Contents

1 Learning OCaml - Notes

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

1.1 Goals

- Learning OCaml
- Literate programming in org-mode

1.2 Remarks on OCaml in org-mode

These variables control something about the way code gets passed to and retrieved from the ocaml toplevel that emacs runs in the background. The toplevel can look a little cluttered with this expression repeated all over the place, but it's probably not worth changing.

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

(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 the type of the result:

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

But with the verbatim tag, the type is displayed as well.

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

And in this source block, when a string printing function is called, the printed string doesn't manage to make it to the results line. Again, the verbatim tag seems fixes this

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

There is also the `:results output` tag. This is similar to `:results verbatim` but it deosn't seem to handle multi-line input:

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

In the event that source blocks aren't sufficient, you can open the actual running toplevel with `M-x tuareg-run-ocaml` and interact with it directly.

1.3 CS3110 - Notes

1.4 TODO CS3110 Exercises [75/212] [35%]

1.4.1 DONE 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)`
   - `"CS " ^ string_of_int 3110`

   The first is `42 : int`, the second is `CS 3110 : string`

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

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

2. **DONE** Operators $[⋆⋆]$$$

   - Write an expression that multiplies 42 by 10

   ```
   42 * 10
   ```

   - Write an expression that divides 3.14 by 2.0

   ```
   3.14 /. 2.0
   ```

   - 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
   ```

3. **DONE** Equality [⋆]

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

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

```
42 = 42
```

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

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

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

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

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

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 it's usually better to use `=`.

4. **DONE** Assertions [⋆]

   - Enter `assert true;;` into utop and see what happens.
   - Enter `assert false;;` into utop and see what happens.
   - Write an expression that asserts 2110 is not (structurally) equal to 3110.

   `assert true;;` seems to do "nothing" with type unit. Assert false throws an exception (Assert_failure)

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

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;;
   ```

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.

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;;

   cube 1.5;;
   cube 2.1;;
   cube Float.pi;;
   ```

- 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.
- 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;;

area 1.0;;
area 2.0;;

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

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);;

rms 3. 4.;;
rms 5. 12.;;
rms 7399. 10200.;;
```

   Test with some 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);;

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

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? 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;;

valid_date "Apr" 20
```

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);;

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

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.

```
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;;

fibtimer 50;;
```

    Prints `found fib 50 = 12586269025 in 257.446328 seconds.` Slow.

    Prints `found fib_fast 50 = 12586269025 in 4.99999998738e-06 seconds.`, 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
```

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, it has to be a boolean. Since the output is always `x`, the output of `f` will be boolean. So `f` is a function `bool -> bool`.

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

    Here, `y` needs to be boolean. 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 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 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`

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;;

divide 0. 0.
```

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

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.

- `add 5 1`

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

- `add 5`

This is `add` applied to one argument. It is the "add five" function.

- `(add 5) 1`

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

- `add (5 1)`

This will produce an error. 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.`

16. **DONE** Hello World [⋆]

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

- `print_endline "Hello world!";;`

Prints the string, with a carriage return at the end. Has type `unit`. Output looks like:

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

- `print_string "Hello world!";;`

Prints the string with no newline. Has type unit. Output looks like:

1.4.2 TODO 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];;
```

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

```
let l2 = 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];;
```

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;;

list_product (l1 @ l2 @ l3)
```

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;;

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

4. **DONE** product test [**]

   Relevant files in `standalone` directory.

   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 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:

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 seem to have a workflow that works and "fixes" (suppresses) 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, create the following files:

The product function is in `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 a `dune` file

But also a `dune-project` file, containing

(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:

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;;

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

   (I'm not sure how to make this polymorphic: if the first element is an integer, I get a type error

   - 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;;

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

   - 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;;

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

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);;
```

- 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;;
```

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

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);;

last_element [1;4;3;2;3;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;;

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]
```

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)
          | [] -> []);;

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

- 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
       | [] -> []);;

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

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

```
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 []);;

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

I am not sure I 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.

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

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

Still not 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 on its own. I also think that `drop 999999 (1 -- 1000000)` would have stack overflowed if this wasn't tail recursive.

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));;

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];;
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]
```

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

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;;

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

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;;

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

15. **DONE** Student [**]

    Assume the following type definition:

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

    Give OCaml expressions that have the following types:

    - `student`

```
let s = {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);;

name_of_student s;;
```

    - `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};;
```

16. **DONE** Pokerecord [**]

    Here is a variant that represents a few Pokémon types:

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

    - Define the type `pokemon` to be a record with fields `name` (a string), `hp` (an integer), and `ptype` (a poketype).

```
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}
```

- 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}
```

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.

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

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

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

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

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 m lst = match lst with
  | [] -> m
  | x :: xs -> if (x.hp > m)
               then (max_hp_acc x.hp xs)
               else (max_hp_acc m xs) in
  match lst with
  | [] -> None
  | x :: xs -> let m = x.hp in Some (max_hp_acc m xs);;

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

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 think this isn't 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;;

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

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);;

earliest [date1; date2]
```

21. **DONE** Assoc list [*]

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

    Here are `insert` and `lookup`:

```
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
```

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

lookup 2 assoc_list;;
lookup 4 assoc_list;;
```

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
```

- 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
```

- 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}
  ```

- 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};;
```

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

(b) `[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`.

(c) `[Some x; _]`

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

(d) `h1 :: h2 :: tl`

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

(e) `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 below, which should return the quadrant of the given `x, y` point according to the diagram on the right (borrowed from Wikipedia). 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;;

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

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;;

quadrant_when (13,-58)
```

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.

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

    Here's the tree from 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)
    )
  )
```

```
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;;

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

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;;

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

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);;

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

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 less accurate.

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;;

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

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 order. 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
```

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

```
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
```

1.4.3 TODO 4.9 Higher-Order Programming - Exercises [13/18]

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.

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

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

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

```
let quad = twice double
```

`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.

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

```
let fourth = twice square
```

The same description of `twice double` applies to `twice square` as well, since `double` and `square` have the same type.

2. **DONE** Mystery Operator 1 [**]

   What does the following operator do?

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



double $ 3 + 1;;

($) (double) (3 + 1) ;;
```

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

3. **DONE** Mystery Operator 2 [**]

   What does the following operator do?

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

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

   `@@` is an "infix" (sort of) operator, where `f @@ g` is a function that, when applied to `x`, give `f (g x)` (as opposed to `f g x`). This is function composition.

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);;

repeat double 10 1;;
```

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;;

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

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;;

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

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. To eliminate the argument from the left hand side of product_right, you could do:

```
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;;

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

   (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:

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

```
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 ;;

sum_cube_odd 10
```

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

   Rewrite the previous function with the pipeline `|>` operator. (I already used it a fair bit in the previous, But I guess with even fewer inner `let` statements and more pipelininig it could be written:

```
  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 ;;

sum_cube_odd_pipeline 10
```

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;;

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 [-2;-4;-6;-8]
```

   - `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;;

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 []
```

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

```
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]
```

(There are probably lots of ways to do this one)

10. **TODO** 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.

11. **DONE** library uncurried [**]

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

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

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

    - `List.append`

```
let uncurried_append (l1, l2) = List.append l1 l2;;
uncurried_append ([1;2;3],[3;4;5])
```

    - `Char.compare`

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

uncurried_compare ('a','a');;
uncurried_compare ('a','z');;
```

    - `Stdlib.max`

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

uncurried_max (15, 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.

    The expression

```
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
```

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;;

long_strings ["a";"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);;

increment_floats [1.;2.;3.;7.];;
```

    - Given a list of strings strs and another string sep, produce the string that contains every element of strs separated by sep. For example, given inputs ["hi";"bye"] and ",", produce "hi,bye", being sure not to produce an extra comma either at the beginning or end of the result string.

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

delimit_strings ["0";"1";"2";"3";"4";"5";"6";"7";] " -- "
```

14. **DONE** association list keys [***]

    Recall that an association list is an implementation of a dictionary in terms of a list of pairs, in which we treat the first component of each pair as a key and the second component as a value.

    Write a function `keys: ('a * 'b) list -> 'a list` that returns a list of the unique keys in an association list. Since they must be unique, no value should appear more than once in the output list. The order of values output does not matter. How compact and efficient can you make your solution? Can you do it in one line and linearithmic space and time? Hint: `List.sort_uniq`.

```
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));;

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

    I don't know if this is n log n space and time. I'm also not sure if this is the "one line" solution they're hinting at, since it's a bit long for one line.

15. **TODO** valid matrix [***]

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

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

    would be represented as the list `[[1; 1; 1]; [9; 8; 7]]`. Let's represent a row vector as an int list. For example, `[9; 8; 7]` is a row vector.

    A valid matrix is an `int list list` that has at least one row, at least one column, and in which every column has the same number of rows. There are many values of type `int list list` that are invalid, for example

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

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

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

      (still need to do the unit test part of this problem)

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

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

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

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.

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

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

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.

```
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;;

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

Done, but still need to do the unit testing on all these matrix problems

1.4.4 TODO 5.11 Modular Programming - Exercises [4/29]

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
```

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
```

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
```

   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
```

   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
```

   Same problem as above: the type `ComplexSig` needs an `add` function. If it's missing, you don't have an instance of that type

   - 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
```

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. **TODO** 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
```

```
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
```

```
(** 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);;

timing fill_listqueue 50000;;
```

`10000` took about 1 second, `50000` took about 30.

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 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
```

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

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;;

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

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

9. **TODO** Make char map [*]
10. **TODO** Char order [*]
11. **TODO** Use char map [**]
12. **TODO** Bindings [**]
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 [*]
24. **TODO** Print string reuse revisited [**]
25. **TODO** Implementation without interface [*]
26. **TODO** Implementation with interface [*]
27. **TODO** Implementation with abstracted interface [*]
28. **TODO** Preinter for date [***]
29. **TODO** Refactor arith [****]

1.4.5 TODO 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 TODO 7.5 Mutability - Exercises [10/11]

1. **DONE** mutable fields [*]

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

```
(* defining a record type with a mutable gpa field: *)
type student = {name : string; mutable gpa: float};;

(* create the specified instance *)
let student_rec = {name = "Alice"; gpa = 3.7};;

(* change the gpa as specified *)
student_rec.gpa <- 4.0;;

(* inspect to confirm *)
student_rec
```

2. **DONE** refs [*]

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

   - `bool ref`

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

   - `int list ref`

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

   - `int ref list`

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

3. **DONE** inc fun [*]

   Define a reference to a function as follows:

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

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

   (This is disgusting)

```
let p = ref 0 in
let q = ref 0 in
let r = ref 0 in
while ((!p) < 2)
do (p := !p + 1)
done;
while ((!q) < 5)
do (q := !q + 1)
done;
while ((!r) < 311)
do (r := !r + 1)
done;
(!p) * (!q) * (!r);
```

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 line noise here, this function is like 60% punctuation)

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

let x = ref 0;;

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

!x;;
```

5. **DONE** physical equality [**]

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

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

   - `# 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
```

   - `# 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
```

- `# 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
```

- `# x := 1;;`

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

```
x := 1
```

- `# 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
```

- `# 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
```

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:

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

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

norm [|5.0; 12.0|];;

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

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...|]
```

   This 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 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);;

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

8. **DONE** norm loop [**]

   Modify your implementation of `norm` to use 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);;

norm [|5.0; 12.0|]
```

9. **DONE** normalize loop [**]

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

```
let normalize vect =
  let n = 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;;

let v = [| 3.0; 4.0 |];;

norm v;;
normalize v;;
v;;
norm v;;
```

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 x o" matrix. All matrices are m x n. C'mon now.)

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

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

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

1.4.7 TODO 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 TODO 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 [****]

2 Org Export Configuration

2.1 emacs-lisp setup latex export:

run this before exporting:

2.2 TODO Results export

How to change every source block to export "both", instead of "code". This is not a latex option, but an org option. Adding `:results both` to every source block will work, but there's probably a way to make the change globally.