# Algorithmics
# Correction Midterm Exam #1

UNDERGRADUATE $1^{st}$ YEAR S1 – EPITA

***Solution 1*** **(Abstract Types: Recursive lists – *5 points*)**

1. The operation `search` is defined only when the searched element exists. Therefore, it is a precondition. Then we have the three axioms applying the observer `ispresent` to the internal operations `emptylist` and `cons`. In order: the element $e$ does not exist in an empty list, the element $e$ exists in a list in which it is equal to the first element and otherwise... try again (it may exist in the tail of the list). Then the axiom explaining that the box returned by `search(e,`$\lambda$`)` is the one which contains $e$.

   **PRECONDITIONS**

   $search(e, \lambda)$ **is-defined-iaoi** $ispresent(e,\lambda) = \text{true}$

   **AXIOMS**

   $ispresent\ (e, emptylist) = \text{false}$
   $e = e' \Rightarrow ispresent\ (e, cons(e', \lambda)) = \text{true}$
   $e \neq e' \Rightarrow ispresent\ (e, cons(e', \lambda)) = ispresent\ (e, \lambda)$
   $contents(search(e, \lambda)) = e$

2. Two axioms suffice. The first says that the concatenation result of an empty list and a list $\lambda$ is the list $\lambda$, which means that the elements of the second list are retained in order and number. The second axiom explains that we also keep in order and number the elements of the first list. How? Showing that if the concatenation is done before or after building (cons) the list, the result is the same, which means that the concatenation modifies neither the order nor the elements.

   **AXIOMS**

   $concatenate\ (emptylist, \lambda 2) = \lambda 2$
   $concatenate\ (cons(e, \lambda), \lambda 2) = cons(e,\ concatenate\ (\lambda, \lambda 2))$

***Solution 2*** **(is_image – *4 points*)**

**Specifications:**

The function `is_image`:

- takes a one-argument function, $f$, and two lists, $[a_1; a_2; \cdots; a_n]$ and $[b_1; b_2; \cdots; b_n]$, as parameters.
- checks whether for all pairs of elements ($a_i$, $b_i$) that $b_i$ is the image of $a_i$ under $f$.
  That is, it returns $f\ a_1 = b_1$ && $f\ a_2 = b_2$ && $\cdots$ && $f\ a_n = b_n$.
- If a pair such that $f\ a_i \neq b_i$ is found, it returns false. Otherwise, it raises `Invalid_argument` if the two lists have different lengths.

```
# let rec is_image f list1 list2 =
   match (list1, list2) with
       ([], []) -> true
     | ([], _) | (_, []) -> invalid_arg "different lengths"
     | (e1::l1, e2::l2) -> f e1 = e2 && is_image f l1 l2 ;;

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

***Solution 3*** **(How many? – *4 points*)**

1. **Specifications:**
   The function `how_many` takes a boolean function $f$ and a list $[a_1; a_2; \cdots; a_n]$ as parameters and returns the number of values $a_i$ such that $f(a_i)$ is true.

```
# let rec how_many f = function
            [] -> 0
          | e::l -> (if f e then 1 else 0) + how_many f l

# let rec how_many f = function
            [] -> 0
          | e::l when f e -> 1 + how_many f l
          | _::l          -> how_many f l

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

2. **Specifications:**
   The function `count_multiples` $n$ $l$ returns the number of multiples of $n$ in the list $l$.

```
# let count_multiples n = how_many (function x -> x mod n = 0) ;;

# let count_multiples n l =
    let div a = a mod n = 0 in how_many div l ;;

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

---

***Solution 4*** **(Insertion at the rank $i$ – *5 points*)**

**Spécifications :** La fonction `insert_nth` $x$ $i$ $l$ insère l'élément $x$ au rang $i$ dans la liste $l$. Une exception est déclenchée si $i \leq 0$ ou est supérieur à la longueur de la liste + 1.

```
# let insert_nth  x i list =
    if i < 1 then
      invalid_arg "negative rank"
    else
      let rec insert = function
          (1, list)    -> x :: list
        | (_, [])   -> failwith "out of bound"
        | (i, e::q) -> e :: insert(i-1, q)
      in
        insert (i, list);;

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

---

***Solution 5*** **(Evaluations – *3 points*)**

```
# let rec decode = function
     [] -> []
   | (1, e)::list -> e::decode list
   | (nb, e)::list -> e::decode ((nb-1, e)::list);;
 val decode : (int * 'a) list -> 'a list = <fun>

# decode [(6, "grr")];;
 - : string list = ["grr"; "grr"; "grr"; "grr"; "grr"; "grr"]

# decode [(1, 'a'); (3, 'b'); (1, 'c'); (1, 'd'); (4, 'e')];;
```

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

# let encode list =
    let rec encode_rec (nb, cur) = function
        []   -> [(nb, cur)]
      | e::list -> if e = cur then
          encode_rec (nb+1, cur) list
        else
          (nb, cur)::encode_rec (1, e) list
    in
      match list with
          []   -> []
        | e::l -> encode_rec (1, e) l;;
 val encode : 'a list -> (int * 'a) list = <fun>

# encode [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];;
 - : (int * int) list = [(10, 0)]

# encode ['b';'b';'b'; 'c'; 'a';'a'; 'e';'e';'e';'e'; 'd';'d'];;
 - : (int * char) list = [(3, 'b'); (1, 'c'); (2, 'a'); (4, 'e'); (2, 'd')]
```