université de BORDEAUX	**Année universitaire :** 2021/2022 **Parcours :** Licence Informatique 2e année UE 4TINA01U **Épreuve :** Devoir surveillé de Programmation fonctionnelle **Date :** lundi 8 novembre 2021 9h30 – 10h50 **Durée :** 1h20 **Documents interdits.**	**Collège Sciences et Technologies**

Exercice 1 *(4pts)* Pour chacune des expressions suivantes, donner sa **valeur et le type** si elle est correcte, sinon expliquer pourquoi elle est incorrecte.

1. `3 * 2 + 5`
2. `3.14 +. 0`
3. `let x = 8 in x + (let x = 5 and y = 3 in x + y)`
4. `let x = 8 in x + (let x = 5 and y = 3 * x in x + y)`
5. `let x = 8 in x + (let x = 5 in let y = 2 * x in x + y)`
6. `fun x y -> (x, x +. y)`
7. `let f x y = 3 * x + 2 * int_of_float y`
8. `f 5` où `f` est la fonction définie précédement.

Exercice 2 *(2pts)* Pour chacun des types suivants, donner une expression ayant ce type.

9. `'a -> 'a`
10. `'a -> 'a -> ('a -> 'b) -> bool`
11. `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`
12. `'a * 'b -> 'a * 'b -> bool`
13. `'a * 'b -> 'b * 'a -> bool`

Exercice 3 *(5pts)* On souhaite représenter les temps d'une journée à l'aide d'un constructeur `Time` et d'un couple d'entiers (h, m) indiquant l'heure et les minutes. L'heure est représentée par un entier h, $0 \leqslant h < 24$, les minutes par un entier m, $0 \leqslant m < 60$.

14. Définir un type nommé `time` tel que `Time(23, 59)` soit de type `time`.
15. Écrire les fonctions accesseur `hour time` et `minute time`, toutes deux de type `time -> int` qui retournent respectivement l'heure et la minute d'un temps de type `time`.
16. Définir les variables `noon` et `midnight` de type `time` représentant respectivement midi (12 : 00) et minuit (00 : 00).
17. Écrire un prédicat `time_inf time1 time2` de type `time -> time -> bool` qui indique si le temps `time1` précède strictement le temps `time2`.
18. Écrire un prédicat `pm_p time` de type `time -> bool` qui indique si le temps `time` est situé dans l'après-midi, c'est-à-dire compris strictement entre midi et minuit.

Exercice 4 *(3pts)* La suite de $Syracuse(m)$ d'un nombre entier m est définie par $u_0 = m$ et pour tout $n > 0$,

$$u_n = \begin{cases} u_{n-1}/2 & si\ n\ est\ pair \\ 3u_{n-1} + 1 & sinon \end{cases}$$

19. Écrire une fonction `syracuse m n` de type `int -> int -> int` qui retourne le nième terme de la suite $Syracuse(m)$.
20. Donner une expression qui retourne la suite $Syracuse(5)$, c'est-à-dire la fonction qui à un entier n associe le nième terme de la suite la suite $Syracuse(5)$.

Exercice 5 *(7pts)* Un *multi-ensemble* est un ensemble pouvant contenir plusieurs occurences d'un même élément. Le nombre d'occurences d'un élément est appelé sa *multiplicité*. C'est un entier positif ou nul. On considère des multi-ensembles d'entiers naturels uniquement.

Soit l'ensemble de fonctionnalités (API) donné par la Figure 1 en Annexe (page 3) et permettant de construire et de manipuler des multi-ensembles. On trouvera aussi en Annexe des exemples d'utilisation de cette API dans la figure 2.

On va implémenter les multi-ensembles (qui peuvent être infinis) à l'aide de fonctions.

Un multi-ensemble est représenté par la **fonction** qui donne la multiplicité de ses éléments (et 0 si l'élément n'appartient pas au multi-ensemble). Ainsi, la fonction qui retourne un multi-ensemble vide s'écrit :

```
let m_empty = fun x -> 0
```

et la fonction `m_multiplicity`

```
let  m_multiplicity e mset = mset e
```

21. Implémenter la fonction `m_full count` qui retourne le multi-ensemble infini contenant tous les entiers naturels ayant chacun pour multiplicité `count`.

    Exemples :

```
utop[21]> m_multiplicity 100 (m_full 10);;
-: int = 10
utop[22]> m_multiplicity 100 (m_adjoin 100 (m_full 10) 2);;
-: int = 12
```

    Implémenter les fonctions

22. `m_adjoin`,
23. `m_union`.

Exercice 6 *(3pts)*

24. Implémenter `m_count n mset` qui compte (avec multiplicité) les éléments e du multi-ensemble `mset` tels que $0 \leqslant e \leqslant$ `n`.

Exemple : La variable `m2` est définie dans les exemples de l'annexe et contient le multi-ensemble $\{3, 0, 0, 3, 3, 4\}$. Ce multi-ensemble contient par exemple, 2 éléments $\leqslant 0$, 5 éléments $\leqslant 3$, 6 éléments $\leqslant 5$.

```
utop[31]> m_count 0 m2;;
2
utop[32]> m_count 3 m2;;
5
utop[33]> m_count 5 m2;;
6
```

FIN

Annexe

fonction	**valeur retournée**
`m_empty`	le multi-ensemble vide
`m_multiplicity e mset`	la multiplicité de l'élément `e` dans le multi-ensemble `mset`
`m_adjoin e mset count`	le multi-ensemble égal à `mset` auquel on a ajouté `count` copies de l'élément `e`
`m_union mset1 mset2`	union de deux multi-ensembles

FIG. 1 : API pour les multi-ensembles

Remarque : Un élément de multiplicité c_1 dans `mset1` et c_2 dans `mset2` aura pour multiplicité $c_1 + c_2$ dans l'union.

```
utop[51]> let m1 = m_adjoin 1 (m_adjoin 0 (m_adjoin 1 (m_adjoin 2 m_empty 3) 1) 4) 3;;
val m1 : int -> int = <fun>
utop[52]> m_multiplicity 1 m1;;
- : int = 4
utop[53]> m_multiplicity 1 (m_adjoin 1 m1 10);;
- : int = 14
utop[54]> let m2 = m_adjoin 3 (m_adjoin 0 (m_adjoin 3 (m_adjoin 4 m_empty 1) 2) 2) 1;;
val m2 : int -> int = <fun>
utop[55]> m_multiplicity 0 (m_union m1 m2);;
- : int = 6
```

FIG. 2 : Exemples d'utilisation de api (implémentation avec fonctions)