Beginner Level Practice Sheet*

Topics

- Basic imperative & functional programming
- Ints, Floats, Strings
- Arrays, Lists, Options
- Basic I/Os

Exercise 1 – Ints and Floats

Question 1.1 – Define `cube` ($x \to x^3$) on integers.

Question 1.2 – Define `mean`, the binary average between floats.

Question 1.3 – Define `max` that returns the biggest of its two parameters.

Question 1.4 – Define `minmax`, that takes two arguments and returns them sorted in a pair.

Question 1.5 – Define `posiive_part` that returns the positive part of a float.

Question 1.6 – Compute `sizeof_int`, the size of OCaml integers on your platform. Note that all integer operations, including comparison, are signed in OCaml.

Question 1.7 – Define `ones` that counts the number of bits set to 1 in an integer.

Exercise 2 – Strings

Question 2.1 – Define `isupper s` that returns `true` iff all letters of `s` are capitals.

Question 2.2 – Define `period` that returns a copy of its input string with a period at the end, unless there was already one.

Question 2.3 – Define `string_find c s` that looks for the character `c` in `s` and returns the position of its first occurence if any, (-1) otherwise.

Question 2.4 – DeFiNe `cowboy` ThAt TuRnS The WoRdS Of A StRiNg To CoWbOy CaSe.

Question 2.5 – Define `hangman_init` that takes a word to guess and produces a string of the same length, filled with underscores.

Question 2.6 – Define `hangman_step` that takes a word to guess, a partial result (as produced by `hangman_init`), a guessed character, and replaces underscores that correspond to the character in the partial result. The functions returns true iff at least one occurence has been uncovered.

Question 2.7 – Define `string_forall` that checks that all characters of a string verify a predicate.

Redefine `isupper` with it.

Exercise 3 – Int and Float Lists

Question 3.1 – Define `cubes` elevating a list of integers to the cube.

Question 3.2 – Define `average` that computes the average of a list of floats.

Question 3.3 – Define a `low_pass` filter that takes a list of floats, a threshold and produces a list with exceeding values filtered out.

Question 3.4 – Define a `increasing` predicate that tells if a list is sorted.

Question 3.5 – Define a `bounds` function that takes a list of floats, and produces a pair representing the bounds of the smallest interval that contains all values.

Question 3.6 – Write `sum` that computes the sum of a list of integers.

Question 3.7 – Write `reduce`, a generalization of `sum` to any binary operator, such as `sum l = reduce (+) 0 l`+.

Question 3.8 – Write `succs` that adds 1 to all the elements of a list in a new result list.

Question 3.9 – Write `map`, a generalization of `succs` that applies a function to all the elements of a list and returns the list of results.

Exercise 4 – String lists

Question 4.1 – Define `string_explode` that extracts all characters of a string in a list.

Question 4.2 – Define `string_split c s` that cuts `s` at each occurence of `c` and return the cut substrings in a list.

Question 4.3 – Define `string_concat` that collates all the elements of a list in a big string, inserting a given separator.

Exercise 5 – Arrays

Question 5.1 – Define `bounds` that takes an array of floats and returns a pair representing the smallest interval in which all values fit.

Question 5.2 – Define `normalize` that takes an array of floats and applies an appropriate affine transform so that it fits between 0 and 1.

Question 5.3 – Define `array_rev_in_place` that takes an array and reverses the order of its elements.

Question 5.4 – Define `transpose` that transposes a square 2D matrix in place.

Question 5.5 – Define `string_stats` that produces an histogram as an array indicating for each character code ($[0, 255]$) the number of its occurences.

Question 5.6 – Define `occurences` that produces an histogram as `string_stats`, but instead of working on the full char domain, takes as parameter an array of characters to take into account, and

returns an array or their number of occurences.

For instance, `occurences "aabbccddaa" [| 'a' ; 'b' ; 'c' |]` will return `[| 4 ; 2 ; 2 |]`.

Exercise 6 – Options

Question 6.1 – Define `string_find c s` that looks for the character `c` in `s` and returns the position of its first occurence, if any.

Question 6.2 – Write `hd` and `tl` that return the head and tail of a list, if it is not empty.

Question 6.3 – Write `option_default` that returns the contents of an optional value if present, or a given default otherwise.

Question 6.4 – Wrtie `option_map` that applies a function on an optional value, if present, and returns the result as an option.

Question 6.5 – Write `present` extracting all the present values of a list of options.

Question 6.6 – Write `nth` that access a element in a list from its index, if it exists.

Exercise 7 – Basic I/Os

Question 7.1 – Define `print_string_list` that prints each string in a list on a new line. Write a version with `List.iter` and another with `while`.

Question 7.2 – Define `print_banner` that takes a console width, a string, and prints it centered, in a box made of characters `'-'`, `'|'` and `'-'`. The output of `print_banner 70 "HELLO WORLD"` should be as follows.

```
                                +------------+
                                |HELLO WORLD|
                                +------------+
```

Question 7.3 – Write `bar_graph` that takes an array of floats between 0.0 and 1.0, a maximum height, and displays vertical bars using `'#'` signs in the terminal, leaving a blank space between two bars.

Question 7.4 – Using `bar_graph`, `occurences` and `normalize`, write `display_occurences` that takes a string, an array of characters, and displays a legended bargraph, as in the example below.

```
let () =
  display_occurences "Voulez-vous␣coucher␣avec␣moi,␣ce␣soir␣?"
    [| 'a' ; 'b' ; (* ... *) ; 'x' ; 'y' ; 'z' |];;
```

That should output:

```
                              #
    #   #                     #
    #   #                     #
    #   #       #             #     # #   # #
#   #   #     # #     # #     #     # #   # #       #
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Beginner Level Practice Sheet

OLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTION

Solution to question 1.1

```
let cube x = x * x * x
```

Solution to question 1.2

```
let mean x y = (x +. y) /. 2.
```

Solution to question 1.3

```
let max x y = if x > y then x else y
```

Solution to question 1.4

```
let minmax x y = if x <= y then (x, y) else (y, x)
let minmax x y = (min x y, max x y)
```

Solution to question 1.5

```
let positive_part x = max 0 x
let positive_part = max 0
```

Solution to question 1.6

```
let sizeof_int =
  let rec shift n =
    if (n + n) <= n then 1 else 1 + shift (n + n)
  in shift 1 + 1
```

Solution to question 1.7

```
let rec ones n =
  if n = 0 then 0 else n land 1 + ones (n lsr 1)
```

Solution to question 2.1

```
let isupper s =
  let ret = ref true in
    for i = 0 to String.length s - 1 do
      if 'a' <= s.[i] && s.[i] <= 'z' then
        ret := false
    done;
    !ret
```

Solution to question 2.2

```
let period s =
  if s.[String.length s - 1] = '.' then s
  else s ^ "."
```

Solution to question 2.3

```
let string_find c s =
  let i = ref 0 in
    while !i < String.length s && s.[!i] <> c do
      incr i
    done;
    if !i < String.length s then !i
    else (-1)
```

Solution to question 2.4

```
let cowboy s =
  let jr = ref 0 in
  for i = 0 to String.length s - 1 do
    match s.[i] with
    | 'a' .. 'z' ->
      if !jr mod 2 = 0 then
        s.[!jr] <- Char.(chr (code s.[!jr] - code 'a' + code 'A')) ;
      incr jr
    | 'A' .. 'Z' ->
      if !jr mod 2 = 1 then
        s.[!jr] <- Char.(chr (code s.[!jr] - code 'A' + code 'a')) ;
      incr jr
    | _ ->
      jr := 0
  done
```

Solution to question 2.5

```
let hangman_init s = String.make (String.length s) '_'
```

Solution to question 2.6

```
let hangman_step s g c =
  let res = ref false in
  for i = 0 to String.length g do
    if s.[i] = c && g.[i] = '_' then begin
      g.[i] <- c ;
      res := true
    end
  done ;
  !res
```

Solution to question 2.7

```
let string_forall f s =
  let i = ref 0 in
    while !i < String.length s && f s.[!i] do incr i done;
    !i = String.length s
let uppercase = string_forall (fun c -> c < 'a' || 'z' < c)
```

Solution to question 3.1

```
let cubes l = List.map cube l
```

Solution to question 3.2

```
let average nums =
  List.fold_left (+.) 0. nums /. float (List.length nums)
```

Solution to question 3.3

```
let rec low_pass l t = match l with
  | [] -> []
  | v :: vs ->
    if v > t then
      low_pass vs t
    else
      v :: low_pass vs t
```

Solution to question 3.4

```
let rec increasing l = match l with
  | [] -> true
  | v1 :: v2 :: vs ->
    v1 <= v2 && increasing (v2 :: vs)
```

Solution to question 3.5

```
let rec bounds l t = match l with
  | [] -> (infinity, -. infinity)
  | v :: vs ->
    let (min, max) = bounds vs in
    (if v < min then v else min,
     if v > max then v else max)
```

Solution to question 3.6

```
let rec sum = function
  | h :: t -> h + sum t
  | [] -> 0
```

Solution to question 3.7

```
let rec reduce neutral f = function
  | h :: t -> f h (reduce neutral f t)
  | [] -> neutral
```

Solution to question 3.8

```
let rec succs f = function
  | h :: t -> f h :: succs f t
  | [] -> []
```

Solution to question 3.9

```
let rec map f = function
  | h :: t -> f h :: map f t
  | [] -> []
```

Solution to question 4.1

```
let string_explode s =
  let list = ref [] in
  let start = ref 0 in
```

```
    for i = String.length s - 1 downto 0 do
      list := s.[i] :: !list;
    done;
    !list
```

Solution to question 4.2

```
let string_split c s =
  let list = ref [] in
  let start = ref 0 in
    for i = 0 to String.length s - 1 do
      if s.[i] = c then (
        list := String.sub s !start (i - !start) :: !list;
        start := i + 1
      )
    done;
    list :=
      String.sub s !start (String.length s - !start)
      :: !list;
    List.rev !list
```

Solution to question 4.3

```
let rec string_concat l s =
  match l with
  | [] -> ""
  | e :: [] -> e
  | e :: e' :: r -> e ^ s ^ string_concat (e' :: r)
```

Solution to question 5.1

```
let bounds a =
  let min = ref infinity
  and max = ref (-. infinity) in
  for i = 0 to Array.length a - 1 do
    min := if a.(i) < !min then a.(i) else !min ;
    max := if a.(i) > !max then a.(i) else !max
  done ;
  (!min, !max)
```

Solution to question 5.2

```
let normalize a =
  let (min, max) = bounds a in
  Array.map (fun v -> (v -. min) /. (max -. min)) a
```

Solution to question 5.3

```
let array_rev_in_place a =
  let last = Array.length a - 1 in
    for i = 0 to last / 2 do
      let x = a.(i) in
        a.(i) <- a.(last - i);
        a.(last - i) <- x
    done
```

Solution to question 5.4

```
let transpose m =
  for i = 1 to Array.length m - 1 do
    for j = 0 to i - 1 do
      let tmp = m.(i).(j) in
      m.(i).(j) <- m.(j).(i) ;
      m.(j).(i) <- tmp
    done
  done
```

Solution to question 5.5

```
let string_stats s =
  let hist = Array.make 0 255 in
  for i = 0 to String.length s - 1 do
    hist.(Char.code s.[i]) <- hist.(Char.code s.[i]) + 1
  done;
  hist
```

Solution to question 5.6

```
let occurences s chars =
  let res = Array.make (Array.length chars) 0 in
  for si = 0 to String.length s - 1 do
    for ai = 0 to Array.length chars - 1 do
      if chars.(ai) = s.[si] then
        res.(ai) <- res.(ai) + 1
    done
  done ;
  res
```

Solution to question 6.1

```
let string_find c s =
  let i = ref 0 in
    while !i < String.length s && s.[!i] <> c do
      incr i
    done;
    if !i < String.length s then Some !i
    else None
```

Solution to question 6.2

```
let head l = match l with h :: _ -> Some h | [] -> None
let tail l = match l with _ :: t -> Some t | [] -> None
```

Solution to question 6.3

```
let option_default opt default =
  match opt with
    | Some x -> x
    | None -> default
```

Solution to question 6.4

```
let option_map f opt =
  match opt with
    | Some x -> Some (f x)
    | None -> None
```

Solution to question 6.5

```
let rec present = function
  | Some x :: t -> x :: present t
  | None :: t -> present t
  | [] -> []
```

Solution to question 6.6

```
let rec list_nth n = function
  | h :: _ when n = 0 -> Some h
  | _ :: t -> list_nth (n - 1) t
  | [] -> None
```

Solution to question 7.1

```
let print_string_list l = List.iter print_endline l (* ou *)
let print_string_list l =
  let r = ref l in
    while !r <> [] do
      print_endline (List.hd !r);
      r := List.tl !r
    done
```

Solution to question 7.2

```
let print_banner w s =
  let len = String.length s in
  for i = 0 to (w - len) / 2 do
    print_string "␣" ;
  done ;
  print_string "+" ;
  for i = 0 to len - 1 do
    print_string "-" ;
  done ;
  print_string "+" ; print_newline () ;
  for i = 0 to (w - len) / 2 do
    print_string "␣" ;
  done ;
  print_string "|" ; print_string s ; print_string "|" ; print_newline () ;
  for i = 0 to (w - len) / 2 do
    print_string "␣" ;
  done ;
  print_string "+" ;
  for i = 0 to len - 1 do
    print_string "-" ;
  done ;
  print_string "+" ; print_newline ()
```

Solution to question 7.3

```
let bar_graph stats height =
  let d = 1. /. float height in
  for y = 0 to height - 1 do
    for x = 0 to Array.length stats - 1 do
      if stats.(x) >= float (height - y) *. d then
```

```
      print_string "# "
    else
      print_string "  "
  done ;
  print_newline ()
done
```

Solution to question 7.4

```
let display_occurences s chars =
  bar_graph (normalize (Array.map float (occurences s chars))) 5 ;
  for i = 0 to Array.length chars - 1 do
    print_char chars.(i) ; print_char ' '
  done ;
  print_newline ()
```