université de BORDEAUX	**Année universitaire :** 2024/2025 **Parcours :** Licence Informatique 3e année UE 4TIN514U **Épreuve :** Devoir surveillé de Programmation fonctionnelle **Date :** mercredi 6 novembre 2024 8 :00 – 9 :20 **Durée :** 1h20 **Documents interdits.**	**Collège Sciences et Technologies**

Le sujet comporte 4 pages.

Exercice 1 *(3pts)*

Pour chacune des expressions suivantes, donner sa **valeur** et son **type** si elle est correcte, sinon expliquer pourquoi elle est incorrecte.

1. `let x = 2 in x + (let x = x * x in let y = 2 * x in x + y)`
2. `let x = 2 in let x = x + 1 and y = x * x in y - xw`
3. `fun x y -> (x +. y, x -. y)`
4. `let f x y = 3 * x + 2 * int_of_float y`
5. `f 4` où `f` est la fonction définie précédement.
6. `f 4 3` où `f` est la fonction définie précédement.

Exercice 2 *(2pts)*

Pour chacun des types suivants, donner une expression ayant ce type.

7. `float -> int -> float`
8. `int -> int -> int * bool`
9. `('a -> 'b) -> ('a -> 'b) -> 'a -> bool`
10. `(int -> 'a) -> ('b -> int) -> 'b -> 'a`

Pour la suite, on trouvera le type de chacune des fonctions demandées en dernière page.

Exercice 3 *(2pts + 1pt si récursif terminal)*

11. Écrire une fonction `iota n` qui retourne la liste des entiers de `0` à `n - 1`.

Exemples :

```
# iota 0;;
- : int list = []
# iota 9;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]
```

Exercice 4 *(3pts + 2pts si récursif terminal)*

12. Écrire une fonction `cartesian x y` qui étant données deux entiers positifs x et y, retourne le produit cartésien de $[0, x] \times [0, y]$ soit l'ensemble des couples (a, b) tels que $0 \leqslant a \leqslant x$ et $0 \leqslant b \leqslant y$.

Exemples :

```
# cartesian 0 0;;
- : (int * int) list = [(0, 0)]
# cartesian 2 3;;
- : (int * int) list =
[(0, 0); (0, 1); (0, 2); (0, 3);
 (1, 0); (1, 1); (1, 2); (1, 3);
 (2, 0); (2, 1); (2, 2); (2, 3)]
# cartesian 3 2;;
- : (int * int) list =
[(0, 0); (0, 1); (0, 2);
 (1, 0); (1, 1); (1, 2);
 (2, 0); (2, 1); (2, 2);
 (3, 0); (3, 1); (3, 2)]
```

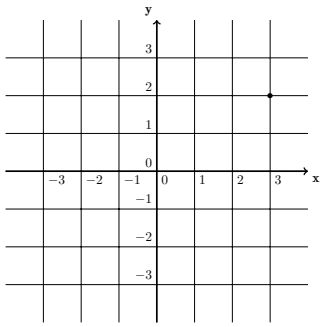

FIG. 1 : Grille planaire infinie avec un point en $(3, 2)$

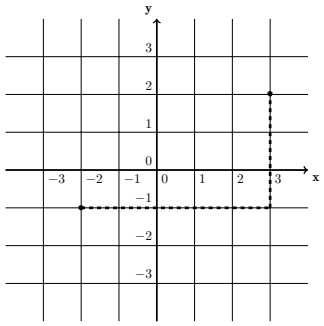

FIG. 2 : Distance de Manhattan

Exercice 5 *(2pts)*

Soit la grille planaire infinie présentée Figure 1. On souhaite représenter les points de coordonnées entières (croisement des lignes horizontales et verticales) de cette grille. Pour représenter ces points, on utilise le type `point` suivant :

```
type point = Point of int * int
```

13. Écrire une fonction constructeur `make_point x y` qui construit le point de coordonnées (x, y).

14. Écrire les accesseurs `p_x point` et `p_y point point` qui retournent respectivement l'abscisse x et l'ordonnée y d'un point `point`.

    Exemples :

```
# make_point 2 5;;
- : point = Point (2, 5)
# let point = make_point 2 5;;
val point : point = Point (2, 5)
# p_x point;;
- : int = 2
# py point;;
- : int = 5
%# p_origin;;
%- : point = Point (0, 0)
```

La *distance de Manhattan* entre 2 points (x, y) et (x', y') est définie par $|x - x'| + |y - y'|$.

C'est la distance correspondant au chemin minimum à parcourir pour aller d'un point à l'autre en suivant les lignes (verticales ou horizontales) de la grille. Par exemple Figure 2, la distance de Manhattan entre les points $(-2, -1)$ et $(3, 2)$ est $5 + 3 = 8$.

15. Écrire la fonction `distance_manhattan point1 point2` qui calcule la distance de Manhattan entre deux points `point1` et `point2` [1].

```
# distance_manhattan (make_point (-2) (-1)) (make_point 3 2);;
- : int = 8
```

[1] On rappelle la fonction prédéfinie `abs` de type `int -> int` qui permet de calculer la valeur absolue d'un entier

Exercice 6 *(8pts)* On souhaite représenter des ensembles de points, potentiellement infinis de la grille. Pour gérer l'infinitude, un tel ensemble est représenté par sa **fonction caractéristique** qui s'applique à un point et retourne un booléen (vrai si le point appartient à l'ensemble et faux sinon). On utilise le type

```
type pset = point -> bool
```

Ainsi, la variable `pset_empty` qui contient le sous-ensemble vide s'écrit :

```
let pset_empty : pset = fun point -> false
```

et la fonction `pset_member point pset` qui retourne `true` si un point `point` appartient à l'ensemble `pset` et `false` sinon :

```
let  pset_member point pset = pset point
```

16. Définir la variable `pset_full` contenant l'ensemble de tous les points de la grille.

17. Écrire la fonction `pset_make_singleton point` qui retourne l'ensemble contenant uniquement le point `point`.

Exemples :

```
# pset_member (make_point 3 4) pset_full;;
- : bool = true
# pset_make_singleton (make_point 3 4);;
- : pset = <fun>
# pset_member (make_point 3 4) (pset_make_singleton (make_point 3 4));;
- : bool = true
# pset_member (make_point 4 3) (pset_make_singleton (make_point 3 4));;
- : bool = false
```

18. Écrire la fonction `pset_complement pset` qui retourne le complémentaire de l'ensemble `pset` dans l'ensemble des points de la grille, c'est-à-dire l'ensemble des points qui ne sont pas dans `pset`.

Exemple :

```
# pset_member (make_point 3 4) (pset_complement pset_full);;
- : bool = false
```

19. Écrire la fonction `pset_union pset1 pset2` qui retourne l'union des deux ensembles de points `pset1` et `pset2`.

Exemple :

```
# pset_member (make_point 3 4)
              (pset_union (pset_make_singleton (make_point 4 3))
                          (pset_make_singleton (make_point 3 4)));;
- : bool = true
```

20. Définir la variable `pset_odd` qui contient les points dont les deux coordonnées sont impaires[2]. La représentation graphique de cet ensemble est présenté Figure 3.

Exemples :

```
# pset_odd;;
- : point -> bool = <fun>
# pset_member (make_point 1 2) pset_odd;;
- : bool = false
# pset_member (make_point 3 1) pset_odd;;
- : bool = true
```

21. Écrire la fonction `pset_of_points points` qui retourne l'ensemble des points de la liste `points`.

Exemples :

```
# pset_of_points;;
- : point list -> pset = <fun>
# let pset = pset_of_points [make_point 1 2; make_point 2 3; make_point 4 5];;
val pset : pset = <fun>
# pset_member (make_point 1 2) pset;;
- : bool = true
# pset_member (make_point 2 1) pset;;
- : bool = false
```

22. Écrire une fonction `all_points x y` qui retourne la **liste** des points de coordonnées $[0, x] \times [0, y]$. On pourra utiliser la fonction `cartesian` vue précédement.

[2] À noter qu'en `OCaML`, l'opérateur **infixe** pour le modulo est `mod`.

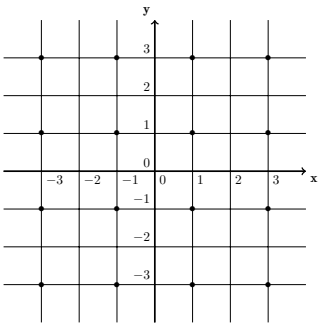

FIG. 3 : `pset_odd`

```
# all_points 3 2;;
- : point list =
[Point (0, 0); Point (0, 1); Point (0, 2); Point (1, 0); Point (1, 1);
 Point (1, 2); Point (2, 0); Point (2, 1); Point (2, 2); Point (3, 0);
 Point (3, 1); Point (3, 2)]
```

23. Écrire une fonction `pset_find_px pset x` qui retourne `None` si aucun point de coordonnées $(a, 0)$ tel que $a < x$ appartient à `pset`. Si un tel point existe la fonction retourne `Some point` où `point` est l'un d'entre eux (le premier trouvé par votre fonction).

```
# pset_find_px pset_empty 10;;
- : point option = None
# pset_find_px (pset_of_points [make_point 6 0; make_point 4 0; make_point 4 5]) 10;;
- : point option = Some (Point (4, 0))
```

FIN

```
val iota : int -> int list
val cartesian : int -> int -> (int * int) list
val make_point : int -> int -> point
val p_x : point -> int
val p_y : point -> int
val p_origin : point
val distance_manhattan : point -> point -> int
val pset_empty : pset
val pset_full : pset
val pset_make_singleton : point -> pset = <fun>
val pset_member : point -> pset -> bool
val pset_complement : pset -> point -> bool = <fun>
val pset_union : pset -> pset -> pset = <fun>
val pset_of_points : point list -> pset
val odd_p : int -> bool
val pset_odd : point -> bool
val all_points : int -> int -> point list
val pset_find_px : pset -> int -> point option
```