Devoir d'Informatique n°5

1er février 2023

*
* *

Durée de l'épreuve : 3 heures

L'usage de tout dispositif électronique est interdit.

Consignes

Pour répondre à une question, il vous est permis de réutiliser le résultat d'une question antérieure même si vous n'avez pas réussi à établir ce résultat.

Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier la correction ou la terminaison de cette fonction, ou de la commenter.

Vous attacherez la plus grande importance à la clarté, à la précision et à la concision de la rédaction.

Si vous repérez ce qu'il vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez la composition en expliquant les éventuelles initiatives que vous aurez pris.

Vous devrez traiter les questions de programmation dans le langage OCaml.

Les trois parties du sujet sont indépendantes

OCaml

Question 1. Trouver des expressions OCaml ayant exactement les types suivants :

1. `[([1;2] , "bla")] : (int list * string) list`
2. `fun x -> x : 'a -> 'a`
3. `fun x -> x + 1 : int -> int`
4. `fun x -> 365 : 'a -> int`
5. `fun (x, y) -> x : ('a * 'b) -> 'a`
6. `fun x y -> (x, y) : 'a -> 'b -> 'a * 'b`
7. `fun f x -> f x : ('a -> 'b)-> 'a -> 'b`
8. `fun x y -> x : 'a -> 'b -> 'a`
9. `fun f (x, y) -> f x y : ('a -> 'b -> 'c)-> ('a * 'b -> 'c)`
10. Il y avait une coquille dans le parenthésage, le type attendu devait plutôt être `(('a * 'b) -> 'c)-> ('a -> 'b -> 'c)`. On peut alors proposer `fun f x y -> f (x, y)`

Question 2. Fonction `zip: ('a list * 'b list) -> ('a * 'b) list` :

```
1  let rec zip (l1, l2) =
2    match (l1, l2) with
3    | [], [] -> []
4    | x1 :: q1, x2 :: q2 -> (x1, x2) :: zip (q1, q2)
5    | _ -> failwith " les listes ne sont pas de même taille"
```

Question 3. Maximum d'une liste (de type 'a list -¿ 'a) :

```
1  let rec max_liste l =
2    match l with
3    | [] -> failwith " liste vide"
4    | [x] -> x
5    | x :: q -> max x (max_liste q)
```

On aurait aussi pu faire une version renvoyant une option.

Question 4. Fonction `empaqueter: float list -> float list list` : on propose de passer par une fonction auxiliaire `empaqueter_debut (l: float list) (accu: float list) (s: float)` qui stocke dans accu les éléments du paquet en cours de construction en mémoire les éléments du paquet actuel, et dans s la somme de ces éléments.

```
1  let empaqueter (l: float list) : float list list =
2     let rec empaqueter_debut (l: float_list ) (accu: float list) (s: float) : float list list =
3      match l with
4      | [] -> [List.rev accu]
5      | x :: q ->
6        if x +. s > 1. then
7          List.rev accu :: empaqueter_debut q [x] x
8        else
9          empaqueter_debut q (x :: accu) (s +. x)
10   in empaqueter_debut l [] 0.
```

En OCaml, les listes doivent être homogènes : tous les éléments d'une liste doivent être de même type. On se propose d'implémenter les ***listes imbriquées***, qui permettent de représenter des objets comme [2; [3; 4]; [[5]; [6]; 7]] (qui n'existent donc pas en OCaml) où cohabitent des entiers, des listes d'entiers, des listes de listes d'entiers, etc... On utilise le type suivant :

```
type 'a nested_list =
    | One of 'a (* élément simple *)
    | Many of 'a nested_list list ;; (* liste *)
```

Par exemple, la liste imbriquée $[1; [2]; 3]$ sera représentée par la valeur `Many [One 1; Many [One 2]; One 3 ]`, et la liste imbriquée $[[]; 3; [[2]]$ par l'expression `Many [ Many []; One 3; Many [Many [One 2]] ]`

Question 5.

Fonction `nested_map: ('a -> 'b)-> 'a nested_list -> 'b nested_list` :

```
let rec nested_map (f: 'a -> 'b) (nl: 'a nested_list ) : 'b nested_list =
  match nl with
  | One x -> One (f x)
  | Many l -> Many (List.map (nested_map f) l)
```

Question 6.

Fonction `flatten: 'a nested_list -> 'a list` :

```
let rec flatten (nl: 'a nested_list ) : 'a list =
  match nl with
  | One x -> [x]
  | Many l -> List.concat (List.map flatten l)

(* solution sans utiliser de concaténation, en faisant essentiellement un
   parcours préfixe avec pile *)
let flatten nl =
  let rec flatten_append (p: 'a nested_list list ) (res: 'a list ) =
    match p with
    | [] -> List.rev res
    | One x :: q -> flatten_append q (x :: res)
    | Many [] :: q -> flatten_append q res
    | Many (nl' :: l) :: q -> flatten_append (nl' :: Many l :: q) res (* on empile le 1er enfant *)
  in flatten_append [nl] []
```