Option informatique : TP 2

Toutes les fonctions de ce TP devront être correctement indentées [1] et documentées, à l'aide d'un commentaire précédant la fonction et expliquant ce qu'elle est censée calculer, ses préconditions et post-conditions dans les cas non-triviaux. Une justification informelle de la terminaison des fonctions est également la bienvenue.
Exemple pour la factorielle (toujours la même !), notez l'indentation :

```
(* calcule la factorielle de son argument. L'argument doit
    etre positif ou nul. Cette fonction termine car son
    argument decroit strictement a chaque appel recursif,
    et reste positif. *)
let rec factorielle = function
    | 0 -> 1
    | n -> n * factorielle (n-1)
;;
factorielle : int -> int = <fun>
```

1. Prédire le type des expressions suivantes, avant de vérifier avec Caml :
   (a) `(1.5,("3",(4,5)));;`
   (b) `((1,2),(3,5));;`
   (c) `[|2,3.5;4,5.2;6,7.5|];;`
   (d) `([|'a';'b'|],[|[||];[|1;2;3|]|]);;`
2. Donner un exemple de variable de chacun des types suivants :
   (a) `int*float*string`
   (b) `(int*string)*(float*int)`
   (c) `(int*(float*string))*int`
   (d) `((int*bool) vect)*float`
3. Écrire une fonction qui retourne le plus petit élément d'un triplet du type `int*int*int`.
4. (a) Écrire les versions « curryfiée » et « décurryfiée » des deux fonctions suivantes :
      — $f(x, y) = (\sin x)(\cos y)$
      — $g(x, y, z) = (x + y) \times z$
   (b) Écrire le type puis le code de deux fonctions `curry` (respectivement `uncurry`) permettant d'obtenir la version curryfiée d'une fonction non curryfiée à deux arguments (et réciproquement).

1. de manière générale, on peut s'inspirer de l'indentation en Python. Penser à revenir à la ligne pour les filtrages s'ils ne sont pas absolument triviaux.

5. f et g étant deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$, définir les fonctions suivantes :
   - **somfct f g** qui donne la fonction $f + g$.
   - **prodfct f g** qui donne la fonction $f \times g$.
   - **prodext a f** qui donne la fonction af ($a \in \mathbb{R}$).
   - **dérive f epsilon** qui donne la fonction $x \mapsto \dfrac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$.
   - **compose f g** qui donne la fonction $f \circ g$.

   Tester ces fonctions sur des exemples.
6. Écrire une fonction `compose f n` qui retourne la fonction $\underbrace{f \circ \ldots \circ f}_{n \text{ fois}}$, où n désigne un entier naturel. Tester notamment pour $n = 0$.
7. Écrire une fonction `integre f a b n` qui retourne $\dfrac{b-a}{n} \sum_{k=1}^{n} f\left(a + k\,\dfrac{b-a}{n}\right)$ (f fonction de $\mathbb{R}$ dans $\mathbb{R}$, a et b réels, n entier non nul).
8. Définir deux fonctions « valeur absolue » à l'aide d'un filtrage conditionnel `when`, l'une opérant sur les entiers et l'autre sur les réels.
9. Écrire une fonction récursive implémentant la « multiplication égyptienne » (méthode de calcul utilisée par les anciens Égyptiens, n'utilisant que des additions et des multiplications par 2) :
   - Si y est pair, alors $x \times y = (2x) \times (y/2)$.
   - Si y est impair, $x \times y = x \times (y-1) + x$.
10. (a) Écrire une fonction récursive **somme** qui calcule la somme des entiers de 0 à un entier n passé en argument (retourner 0 si $n < 0$).
    (b) Faire de même avec une fonction récursive **fact**.
    (c) (Toujours plus fort) : on remarque que les deux fonctions précédentes opèrent de la même façon, à ceci près que l'une utilise l'addition dans $\mathbb{N}$ et l'autre la multiplication. Plus généralement, si $\odot$ désigne une loi de composition interne sur $\mathbb{N}$, écrire une fonction récursive
    `general : (int -> int -> int) -> int -> int`
    qui recevra en arguments la fonction $x\ y \mapsto x \odot y$ et un entier $n \geq 1$ et retournera $\bigodot_{k=1}^{n} k$, avec la convention $\bigodot_{k=1}^{1} k = 1$:
    ```
    # general (fun x y -> x+y) 10;; (* calcule sum_{k=1}^{10} k *)
    - : int = 55
    # general (fun x y -> x*y) 5;; (* calcule 5! *)
    - : int = 120
    ```
    Calculer $\sum_{k=1}^{10} k^2$ à l'aide de cette fonction.
11. (a) Écrire une fonction récursive **binomial n p** qui calcule $\binom{n}{p}$ de manière « naïve » à l'aide de la relation $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$.

(b) À l'aide cette fois de la relation $p\binom{n}{p} = n\binom{n-1}{p-1}$, écrire deux (version itérative et version récursive) fonctions **binomial2 n p**.

(c) Écrire une fonction récursive **pascal n** retournant un vecteur représentant la n-ième ligne du triangle de Pascal.

(d) Que remarque-t-on ?

12. On sait que pour abs(x) suffisamment petit, on a $\sin x \simeq x$ et $\cos x \simeq 1 - \frac{x^2}{2}$. Programmer deux fonctions mutuellement récursives `cos_app` et `sin_app` permettant le calcul approché du sinus et du cosinus d'un réel quelconque x ; on se ramènera d'abord à l'intervalle $\left[0, \frac{\pi}{2}\right]$ grâce entre autres à la périodicité, puis on utilisera les formules d'angle double $\cos 2x = \cos^2 x - \sin^2 x$ et $\sin 2x = 2 \sin x \cos x$ pour se ramener à un voisinage de 0 (défini par un epsilon donné).

13. **Marche aléatoire en dimensions 1 et 2** : La fonction `random__int:int->int` renvoie une valeur aléatoire entière comprise entre 0 et $n-1$, où n désigne son argument. Ainsi, l'utilisation répétée de l'instruction `random__int 2` simulera un déplacement aléatoire le long d'un axe (selon le résultat, on fera un pas à gauche ou un pas à droite) tandis que `random__int 4` sera utilisée pour simuler un déplacement sur un quadrillage plan (déplacement en direction de l'un des quatre points cardinaux). Dans tous les cas, on part de l'origine, et on appelle *retour à l'origine* tout passage par le point de départ après k pas ($k > 0$).

    (a) (Marche aléatoire en dimension 1) : la position courante est désignée par une variable `position` de type `int ref`.

    — Écrire une fonction `deplace : int ref -> int` simulant un déplacement d'un pas et renvoyant la position obtenue.

    — Écrire une fonction `retour : int ref -> bool` testant un retour à l'origine.

    — Écrire une fonction simulant une marche aléatoire de n pas sur un axe. Cette fonction devra retourner le triplet d'entiers formé par l'abscisse du point d'arrivée, l'abscisse du point « le plus à droite » atteint au cours de la marche et le nombre de retours à l'origine, de type `int -> int * int * int` donc.

    (b) Écrire une fonction simulant une marche aléatoire d'au plus n pas sur un quadrillage plan, jusqu'au premier retour à l'origine, et calcule le nombre de pas effectués. on pourra adapter les sous-questions précédentes avec `position : (int * int) ref` au lieu de `position : int ref`.