

# Programming Survey

Programmation Fonctionnelle

Année 2014-2015

There are two sets of three problems each. The first set 1 requires you to write fresh programs. The second set 2 gives you solutions to problems and asks you to identify which ones you prefer and why.

## 1 Programming Problems

### 1.1 Palindrome Detection Modulo Spaces and Capitalization

A palindrome is a string with the same letters in each of forward and reverse order (ignoring capitalization). Design a program called `is_palindrome` that consumes a string and produces a boolean indicating whether the string with all spaces and punctuation removed is a palindrome. Treat all non-alphanumeric characters (i.e., ones that are not digits or letters) as punctuation.

Examples:

- `is_palindrome("a man, a plan, a canal: Panama")` is true
- `is_palindrome("abca")` is false
- `is_palindrome("yes, he did it")` is false

Ocaml documentation on strings: <http://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html>

### 1.2 Sum Over Table

Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths. Design a program `sum_largest` that consumes a table of numbers and produces the sum of the largest item from each row. Assume that no row is empty.

Example:

```
sum_largest [ [1; 7; 5; 3]; [20]; [6; 9] ]
```

is the result of  $7 + 20 + 9$ , which is 36.

### 1.3 Adding Machine

Design a program called `adding-machine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Example:

```
adding-machine [1; 2; 0; 7; 0; 5; 4; 1; 0; 0; 6]
```

is `[3; 7; 10]`.

## 2 Review Problems

Below you are given problem statements followed by multiple solutions to each problem. Assume that the solutions are correct; ignore any small deviations in behavior. Also assume that any missing helper functions are defined in the obvious way. For instance, we rely on the `sum_of` function that takes a list of floats and returns their sum.

```
let sum_of l = List.fold_right (fun x acc -> x +. acc) l 0.;;
```

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

We also rely on the `max_of` function that returns the biggest element of a list.

```
let max_of l = List.fold_right (fun x acc -> max x acc) l 0.;;
```

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

Finally, ignore stylistic differences in naming. Instead, focus on the structure of the solutions.

For each problem, rank the solutions in order (from most to least) of your preference. Explain why you picked that ordering. You are allowed to have ties.

The solutions are labeled A, B, etc. Indicate your ordering with the labels and `>`, using commas for ties. For instance, the ordering

`B > A, C > D`

means you liked B the most, followed by A and C (tied), followed by D.

Remember to explain your choice!

### 2.1 Rainfall

Design a program called `rainfall` that consumes a list of real numbers representing daily rainfall readings. The list may contain the number `-999`, indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first `-999`, (if it shows up). There may be negative numbers other than `-999`, in the list (representing faulty readings). Assume that there is at least one non-negative number before `-999`.

Example:

```
rainfall [1.; -2.; 5.; -999.; 8.]
```

is `3`.

### 2.1.1 Solution A

```
let rainfall l =
  let rec helper rds total days =
    match rds with
    | [] -> total /. (float_of_int days)
    | f :: r ->
      if f = -999. then total /. (float_of_int days)
      else if f < 0. then helper r total days
      else helper r (total +. f) (days + 1)
  in
  helper l 0. 0;;
```

val rainfall : float list -> float = <fun>

### 2.1.2 Solution B

```
let rainfall l =
  let rec sum_rfs l =
    match l with
    | [] -> 0.
    | f :: r ->
      if f = -999. then 0.
      else if f < 0. then sum_rfs r
      else f +. sum_rfs r
  in
  let rec count_days l =
    match l with
    | [] -> 0
    | f :: r ->
      if f = -999. then 0
      else if f < 0. then count_days r
      else 1 + count_days r
  in
  let total = sum_rfs l in
  let days = count_days l in
  total /. (float_of_int days);;
```

val rainfall : float list -> float = <fun>

### 2.1.3 Solution C

```
let rainfall l =
  let rec cleanse l =
    match l with
    | [] -> []
    | f :: r ->
      if f = -999. then []
      else if f < 0. then cleanse r
      else f :: (cleanse r)
  in
  let actual = cleanse l in
  let total = sum_of actual in
  let days = List.length actual in
  total /. (float_of_int days);;
```

val rainfall : float list -> float = <fun>

## 2.2 Length of Triples

Design a program called `max_triple_length` that consumes a list of strings and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings.

Example:

```
max_triple_length ["a"; "bb"; "c"; "dd"]
```

is 5.

We recall that `List.hd l` returns the head (the first element) of the list `l`, and `List.tl l` returns the tail (the rest) of the list `l`.

### 2.2.1 Solution A

```
let max_triple_length l =
  let rec break_into_triples l =
    ((List.hd l), (List.hd (List.tl l)), (List.hd (List.tl (List.tl l)))) ::
    (if (List.tl (List.tl (List.tl l))) = [] then [] else break_into_triples (List.tl l))
  in
  let triple_lengths =
    List.map (fun (e1,e2,e3) -> String.length e1 + String.length e2 + String.length e3)
      (break_into_triples l)
  in
  max_of triple_lengths;;
```

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

### 2.2.2 Solution B

```
let max_triple_length l =
  let rec break_into_triples l =
    ((List.hd l), (List.hd (List.tl l)), (List.hd (List.tl (List.tl l)))) ::
    (if (List.tl (List.tl (List.tl l))) = [] then [] else break_into_triples (List.tl l))
  in
  let triples_as_nums = List.map String.length l in
  let triple_lengths =
    List.map (fun (s1,s2,s3) -> s1 + s2 + s3) (break_into_triples triples_as_nums)
  in
  max_of triple_lengths;;
```

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

### 2.2.3 Solution C

```
let max_triple_length l =
  let l = List.map String.length l in
  let rec helper l max_so_far prev_2 prev_1 =
    match l with
    | [] -> max_so_far
    | f :: r ->
      let prev_3 = prev_2 + f in
      helper r (max prev_3 max_so_far) (prev_1 + f) f
  in
  helper (List.tl (List.tl (List.tl l)))
  (List.hd l + List.hd (List.tl l) + List.hd (List.tl (List.tl l)))
  (List.hd (List.tl l) + List.hd (List.tl (List.tl l)))
  (List.hd (List.tl (List.tl l)));;
```

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

## 2.3 Shopping Discount

An online clothing store applies discounts during checkout. A shopping cart is a list of the items being purchased. Each item has a name, a string like "shoes", and a price, a real number like 12.50. Design a program called `checkout` that consumes a shopping cart and produces the total cost of the cart after applying the following two discounts:

- if the cart contains at least 100. worth of shoes, take 20% off the cost of all shoes (match only items whose exact name is "shoes")
- if the cart contains at least two hats, take 10. off the total of the cart (match only items whose exact name is "hat")

Assume the cart is represented as follows:

```
type cart_item = { name : string; cost : float; }
and cart = cart_item list
```

Example:

```
checkout [{name="shoes"; cost=25.}; {name="bag"; cost=50.};
          {name="shoes"; cost=85.}; {name="hat"; cost=15.}] ;;
```

is 153.

### 2.3.1 Solution A

```
let checkout c =
  let shoes = List.filter (fun ci -> ci.name = "shoes") c in
  let shoe_cost = sum_of (List.map (fun ci -> ci.cost) shoes) in
  let shoe_discount = if shoe_cost >= 100. then shoe_cost *. 0.20 else 0. in
  let hats = List.filter (fun ci -> ci.name = "hat") c in
  let hat_count = List.length hats in
  let hat_discount = if hat_count >= 2 then 10. else 0. in
  let init_cost = sum_of (List.map (fun ci -> ci.cost) c) in
  init_cost -. shoe_discount -. hat_discount ;;
```

```
val checkout : cart_item list -> float = <fun>
```

### 2.3.2 Solution B

```
let checkout c =
  let rec helper ct total shoe_cost hat_count =
    match ct with
    | [] ->
      let shoe_discount = if shoe_cost >= 100. then shoe_cost *. 0.20 else 0. in
      let hat_discount = if hat_count >= 2 then 10. else 0. in
      total -. shoe_discount -. hat_discount
    | f :: r ->
      let new_total = total +. f.cost in
      if f.name = "shoes" then helper r new_total (shoe_cost +. f.cost) hat_count
      else if f.name = "hats" then helper r new_total shoe_cost (hat_count + 1)
      else helper r new_total shoe_cost hat_count
  in
  helper c 0. 0. 0
```

```
val checkout : cart_item list -> float = <fun>
```