Algorithmique
Correction Contrôle n° 1

INFO-SUP S1 – EPITA

Solution 1 **(Types Abstraits : Listes récursives – *5 points*)**

1. L'opération `rechercher` n'est définie que si l'élément recherché existe, d'où précondition. Ensuite les trois axiomes appliquant l'observateur `est-présent` aux opérations internes `liste-vide` et `cons`. Dans l'ordre : l'élément e n'existe pas dans une liste-vide, l'élément e existe dans une liste dont il est égal au premier élément et dans le cas contraire... Essaie encore (il existe peut-être dans la liste restante). Enfin l'axiome expliquant que la place retournée par `rechercher(e,`λ`)` est celle qui contient e.

   PRÉCONDITIONS

   rechercher(e, λ) **est-défini-ssi** *est-présent*(e,λ)= vrai

   AXIOMES

   est-présent $(e, listevide) =$ faux
   $e = e' \Rightarrow$ *est-présent* $(e, cons(e', \lambda)) =$ vrai
   $e \neq e' \Rightarrow$ *est-présent* $(e, cons(e', \lambda)) =$ *est-présent* (e, λ)
   $contenu(rechercher(e, \lambda)) = e$

2. Deux axiomes suffiront, le premier explique que le résultat de la concaténation d'une liste vide avec une liste λ est la liste λ, ce qui signifie que l'on conserve les éléments en ordre et nombre de la deuxième liste. Le second axiome explique que l'on conserve aussi les éléments en ordre et nombre de la première liste. Comment ? En montrant que si l'on fait la concaténation avant ou après la construction (cons), le résultat est le même, ce qui signifie que la concaténation ne modifie ni l'ordre, ni les éléments.

   AXIOMES

   concaténer $(listevide, \lambda 2) = \lambda 2$
   concaténer $(cons(e, \lambda), \lambda 2) = cons(e,$ *concaténer* $(\lambda, \lambda 2))$

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

Spécifications :

La fonction `is_image` :
- prend en paramètre une fonction à un paramètre : `f` ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
- vérifie pour toutes les paires (a_i, b_i) que b_i est l'image de a_i par f.
  C'est à dire elle calcule $f\ a_1 = b_1$ && $f\ a_2 = b_2$ && $\cdots$ && $f\ a_n = b_n$.
- Si elle trouve une paire telle que $f\ a_i \neq b_i$, elle retourne faux. Sinon, elle déclenche une exception `Invalid_argument` si les deux listes sont de longueurs différentes.

```
# 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 **(Combien ? – *4 points*)**

1. **Spécifications :**
   La fonction `how_many` prend en paramètre une fonction booléenne : f ainsi qu'une liste : $[a_1; a_2; \cdots; a_n]$. Elle recherche dans la liste les valeurs a_i telle que $f(a_i)$ soit vrai et retourne le nombre de valeurs trouvées.

```
# 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. **Spécifications :**
   La fonction `count_multiples` n l retourne le nombre de multiples de n dans la liste 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 à la $i^{ème}$ place – *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 **(Évaluations – *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')]
```