when applied to `x`, gives `f (g x)`. This is function composition. See below for an example usage:

```
(String.length @@ string_of_int) 10;
```

```
- : int = 2
```

Note that this does **not** have the same kind of notationally-favorable binding precedence as the preceeding operator. It would be nice if we didn't need the parentheses in the above example.

4. **DONE** Repeat [⋆⋆]

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

```
let rec repeat f n x = match n with
  | 0 -> x
  | n -> f (repeat f (n-1) x);;
```

```
val repeat : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

If we `double 1` eleven times, we should get `2048`

```
repeat double 11 1;;
```

```
- : int = 2048
```

5. **DONE** 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.

   `fold left` is defined below. For a specific binary function `f`, a starting "accumulation" value `a` and a list like (for example) `[1;2;3]`, it gives `f (f (f a 1) 2) 3`. If the binary function is multiplication and the initial accumulation value is `1`, you'll get the product of the elements in the list.

```
let rec fold_left f acc = function
  | [] -> acc
  | h :: t -> fold_left f (f acc h) t;;

let product_left = fold_left ( * ) 1;;
```

```
val product_left : int list -> int = <fun>
```

```
product_left [1;2;3;4]
```

```
- : int = 24
```

Use `fold_right` to write a function `product_right` that computes the product of a list of floats. Same hint applies

Again, `fold_right` is defined below: Given `f`, `a` and `[1;2;3]` as above, you'd get `f 1 (f 2 (f 3 a))`.

I think the only difference here is that you "need" (probably a way around it though) to specificy the list argument to `product_right`.

```
let rec fold_right f lst acc = match lst with
  | [] -> acc
  | h :: t -> f h (fold_right f t acc);;

let product_right lst = fold_right ( *. ) lst 1.;;
```

```
val product_right : float list -> float = <fun>
```

```
product_right [1.;2.;3.;4.;5.]
```

```
- : float = 120.
```

6. **DONE** 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`.

I think my `product_left` is about as terse as possible already. As noted in the statement of this problem, it doesn't have an explicit list argument. To eliminate the argument from the left hand side of `product_right`, you could use labelled arguments as follows:

```
let rec fold_right ~fn:f ~list:lst ~a:acc = match lst with
  | [] -> acc
  | h :: t -> f h (fold_right ~fn:f ~list:t ~a:acc)

let product_right_terse = fold_right ~fn:( * ) ~a:1;;
```

```
val fold_right : fn:('a -> 'b -> 'b) -> list:'a list -> a:'b -> 'b = <fun>
val product_right_terse : list:int list -> int = <fun>
```

The downside to this approach is that (it seems) you also need to label the argument any time you call `product_right_terse`, though omitting this label only causes a warning and not a true error

```
product_right_terse ~list:[1;2;3;4;5;6]
```

```
- : int = 720
```

(I should figure out exactly the syntax and conventions for labelled argument, since I don't feel like I did this exactly the right way.)

7. **DONE** 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).

   The infix range operator from earlier in the chapter (note to self, it's a little surprising that the expression in the `else` branch doesn't need parentheses around the argument after the `::`, but it does seem to work fine without them)

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

```
val ( -- ) : int -> int -> int list = <fun>
```

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

```
val sum_cube_odd : int -> int = <fun>
```

```
sum_cube_odd 10
```

```
- : int = 1225
```

8. **DONE** sum cube odd pipeline [⋆⋆]

   Rewrite the previous function with the pipeline `|>` operator.

   I already used the `|>` operator a fair bit in the previous work, But I guess with even fewer inner `let` statements and more pipelininig it could be written like this:

```
let sum_cube_odd_pipeline n =
  n
  |> (--) 1
  |> List.filter (fun m -> m mod 2 = 1)
  |> List.map (fun x -> x * x * x)
  |> List.fold_left (+) 0 ;;
```

```
val sum_cube_odd_pipeline : int -> int = <fun>
```

```
sum_cube_odd_pipeline 10
```

```
- : int = 1225
```

9. **DONE** 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.

```
let rec exists_rec p lst = match lst with
  | [] -> false
  | x :: xs -> if p x then true else exists_rec p xs;;
```

```
val exists_rec : ('a -> bool) -> 'a list -> bool = <fun>
```

   Bit of testing with some trivial examples:

```
let even n = n mod 2 = 0;;
let odd n = n mod 2 = 1 || n mod 2 < 0;;

[exists_rec even [1;2;3;4;5;6;7];
 exists_rec odd [-0;-2;-4;-6;-8]]
```

```
- : bool list = [true; false]
```

   - `exists_fold`, which uses either `List.fold_left` or `List.fold_right`, but not any other `List` module functions nor the `rec` keyword.

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

```
val exists_fold : ('a -> bool) -> 'a list -> bool = <fun>
```

   Some tests:

```
[exists_fold even [1;3;5;7];
 exists_fold odd [-2;0;2;6];
 exists_fold even [1;2;3;4;5];
 exists_fold even []]
```

```
- : bool list = [false; false; true; false]
```

- `exists_lib`, which uses any combination of `List` module functions other than `fold_left` or `fold_right`, and does not use the `rec` keyword.

I feel like I've done this in a sort of lazy way. I also don't like the way this is indented, though I think it's "right".

```
let exists_lib p lst =
  match lst
        |> List.find_map (fun x -> if (p x)
                                   then (Some x)
                                   else (None)) with
  | Some x -> true
  | None -> false;;

exists_lib even [1;3;5;8]
```

```
- : bool = true
```

10. **DONE** 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.

    Using `fold_left`:

```
let balance_left acct deblist =
  List.fold_left (+) acct deblist
```

```
val balance_left : int -> int list -> int = <fun>
```

Using `fold_right`:

```
let balance_right acct deblist =
  List.fold_right (+) deblist acct
```

```
val balance_right : int -> int list -> int = <fun>
```

Direct recursive function

```
let rec balance_rec acct deblist = match deblist with
  | [] -> acct
  | d :: ds -> balance_rec (d + acct) ds
```

```
val balance_rec : int -> int list -> int = <fun>
```

some tests:

```
let debs = [1;2;3;-4;10;-2] in
    [balance_left 100 debs;
     balance_right 100 debs;
     balance_rec 100 debs]
```

```
- : int list = [110; 110; 110]
```

11. **DONE** library uncurried [⋆⋆]

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

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

```
val uncurried_nth : 'a list * int -> 'a = <fun>
```

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

- `List.append`

```
let uncurried_append (l1, l2) = List.append l1 l2;;
```

```
val uncurried_append : 'a list * 'a list -> 'a list = <fun>
```

Quick test:

```
uncurried_append ([1;2;3],[3;4;5])
```

```
- : int list = [1; 2; 3; 3; 4; 5]
```

- `Char.compare`

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

```
val uncurried_compare : Char.t * Char.t -> int = <fun>
```

Quick tests:

```
[uncurried_compare ('a','a');
 uncurried_compare ('t','a');
 uncurried_compare ('a','z')]
```

```
- : int list = [0; 19; -25]
```

- `Stdlib.max`

```
let uncurried_max (v1, v2) = Stdlib.max v1 v2;;
```

```
val uncurried_max : 'a * 'a -> 'a = <fun>
```

```
uncurried_max (15, 16)
```

```
- : int = 16
```

12. **DONE** 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.

    With the following setup that loses no generality:

```
let f x = x + 1;;
let g x = 3 * x;;
let lst = [1;2;3;4];;
```

The expression:

```
List.map f (List.map g lst)
```

Could instead be written as follows

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

(Is this what they were expecting? Seems easy for a "three star" exercise.)

13. **DONE** 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`.

```
let long_strings lst =
  let long_enough s = String.length s > 3 in
  List.filter long_enough lst;;
```

```
val long_strings : string list -> string list = <fun>
```

```
long_strings ["a";"hello";"world";"!!";"!";"!!!!"]
```

```
- : string list = ["hello"; "world"; "!!!!"]
```

- Add `1.0` to every element of a list of floats.

```
let increment_floats lst =
  lst |> List.map (fun x -> x +. 1.0);;
```

```
val increment_floats : float list -> float list = <fun>
```

Verify:

```
increment_floats [1.;2.;3.;7.5];;
```

```
- : float list = [2.; 3.; 4.; 8.5]
```

- 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.

Note that the first two cases in the match expression are needed to avoid a comma to the left of the first element.

```
let delimit_strings strs sep = match strs with
  | [] -> ""
  | x :: [] -> x
  | x :: xs -> x ^ (List.fold_left (fun a b -> a ^ sep ^ b) "" xs);;
```

```
val delimit_strings : string list -> string -> string = <fun>
```

```
[delimit_strings ["0";"1";"2";"3";"4";"5";"6";"7";] ", ";
 delimit_strings ["a";"b"] ":";
 delimit_strings [] "delimiter"]
```

```
- : string list = ["0, 1, 2, 3, 4, 5, 6, 7"; "a:b"; ""]
```

14. **DONE** association list keys [$\star\star\star$]

    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`.

    From the initial association list, turn each pair into just it's key. Then take that list of keys and hit it with `sort_uniq` with the appropriate comparison function. The first scan which picks out the keys should be O(n), the sort should be O(n log n). I don't know the space complexity. Creating a new list containing just the keys is O(n), so I'm guessing `sort_uniq` uses O(n log n) space, but I'm not sure.

```
let keys al = List.map (fun (k,v) -> k) al
              |> List.sort_uniq (fun k1 k2 -> if (k1 < k2)
                                              then (-1)
                                              else (if k1 > k2
                                                    then 1
                                                    else 0));;
```

```
val keys : ('a * 'b) list -> 'a list = <fun>
```

```
keys [('a',12);('b',13);('c',120);('c',14);('c',9356);('z',19);('a',53);('d',13);(
↪  'e',63)]
```

```
- : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'z']
```

It's also not clear to me that this is the "one line" solution they're hinting at. My guess is no. Should revisit.

15. **TODO** valid matrix [$\star\star\star$]

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. Unit test the function.

```
let is_valid_matrix m = match m with
  | [] -> false
  | r :: rs -> (match r with
                | [] -> false
                | _ -> let n = List.length r in
                       if List.exists (fun r2 -> List.length r2 <> n) rs then
                       ↪  false else true);;
```

```
val is_valid_matrix : 'a list list -> bool = <fun>
```

Some ordinary tests:

```
[is_valid_matrix [[1;2];[3;4]];
 is_valid_matrix [[1;2;3]];
 is_valid_matrix [[1;2;3];[4;5]];
 is_valid_matrix [];
 is_valid_matrix [[1;2];[3]]]
```

```
- : bool list = [true; true; false; false; false]
```

(I still need to do the unit test part of this problem, so I'm not marking it as done just yet)

16. **TODO** 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`. Unit test the function

    This is what I think they're expecting:

```
let add_row_vectors r1 r2 = List.map2 (+) r1 r2;;
```

```
val add_row_vectors : int list -> int list -> int list = <fun>
```

    Quick test:

```
add_row_vectors [1;2;3] [6;7;10];;
```

```
- : int list = [7; 9; 13]
```

17. **TODO** 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`. Unit test the function.

    Again, I think this is what they're hinting at:

```
let add_matrices m1 m2 = List.map2 add_row_vectors m1 m2;;
```

```
val add_matrices : int list list -> int list list -> int list list = <fun>
```

    Quick test:

```
add_matrices [[0;1;2];[3;4;5];[6;7;8]] [[9;10;11];[12;13;14];[15;16;17]]
```

```
- : int list list = [[9; 11; 13]; [15; 17; 19]; [21; 23; 25]]
```

    Still need to do the unit test part of this problem

18. **TODO** 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.

This seems a little verbose after how concise the previous two problems were. Maybe this can be made ever shorter.

```
let rec multiply_matrices m1 m2 =
  let dot r1 r2 = List.fold_left (+) 0 (List.map2 ( * ) r1 r2) in
  let rec row_to_column r = match r with
    | [] -> []
    | e :: es -> [e] :: row_to_column es in
  let rec transpose m = match m with
    | [] -> []
    | r :: [] -> row_to_column r
    | r :: rs -> List.map2 (@) (row_to_column r) (transpose rs) in
  let rec row_of_r_m r m = match m with
    | [] -> []
    | t :: ts -> (dot r t) :: (row_of_r_m r ts) in
  match m1 with
  | [] -> []
  | r :: rs -> (row_of_r_m r (transpose m2)) :: multiply_matrices rs m2;;
```

```
val multiply_matrices : int list list -> int list list -> int list list =
  <fun>
```

Quick test, using an element of $\mathrm{SL}(2, \mathbb{Z})$ and its inverse:

```
multiply_matrices [[6;41];[1;7]] [[7;-41];[-1;6]]
```

```
- : int list list = [[1; 0]; [0; 1]]
```

Still need to do the unit testing on all these matrix problems before I can mark them as done.

1.4.4 5.11 Modular Programming - Exercises [9/29]

1. **DONE** 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
```

```
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
```

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

2. **DONE** Complex encapsulation [⋆⋆]

   Here is a module for the module type from the previous exercise:

```
module Complex : ComplexSig = struct
  type t = float * float
  let zero = (0., 0.)
  let add (r1, i1) (r2, i2) = r1 +. r2, i1 +. i2
end
```

```
module Complex : ComplexSig
```

Investigate what happens if you make the following changes (each independently), and explain why any errors arise:

- remove `zero` from the structure

```
module Complex : ComplexSig = struct
  type t = float * float
  (*let zero = (0., 0.)*)
  let add (r1, i1) (r2, i2) = r1 +. r2, i1 +. i2
end
```

You get an `Error: Signature mismatch` Specifically, it says `The value 'zero' is required but not provided`. The `ComplexSig` type, defined in the previous problem, requires a `zero` and an `add`. When zero is missing, the structure defined here is not an instance of the ComplexSig type specified.

- remove `add` from the signature

```
module Complex : ComplexSig = struct
  type t = float * float
  let zero = (0., 0.)
  (*let add (r1, i1) (r2, i2) = r1 +. r2, i1 +. i2*)
end
```

Again you get an `Error: Signature mismatch`. This time it says `The value 'add' is required but not provided`. Same issue as above.

- change `zero` in the structure to `let zero = 0, 0`

  ```
  module Complex : ComplexSig = struct
    type t = float * float
    let zero = 0, 0
    let add (r1, i1) (r2, i2) = r1 +. r2, i1 +. i2
  end
  ```

This is a `Signature mismatch` as well, this time because `zero` doesn't have the right type. The `ComplexSig` type needs `zero` to have type `float * float`. Since the `zero` in this module has type `int * int`, it doesn't typecheck as being an instance of `ComplexSig`.

3. **DONE** 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.)

   Need the `Queue` signatur and the `ListQueue` type from section 5.6. Copied here with comments removed, since they were interfering with the emacs / tuareg process in some way.

```
module type Queue = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val enqueue : 'a -> 'a t -> 'a t
  val front : 'a t -> 'a
  val dequeue : 'a t -> 'a t
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end
```

```ocaml
module type Queue =
  sig
    type 'a t
    exception Empty
    val empty : 'a t
    val is_empty : 'a t -> bool
    val enqueue : 'a -> 'a t -> 'a t
    val front : 'a t -> 'a
    val dequeue : 'a t -> 'a t
    val size : 'a t -> int
    val to_list : 'a t -> 'a list
  end
```

```ocaml
module ListQueue : Queue = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let enqueue x q = q @ [x]
  let front = function [] -> raise Empty | x :: _ -> x
  let dequeue = function [] -> raise Empty | _ :: q -> q
  let size = List.length
  let to_list = Fun.id
end
```

```ocaml
module ListQueue : Queue
```

```ocaml
(** 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;;

let timing f x =
  let t1 = Sys.time() in
  let result = f x in
  let t2 = Sys.time() in
  (result, t2 -. t1);;
```

```ocaml
val timing : ('a -> 'b) -> 'a -> 'b * float = <fun>
```

Now we can do `timing fill_listqueue n;;` to time it `n = 10000` took about 1 second, `n = 50000` took about 30 seconds.

4. **TODO** Big batched queue [★★]

5. **TODO** Queue efficiency [★★★]

6. **TODO** Binary search tree map [⋆⋆⋆⋆]
7. **DONE** Fration [⋆⋆⋆]

   Write a module that implements the Fraction module type below:

```
module type Fraction = sig
  type t
  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 type Fraction =
  sig
    type t
    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 Frac : Fraction = struct
  type t = int * int
  let make a b = (a, b)
  let numerator (a,b) = a
  let denominator (a,b) = b
  let to_string (a,b) = (string_of_int a)
                        ^ "/"
                        ^ (string_of_int b)
  let to_float (a,b) = (float_of_int a)
                       /. (float_of_int b)
  let add (a,b) (c,d) = (a*d + b*c, b*d)
  let mul (a,b) (c,d) = (a*c, b*d)
end
```

```
module Frac : Fraction
```

```
let q = Frac.make 1 2;;
let r = Frac.make 2 7;;
let s = Frac.add q r in
  Frac.to_string s
```

```
- : string = "11/14"
```

Didn't really think about how to handle / avoid the case where the denominator is zero.

8. **DONE** Fraction reduced [⋆⋆⋆]

   Modify your implementation of `Fraction` to ensure these invariants hold of every value `v` of type `t` that is returned from `make`, `add`, and `mul`:

   - `v` is in reduced form
   - the denominator of `v` is positive

   For the first invariant, you might find this implementation of Euclid's algorithm to be helpful:

```
(** [gcd x y] is the greatest common divisor of [x] and [y].
    Requires: [x] and [y] are positive. *)
let rec gcd x y =
  if x = 0 then y
  else if (x < y) then gcd (y - x) x
  else gcd y (x - y)
```

```
module Frac : Fraction = struct
  type t = int * int

  let make a b = let d = gcd a b in
                 (a/d, b/d)

  let numerator (a,b) = a

  let denominator (a,b) = b

  let to_string (a,b) = (string_of_int a)
                        ^ "/"
                        ^ (string_of_int b)

  let to_float (a,b) = (float_of_int a)
                       /. (float_of_int b)

  let add (a,b) (c,d) = let d = gcd (a*d + b*c) (b*d) in
                        (a*d + b*c, b*d)

  let mul (a,b) (c,d) = let d = gcd (a*c) (b*d) in
                        (a*c, b*d)
end;;
```

```
module Frac : Fraction
```

```
Frac.make 31991 101 |> Frac.to_string |> print_endline;
Frac.make 72 324 |> Frac.to_string |> print_endline;;
```

```
31991/101
2/9
- : unit = ()
```

```
let q = Frac.make 72 324 in
  let r = Frac.make 31991 101 in
  Frac.mul q r |> Frac.to_string
```

```
- : string = "63982/9"
```

9. **DONE** Make char map [⋆]

   To create a standard library map, we first have to use the Map.Make functor to produce a module that is specialized for the type of keys we want. Type the following in utop: `module CharMap = Map.Make(Char);;` The output tells you that a new module named CharMap has been defined, and it gives you a signature for it. Find the values `empty`, `add`, and `remove` in that signature. Explain their types in your own words.

   Here's what I get:

```
module CharMap = Map.Make(Char);;
```

```
module CharMap :
  sig
    type key = Char.t
    type 'a t = 'a Map.Make(Char).t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val mem : key -> 'a t -> bool
    val add : key -> 'a -> 'a t -> 'a t
    val update : key -> ('a option -> 'a option) -> 'a t -> 'a t
    val singleton : key -> 'a -> 'a t
    val remove : key -> 'a t -> 'a t
    val merge :
      (key -> 'a option -> 'b option -> 'c option) -> 'a t -> 'b t -> 'c t
    val union : (key -> 'a -> 'a -> 'a option) -> 'a t -> 'a t -> 'a t
    val compare : ('a -> 'a -> int) -> 'a t -> 'a t -> int
    val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val for_all : (key -> 'a -> bool) -> 'a t -> bool
    val exists : (key -> 'a -> bool) -> 'a t -> bool
    val filter : (key -> 'a -> bool) -> 'a t -> 'a t
    val filter_map : (key -> 'a -> 'b option) -> 'a t -> 'b t
    val partition : (key -> 'a -> bool) -> 'a t -> 'a t * 'a t
    val cardinal : 'a t -> int
    val bindings : 'a t -> (key * 'a) list
    val min_binding : 'a t -> key * 'a
    val min_binding_opt : 'a t -> (key * 'a) option
    val max_binding : 'a t -> key * 'a
    val max_binding_opt : 'a t -> (key * 'a) option
    val choose : 'a t -> key * 'a
    val choose_opt : 'a t -> (key * 'a) option
    val split : key -> 'a t -> 'a t * 'a option * 'a t
    val find : key -> 'a t -> 'a
    val find_opt : key -> 'a t -> 'a option
    val find_first : (key -> bool) -> 'a t -> key * 'a
    val find_first_opt : (key -> bool) -> 'a t -> (key * 'a) option
    val find_last : (key -> bool) -> 'a t -> key * 'a
    val find_last_opt : (key -> bool) -> 'a t -> (key * 'a) option
    val map : ('a -> 'b) -> 'a t -> 'b t
    val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
    val to_seq : 'a t -> (key * 'a) Seq.t
    val to_rev_seq : 'a t -> (key * 'a) Seq.t
    val to_seq_from : key -> 'a t -> (key * 'a) Seq.t
    val add_seq : (key * 'a) Seq.t -> 'a t -> 'a t
    val of_seq : (key * 'a) Seq.t -> 'a t
  end
```

For the functions in question:

- `val empty : 'a t`

`empty` creates a new map with empty domain. It has type `'a t`, where `'a` is the (as of yet unknown) type of the codomain of the map. It's going to be a map from a set of things of type `char` to a set of things of type `'a`. That type variable will be clarified for maps where the type of the elements in codomain is known.

Note: in `org-mode` I can run this block, which lets me avoid prefacing future code with the module name, so I can just call `add` and `remove` instead of `CharMap.add` and `CharMap.remove` in future source blocks. But for some reason, this is not the case when org-export re-runs the source blocks. It will complain that `add` is a function `int -> int -> int`. (This can be worked around by changing the settings, see the "org latex export" section above.)

```
open CharMap
```

Here we create an empty map using `empty`, noting that its type ia `'a CharMap.t`:

```
let emptymap = empty
```

```
val emptymap : 'a CharMap.t = <abstr>
```

But if we then add a key-value pair to it, the type is clarified:

```
let map_to_ints = add 'x' 3 emptymap
```

```
val map_to_ints : int CharMap.t = <abstr>
```

But if we had instead added a key-value pair where the value was of type `string`, the type of the result will change accordingly:

```
let map_to_strings = add 'x' "target value" empty
```

```
val map_to_strings : string CharMap.t = <abstr>
```

- `val add : key -> 'a -> 'a t -> 'a t`

`add` takes a value of type `key` (which in this case is a type synonym for `char`, I think), a value of type `'a`, and an existing map from type `char` to type `'a` and gives a new map, with the specified key-value pair added. If `'a` is not yet known (like when adding a new pair to the empty map above) it can be inferred. But when `'a` is known, the types need to match or an error will be thrown. So for example, this works fine:

```
let map_to_strings = add 'x' "aaa" empty in
    add 'y' "bbb" map_to_strings
```

```
- : string CharMap.t = <abstr>
```

While this will fail (output suppressed) because: we can't add a `char*int` key-value pair to a map full of of `char*string` key-value pairs

```
let map_to_strings = add 'x' "aaa" empty in
    add 'y' 7 map_to_strings
```

- `val remove : key -> 'a t -> 'a t`

`remove` takes something of type `key` (again, `char` in this case) and a map and removes the key-value pair from that map.

```
let map_to_strings = add 'x' "aaa" empty in
    map_to_strings
    |> add 'y' "bbb"
    |> remove 'x'
    |> remove 'y'
```

```
- : string CharMap.t = <abstr>
```

It's worth noting that once the type of the codomain is known, it is never forgotten. The result of the above expression is an empty map, but it still knows that it's a `string CharMap.t`. Even though it contains no keys, you would get an error if you added the line `|> add 'x' 2` at the bottom of that expression, since `2` is not a `string`

10. **DONE** Char order [⋆]

    The `Map.Make` functor requires its input module to match the `Map.OrderedType` signature. Look at that signature as well as the signature for the Char module. Explain in your own words why we are allowed to pass `Char` as an argument to `Map.Make`.

    The `Map.OrderedType` signature has two entries (what's the right noun for these? It's not "entries")

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

    A type needs to implement both in order to be accepted as an input to the `Map.Make` functor. (This seems like an abuse of category theory, though maybe I'm thinking of it incorrectly)

    Meanwhile, the siguanture for the `Char` module contains (among other things)

    - `type t = char`
    - `val compare : t -> t -> int`

    This means `Char` implements the comparison necessary to be passed as an input type to the `Map.Make` functor.

    Maybe another way to say this is that the domain of `Map.Make` is the category of ordered types, and and `Char` is an object in that category.

11. **DONE** 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"`

(This relies on the following two lines from the "Make char map" exercise above. The first creates the module, the second lets you avoid prefacing everything with `CharMap.`, so you can just use `add` instead of `CharMap.add`, etc)

```
module CharMap = Map.Make(Char);;
open CharMap
```

Here's what they're asking for.

```
let my_map =
  empty
  |> add 'A' "Alpha"
  |> add 'E' "Echo"
  |> add 'S' "Sierra"
  |> add 'V' "Victor"
```

```
val my_map : string CharMap.t = <abstr>
```

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

```
find 'E' my_map
```

```
- : string = "Echo"
```

Now remove the binding for `'A'`. Use `CharMap.mem` to find whether `'A'` is still bound. (of course, need to use a `let` expression since the `remove` doesn't change anything, it just makes a new `CharMap`.

```
let my_map = remove 'A' my_map
```

```
val my_map : string CharMap.t = <abstr>
```

```
mem 'A' my_map
```

```
- : bool = false
```

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

```
bindings my_map
```

```
- : (CharMap.key * string) list =
[('E', "Echo"); ('S', "Sierra"); ('V', "Victor")]
```

12. **DONE** Bindings [⋆⋆]

    Investigate the documentation of the `Map.S` signature to find the specification of `bindings`. Which of these expressions will return the same association list?

    - `CharMap.(empty |> add 'x' 0 |> add 'y' 1 |> bindings)`
    - `CharMap.(empty |> add 'y' 1 |> add 'x' 0 |> bindings)`
    - `CharMap.(empty |> add 'x' 2 |> add 'y' 1 |> remove 'x' |> add 'x' 0 |> bindings)`

    My guess is that they'll all return the same association list, because all three of these result in a map sending `'x'` to `0` and `'y'` to `'1'`. The order in which the keys are added (or added then removed then added again) to the maps are different in each case. But `bindings` specifies that the association list it returns will be sorted by keys. So the same maps will give the same `bindings` regardless of how the maps were created. Let's test:

    ```
    [CharMap.(empty
              |> add 'x' 0
              |> add 'y' 1
              |> bindings);
     CharMap.(empty
              |> add 'y' 1
              |> add 'x' 0
              |> bindings);
     CharMap.(empty
              |> add 'x' 2
              |> add 'y' 1
              |> remove 'x'
              |> add 'x' 0
              |> bindings)]
    ```

    ```
    - : (CharMap.key * int) list list =
    [[('x', 0); ('y', 1)]; [('x', 0); ('y', 1)]; [('x', 0); ('y', 1)]]
    ```

    They are indeed all the same association list.

13. **TODO** Date order [⋆⋆]
14. **TODO** Calendar [⋆⋆]
15. **TODO** Print calendar [⋆⋆]
16. **TODO** Is for [⋆⋆⋆]
17. **TODO** First after [⋆⋆⋆]
18. **TODO** Sets [⋆⋆⋆]
19. **TODO** ToString [⋆⋆]
20. **TODO** Print [⋆⋆]
21. **TODO** Print int [⋆⋆]
22. **TODO** Print string [⋆⋆]

23. **TODO** Print reuse [$\star$]

24. **TODO** Print string reuse revisited [$\star\star$]

25. **TODO** Implementation without interface [$\star$]

26. **TODO** Implementation with interface [$\star$]

27. **TODO** Implementation with abstracted interface [$\star$]

28. **TODO** Preinter for date [$\star\star\star$]

29. **TODO** Refactor arith [$\star\star\star\star$]

1.4.5 6.11 Correctness - Exercises [1/22]

1. **TODO** spec game [***]

2. **TODO** poly spec [***]

3. **TODO** poly impl [***]

4. **TODO** interval arithmetic [****]

5. **TODO** function maps [****]

6. **TODO** set black box [***]

7. **TODO** set glass box [***]

8. **TODO** random lists [***]

9. **TODO** qcheck odd divisor [***]

10. **TODO** qcheck avg [****]

11. **DONE** exp [**]

    Prove that `exp x (m + n) = exp x m * exp x n`, where

    ```
    let rec exp x n =
      if n = 0 then 1 else x * exp x (n - 1)
    ```

    Proceed by induction on n.

    When $n = 0$, we have:

    ```
      exp x (m + n)
    = exp x (m + 0)      (by assumption)
    = exp x m            (arith)
    = exp x m * 1        (arith)
    = exp x m * exp x 0 (by definition)
    = exp x m * exp x n (by assumption)
    ```

    Now assume the equality holds for some fixed n value, say $n = k$. It remains to prove the equality in the case where $n = k + 1$:

```
  exp x (m + (k + 1))
= exp x ((m + k) + 1)         (associativity of +)
= exp x (m + k) * x           (by definition of exp)
= exp x m * exp x k * x       (by induction)
= exp x m * exp x k * exp x 1 (by definition)
= exp x m * exp x (k + 1)     (by definition of exp)
```

This concludes the proof.

12. **TODO** fibi [***]
13. **TODO** expsq [***]
14. **TODO** mult [**]
15. **TODO** append nil [**]
16. **TODO** rev dist append [***]
17. **TODO** rev involutize [***]
18. **TODO** reflect size [***]
19. **TODO** fold theorem 2 [****]
20. **TODO** propositions [****]
21. **TODO** list spec [***]
22. **TODO** bag spec [****]

1.4.6 7.5 Mutability - Exercises [10/11]

1. **DONE** mutable fields [$\star$]

   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`

   Here's a record type with a mutable gpa field:

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

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

   Create the specified instance:

```
let student_rec = {name = "Alice"; gpa = 3.7};;
```

```
val student_rec : student = {name = "Alice"; gpa = 3.7}
```

   Change the mutable field, as specified:

```
student_rec.gpa <- 4.0;;
```

```
- : unit = ()
```

Inspect to confirm:

```
student_rec
```

```
- : student = {name = "Alice"; gpa = 4.}
```

2. **DONE** refs [⋆]

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

   - `bool ref`

```
let br = ref true;;
```

```
val br : bool ref = {contents = true}
```

   - `int list ref`

```
let ilr = ref [1;2;3]
```

```
val ilr : int list ref = {contents = [1; 2; 3]}
```

   - `int ref list`

```
List.map (fun i -> ref i) [1;2]
```

```
- : int ref list = [{contents = 1}; {contents = 2}]
```

3. **DONE** inc fun [⋆]

   Define a reference to a function as follows:

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

```
val inc : (int -> int) ref = {contents = <fun>}
```

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

This is pretty gross, but it feels like cheating to just do something like `!inc 3109`. So, start with three `int ref` counters, all initially containing `0`. The increment each of them until they're equal to `2`, `5`, and `311` (prime factorization of `3110`. Then multiply them together).

```
let p = ref 0 in
let q = ref 0 in
let r = ref 0 in
while ((!p) < 2)
do (p := !inc !p)
done;
while ((!q) < 5)
do (q := !inc !q)
done;
while ((!r) < 311)
do (r := !inc !r)
done;
(!p) * (!q) * (!r);
```

```
- : int = 3110
```

4. **DONE** 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`.

   Uncomfortably close to unreadable line noise here. This function definition is like 60% punctuation:

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

```
val ( +:= ) : int ref -> int -> unit = <fun>
```

   A quick test:

```
let x = ref 0;;

x +:= 12;;
x +:= 28;;
x +:= -3;;

!x;;
```

```
- : int = 37
```

5. **DONE** physical equality [⋆⋆]

   Define `x`, `y`, and `z` as follows:

```
let x = ref 0
let y = x
let z = ref 0
```

```
val x : int ref = {contents = 0}
val y : int ref = {contents = 0}
val z : int ref = {contents = 0}
```

Predict the value of the following series of expressions:

- `x == y;;`
- `x == z;;`
- `x = y;;`
- `x = z;;`
- `x := 1;;`
- `x = y;;`
- `x = z;;`
- `# x == y;;`

`y` is another name for `x`. They should be equal.

```
x == y
```

```
- : bool = true
```

- `# x == z;;`

`x` and `z` are two different references. Different boxes with the same content are not the same box. They should not be equal

```
x == z
```

```
- : bool = false
```

- `# x = y;;`

My guess is that structural equality (same thing in memory) is stronger than mathematical equality (evaluate to the same value), so I'm guessing this is true:

```
x = y
```

```
- : bool = true
```

- `# x = z;;`

both `x` and `z` are the same "value" (a reference containing a zero), so I expect them to be "equal" despite not being the same reference.

```
x = z
```

```
- : bool = true
```

- `# x := 1;;`

Switching the contents of reference `x` from `0` to `1`.

```
x := 1
```

```
- : unit = ()
```

- `# x = y;;`

`y` is just a different name for the exact same location in memory. When we changed `x`, we also changed `y`. They are still (structurally) equal so they should still be mathematically equal

```
x = y
```

```
- : bool = true
```

- `# x = z;;`

These two used to be references containing the same value. But now `x` contains `1` while `z` still contains `0`. So they should no longer be equal.

```
x = z
```

```
- : bool = false
```

6. **DONE** 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:

```
type vector = float array
```

```
let norm (v: vector) =
  v
  |> Array.map (function x -> x *. x)
  |> Array.fold_left (+.) 0.
  |> Float.sqrt;;
```

```
val norm : vector -> float = <fun>
```

It would probably be fine to leave the type declaration (`v: vector`)out of the function definition, which would result in an "identical" function with type signature `float array -> float` instead of `vector -> float`. Since `vector` is just a type synonym this isn't a meaningful change. But it's nice to know how to make the change. Note that the parentheses are necessary; without them, OCaml thinks that `vector` is the type of the function's output, and it throws a type error.

Some tests:

```
[norm [|5.0; 12.0|];
 norm [|0.0;12.0;34.0;56.0;78.0|]]
```

```
- : float list = [13.; 102.567051239664679]
```

7. **DONE** normalize [★★]

   Every vector x can be normalized by dividing each component by $|x|$. This yields a vector with norm 1.

   Write a function `normalize : vector -> unit` that normalizes a vector "in place" by mutating the input array. Here's a sample usage:

```
# let a = [|1.; 1.|];;
val a : float array = [|1.; 1.|]

# normalize a;;
- : unit = ()

# a;;
- : float array = [|0.7071...; 0.7071...|]
```

The following works and doesn't use a loop, but it's not clear to me that it's the "right" way to do this. Seems like an abuse of `mapi`, and my suspicion is there's something in the standard library that's better suited to this purpose.

```
let normalize vect =
  let n = norm vect in
  let replace_at i e = vect.(i) <- e /. n in
  ignore (vect |> Array.mapi replace_at);;
```

```
val normalize : vector -> unit = <fun>
```

Quick check:

```
let v = [|3.0; 4.0|] in
normalize v;
norm v
```

```
- : float = 1.
```

8. **DONE** norm loop [★★]

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

   Here it is with a loop:

```
let norm vect =
  let len = Array.length vect in
  let sum_of_squares = ref 0.0 in
  let i = ref 0 in
  while (!i < len)
  do (sum_of_squares := !sum_of_squares +. (vect.(!i) *. vect.(!i));
      i := !i + 1)
  done;
  Float.sqrt(!sum_of_squares);;
```

```
val norm : float array -> float = <fun>
```

   Quick check:

```
norm [|5.0; 12.0|]
```

```
- : float = 13.
```

9. **DONE** normalize loop [★★]

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

```
let normalize vect =
  let len = Array.length vect in
  let n = norm vect in
  let i = ref 0 in
  while !i < len
  do (vect.(!i) <- vect.(!i) /. n;
      i := !i + 1)
  done;;
```

```
val normalize : float array -> unit = <fun>
```

```
let v = [| 3.0; 4.0 |] in
print_endline (string_of_float (norm v));
normalize v;
print_endline (string_of_float (norm v));;
```

```
5.
1.
- : unit = ()
```

10. **DONE** 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.

    See the documentation for `make_matrix` for more information on the representation of matrices as arrays.

    (I refuse to use "$n \times o$ matrix". o is not an index variable. All matrices are $m \times n$. C'mon now.)

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

```
val init_matrix : int -> int -> (int -> int -> 'a) -> 'a array array = <fun>
```

    Quick check:

```
init_matrix 4 4 (fun i j -> i + 2*j)
```

```
- : int array array =
[|[|0; 2; 4; 6|]; [|1; 3; 5; 7|]; [|2; 4; 6; 8|]; [|3; 5; 7; 9|]|]
```

11. **TODO** doubly linked list [⋆⋆⋆⋆]

1.4.7 8.9 Data Structures - Exercises [0/44]

1. **TODO** hash insert [**]
2. **TODO** relax bucket RI [**]
3. **TODO** strengthen bucket RI [**]
4. **TODO** hash values [**]
5. **TODO** hashtbl usage [**]
6. **TODO** hashtbl stats [*]
7. **TODO** hashtbl bindings [**]
8. **TODO** hashtbl load factor [**]
9. **TODO** functorial interface [***]
10. **TODO** equals and hash [**]
11. **TODO** bad hash [**]
12. **TODO** linear probing [****]

13. **TODO** functorized BST [***]
14. **TODO** efficient traversal [***]
15. **TODO** RB draw complete [**]
16. **TODO** RB draw insert [**]
17. **TODO** standard library set [**]
18. **TODO** pow2 [**]
19. **TODO** more sequences [**]
20. **TODO** nth [**]
21. **TODO** hd tl [**]
22. **TODO** filter [***]
23. **TODO** interleave [***]
24. **TODO** sift [***]
25. **TODO** primes [***]
26. **TODO** approximately e [****]
27. **TODO** better e [****]
28. **TODO** different sequence rep [***]
29. **TODO** lazy hello [*]
30. **TODO** lazy and [**]
31. **TODO** lazy sequence [***]
32. **TODO** promise and resolve [**]
33. **TODO** promise and resolve lwt [**]
34. **TODO** timing challenge 1 [**]
35. **TODO** timing challenge 2 [***]
36. **TODO** timing challenge 3 [***]
37. **TODO** timing challenge 4 [***]
38. **TODO** file monitor [****]
39. **TODO** add opt [**]
40. **TODO** fmap and join [**]
41. **TODO** fmap and join again [**]
42. **TODO** bind from fmap+join [***]
43. **TODO** list monad [***]
44. **TODO** trivial monad laws [***]

1.4.8 9.5 Interpreters - Exercises [0/32]

1. **TODO** parse [*]
2. **TODO** simpl ids [**]
3. **TODO** times parsing [**]
4. **TODO** infer [**]
5. **TODO** subexpression types [*]
6. **TODO** typing [**]
7. **TODO** substitution [**]
8. **TODO** step expression [*]
9. **TODO** step let expression [**]
10. **TODO** variants [*]
11. **TODO** application [**]
12. **TODO** omega [***]
13. **TODO** pair parsing [***]
14. **TODO** pair type checking [***]
15. **TODO** pair evaluation [***]
16. **TODO** desugar list [*]
17. **TODO** list not empty [**]
18. **TODO** list not empty [****]
19. **TODO** let rec [****]
20. **TODO** simple expression [*]
21. **TODO** let and match expressions [**]
22. **TODO** closures [**]
23. **TODO** lexical scope and shadowing [**]
24. **TODO** more evaluation [**]
25. **TODO** dynamic scope [***]
26. **TODO** more dynamic scope [***]
27. **TODO** constraints [**]
28. **TODO** unify [**]
29. **TODO** unify more [***]
30. **TODO** infer apply [***]
31. **TODO** infer double [***]
32. **TODO** infer S [****]