université de BORDEAUX	**Année universitaire :** 2025-2026 **Parcours :** Licence Informatique 3e année **UE :** Programmation fonctionnelle UE 4TIN514U **Épreuve :** Examen de Programmation fonctionnelle **Date :** mardi 16 décembre 2025 9 :00 – 10 :30 **Durée :** 1h30 **Documents interdits.**	**Collège Sciences et Technologies**

Dans tout l'examen, on essaiera

- **d'utiliser de préférence les fonctions du module `List`,**
- **d'écrire des fonctions récursives terminales.**

Exercice 1 *(5pts)* Soit le fichier `point.ml` présenté Figure 1 page 1.

En supposant qu'on soit positionné dans le même répertoire que le fichier `point.ml`,

1. donner une commande permettant de compiler le fichier `point.ml` pour obtenir le fichier objet `point.cmo`.

   En supposant lancée une boucle d'interaction `OCaML` (`utop` par exemple),

2. donner une suite d'instructions et expressions permettant
   - de charger le module `Point`,
   - de créer deux points de type `point`,
   - de calculer la distance entre ces deux points.

```
type point = P of float * float

(* constructeur *)
let make_point x y = P(x, y)

(* accesseurs *)
let p_x p = let P(x,_) = p in x
let p_y p = let P(_,y) = p in y

let p_origin = make_point 0. 0.
let p_i = make_point 0. 1.

let dist x y x' y' =
  let dx, dy = abs_float(x -. x'), abs_float(y -. y') in
  sqrt (dx *. dx +. dy *. dy)

let p_dist (p1 : point) (p2 : point) = dist (p_x p1) (p_y p1) (p_x p2) (p_y p2)

let p_abs p = p_dist p_origin p
```

FIG. 1 : Fichier `point.ml`

3. Écrire une fonction `somme_distances p0 points` de type `point -> point list -> float` qui retourne la somme des distances des points de la liste `points` au point `p0`.

Exemple :

```
# somme_distance p_origin [make_point 0. 4.; make_point 10. 0.; make_point 3. 4.];;
- : float = 19.
```

4. Écrire une fonction `points_dans_disque centre rayon points` de type

`point -> float -> point list -> point list` qui retourne la liste des points de la liste `points` qui sont dans le disque de rayon `rayon` centré au point `centre`.

Exemple :

```
# dans_disque p_origin 10. [make_point 3. 4.; make_point 10. 4.; make_point 5. 5.];;
- : point list = [P (3., 4.); P (5., 5.)]
```

Exercice 2 *(5pts)* Soit la fonction `mystery` suivante :

```
let rec mystery x y =
  if x = 0 then 0
  else y + mystery (x - 1) y
```

5. Que retourne l'appel `mystery 4 6` ?
6. Que retourne `mystery` en fonction de `x` et de `y` ?
7. Quelle est la complexité de la fonction `mystery` ?
8. Donner une version logarithmique[1] ($O(log(x))$) `mystery_log` de cette fonction.

Exercice 3 *(4pts)* Soit la fonction `unknown n p` implémentée par les trois fonctions suivantes : `unknown1, unknown2, unknown3`.

```
let rec unknown1 n p =
  if n > p then []
  else n :: unknown1 (succ n) p

let rec unknown2 n p =
  if n > p then []
  else unknown2 n (pred p) @ [p]

let unknown3 n p =
  let rec aux p l =
    if n > p then l
    else aux (pred p) (p :: l)
  in aux p []
```

9. Quel est le type de cette fonction ?
10. Que retourne la fonction en fonction des paramètres `n` et `p` ?

Soient les tests présentés Figure 2 page 3 qui utilisent la fonction `time`.

```
let time f =
  let start = Sys.time() in
  let _ = f () in
  Sys.time () -. start
```

11. Expliquer les différences de comportement de ces trois implémentations.

Exercice 4 *(3pts)* Soit la fonction `mapfun` suivante :

```
let mapfun fs x = List.map (fun f -> f x) fs
```

12. Quel est le type de la fonction `mapfun` ?

[1] On utilisera la méthode *à la Grecque* qui utilise le fait que $x \times 2p = 2x \times p$ et $x \times (2p+1) = 2x \times p + x$.

```
# time (fun () -> unknown1 0 1000);;
- : float = 4.80000000000480043e-05
# time (fun () -> unknown2 0 1000);;
- : float = 0.0102010000000001266
# time (fun () -> unknown3 0 1000);;
- : float = 1.6000000000016e-05
# time (fun () -> unknown1 0 10000);;
- : float = 0.00140600000000001835
# time (fun () -> unknown2 0 10000);;
- : float = 0.934193999999999747
# time (fun () -> unknown3 0 10000);;
- : float = 0.000161999999999995481
# time (fun () -> unknown1 0 1000000);;
Stack overflow during evaluation (looping recursion?).
# time (fun () -> unknown2 0 1000000);;
Stack overflow during evaluation (looping recursion?).
# time (fun () -> unknown3 0 1000000);;
- : float = 0.0529409999999996828
```

FIG. 2 : Test des fonctions `unknown`i

13. Que fait la fonction `mapfun` ?

14. Donner un exemple non trivial[2] d'appel à `mapfun` ainsi que la valeur retournée et son type.

Exercice 5 *(5pts)* On considère des dictionnaires multi-valeurs : à une clé k de type `'k` est associée non pas une seule valeur mais une liste de valeurs de type `'v`. On utilise le type suivant pour représenter de tels dictionnaires.

```
type ('k, 'v) dict = ('k * 'v list) list
```

15. Définir le dictionnaire vide dans la variable `empty_dict`.

16. Écrire la fonction `add_k_v k v dict` qui retourne le dictionnaire `dict` auquel on a ajouté le couple `(k, v)`. Il pourra y avoir des doublons dans la liste des valeurs associées à une clé et l'ordre des valeurs n'importe pas.

Exemples :

```
# add_k_v 1 2 empty_dict;;
- : (int * int list) list = [(1, [2])]
# add_k_v 1 3 (add_k_v 1 2 empty_dict);;
- : (int * int list) list = [(1, [3; 2])]
# add_k_v 2 5 (add_k_v 1 3 (add_k_v 1 2 empty_dict));;
- : (int * int list) list = [(1, [3; 2]); (2, [5])]
# add_k_v 1 2 (add_k_v 2 5 (add_k_v 1 3 (add_k_v 1 2 empty_dict)));;
- : (int * int list) list = [(1, [2; 3; 2]); (2, [5])]
```

17. Écrire une fonction `dict_of_couples couples` qui construit un dictionnaire à partir d'une liste de couples (clé, valeur) `(k, v)`.

```
# dict_of_couples [(1, 2); (2, 5); (1, 3); (1, 2)];;
- : (int * int list) list = [(1, [2; 3; 2]); (2, [5])]
```

FIN

[2] sur une liste non vide.