Algorithmique
Correction Contrôle n° 1

INFO-SUP S1 – EPITA

Solution 1 **(Un peu de cours... –** ***4 points*****)**

1. Une opération interne retourne un type défini.
2. Une opération servant à spécifier le domaine de définition d'un autre est une opération auxiliaire
3. Les problèmes qui se posent lors de la conception de l'ensemble des axiomes sont la complétude et la consistance.
4. Les zones qui composent la signature d'un type abstrait sont les zones TYPES, UTILISE et OPERATIONS.
5. Nous écrivons des axiomes en appliquant des observateurs aux opérations internes

---

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

```
# let rec is_dominoes = function
    | [] | _::[] -> true
    | (_, b)::(c, d)::l -> (b = c) && is_dominoes ((c, d)::l) ;;

# let rec is_dominoes = function
    | [] | [_] -> true
    | (_, b)::(c, _)::_ when b <> c -> false
    | _::(c, d)::l -> is_dominoes ((c, d)::l) ;;

# let rec is_dominoes = function
    | [] | _::[] -> true
    | (_, b)::(c, d)::l -> if b <> c then false
                           else is_dominoes ((c,d)::l) ;;

  val is_dominoes : ('a * 'a) list -> bool = <fun>
```

Solution 3 **(Suppression du $i^{ème}$ – *5 points*)**

Spécifications :

La fonction `supprime_ieme` prend en paramètres un entier i et une liste l. Elle retourne la liste privée de son i^{ème}élément (le premier élément étant à la position 1). Une exception est déclenchée si la liste est trop courte, ou si l'entier i est négatif ou nul.

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

# let remove_nth i list =
    let rec del i = function
        []   -> failwith "out of bound"
      | e::q -> if i = 1 then
                  q
                else
             e :: del (i-1) q
    in
      if i < 1 then
        failwith "negative rank"
      else
        del i list ;;

# let remove_nth i list =
    if i < 1 then
      failwith "negative rank"
    else
      let rec del i list = match list with
       []   -> failwith "out of bound"
        | _::q when i = 1 -> q
        | e::q -> e::del (i-1) q
      in
        del i list ;;

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

Solution 4 **(for_all2 – *5 points*)**

1. **Spécifications :**
   La fonction `for_all2` :
   - prend en paramètre une fonction de test (un prédicat) à deux paramètres : `p` ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
   - calcule $p\ a_1\ b_1$ && $p\ a_2\ b_2$ && $\cdots$ && $p\ a_n\ b_n$.
   - déclenche une exception `Invalid_argument` si les deux listes sont de longueurs différentes.

```
# let rec for_all2 p list1 list2 =
    match (list1, list2) with
        ([],[]) -> true
      | (_,[]) | ([],_) -> invalid_arg "for_all2: different lengths"
      | (a::l1,b::l2) -> p a b && for_all2 p l1 l2 ;;

# let rec for_all2 p list1 list2 =
    match (list1, list2) with
        ([], []) -> true
      | (_, []) | ([], _) -> invalid_arg "for_all2: different lengths"
      | (a::l1, b::l2) when p a b -> for_all2 p l1 l2
      | _ -> false ;;

# let rec for_all2 p list1 list2 =
    match (list1, list2) with
        ([], []) -> true
      | (_, []) | ([], _) -> invalid_arg "for_all2: different lengths"
      | (a::l1, b::l2) -> if p a b then
                            for_all2 p l1 l2
                          else
                            false ;;
 val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool = <fun>
```

2. **Spécifications :**
   La fonction `almost` prend deux listes en paramètres. Elle retourne *vrai* si elles sont "quasi"-identiques (pas plus de 1 de différence entre 2 éléments), *faux* sinon. Elle déclenche une exception si les deux listes sont de longueurs différentes.

```
# let almost list1 list2 =
    let near x y = (y - 2 < x) && (x < y + 2)
    in
      for_all2 near list1 list2 ;;

 val almost : int list -> int list -> bool = <fun>
```

Solution 5 **(Mystery – 2 points)**

```
# let mystery a b =
    let rec what = function
        ([], _) -> true
      | (_, []) -> false
      | (e::l1, f::l2) -> (e = f) && what (l1, l2)
    in
    let rec is_that x y = match y with
        [] -> 0
      | e::q -> (if what (x, y) then 1 else 0) + (is_that x q)
    in
      is_that a b ;;

val mystery : 'a list -> 'a list -> int = <fun>

# mystery [1; 2] [1; 2];;
- : int = 1

# mystery [1; 2] [1; 1; 2; 3; 3; 1; 2; 3];;
- : int = 2

# mystery [1; 2] [2; 1];;
- : int = 0
```