CY TECH	**Cycle ingénieur 2ème année** **TD n° 2 – Fonctions**	
	Matière : **Programmation fonctionnelle**	*Date* : **2023 – 2024**
		Durée : **3 heures**
		Nombre de pages : **3**

Exercice 1.

a. *Sans utiliser la console GHCi,* proposer un type valide pour chacune des expressions suivantes.

(i) `iter f x = f (f x)`

(ii) `iter (+ 1)`

(iii) `iter (+ 1) 3`

b. Trouver des fonctions pouvant avoir le type suivant :

(i) `Num a =>  a ->  a  -> a`

(ii) `Num a => (a ->  a) -> a`

(iii) `Num a =>  a ->  a  -> a  -> a`

(iv) `Num a =>  a -> (a  -> a) -> a`

(v) `Num a => (a ->  a) -> a  -> a`

(vi) `Num a => (a ->  a  -> a) -> a`

c. Trouver des fonctions ayant le type suivant :

(i) `a -> a`

(ii) `a -> a -> Bool`

(iii) `a -> b -> a`

(iv) `(a -> b) -> a -> b`

(v) `(a -> b) -> (b -> c) -> a -> c`

(vi) `(a -> b) -> (c -> b) -> a -> c -> Bool`

(vii) `(a -> b -> c) -> (a -> b) -> a -> c`

Exercice 2.

a. Écrire en Haskell la fonction

```
rateOfChange :: Double -> (Double -> Double) -> Double -> Double
```

telle que `rateOfChange h f x` soit égal au taux d'accroissement de la fonction `f` en `x` avec un pas `h`.

b. À l'aide de la fonction précédente, écrire en Haskell la fonction

```
derivative :: (Double -> Double) -> Double -> Double
```

telle que `derivative f x` soit une valeur approchée de la dérivée de `f` en `x` avec un pas de 10^{-12}.

Exercice 3.

a. Écrire en Haskell la fonction récursive

```
fastPow :: Int -> Double -> Double
```

telle que `fastPow n x` renvoie la valeur de x^n en utilisant la technique de l'exponentiation rapide. `n` est un entier relatif, donc votre fonction doit traiter tous les cas.

b. *(Pour les plus courageux !)* Écrire une version récursive terminale de la fonction précédente.

Exercice 4.

a. Écrire en Haskell la fonction récursive terminale

```
fixed :: (a -> a) -> a -> Int -> a
```

telle que `fixed next initial times` renvoie la même résultat que le pseudocode suivant :

$x \leftarrow$ `initial`
pour $i \leftarrow 1$ **à** `times` **faire**
  $x \leftarrow$ `next`(x)
fin pour
retourner x

Aucune boucle n'est autorisée.

b. Écrire en Haskell la fonction récursive terminale

```
whilst :: (a -> a) -> a -> (a -> Bool) -> a
```

telle que `whilst next initial cont` renvoie le même résultat que le pseudocode suivant :

$x \leftarrow$ `initial`
tant que `cont`(x) **faire**
  $x \leftarrow$ `next`(x)
fin tant que
retourner x

Aucune boucle n'est autorisée.

Exercice 5 (Pour les plus courageux!)**.**
(Source : cours en ligne *Principes de la programmation fonctionnelle en Scala*, Martin ODERSKY)

a. Soit I un intervalle de $\mathbb{R}$ et f une fonction réele dérivable sur I.
On appelle méthode de NEWTON la suite récurrente $(x_n)_{n\in\mathbb{N}}$ définie par la récurrence :

$$\forall n \in \mathbb{N},\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On admet que la suite $(x_n)_{n\in\mathbb{N}}$ converge vers un zéro de f.
Écrire en Haskell la fonction récursive terminale

```
squareRoot :: Double -> Double -> Double
```

telle que `squareRoot eps a` soit une valeur approchée de la racine carré de `a` à l'aide de la méthode de NEWTON appliquée à $f : x \mapsto x^2 - a$ avec $x_0 = 1$. `eps` la précision requise sur le carré de l'approximation.

b. On cherche à généraliser le processus itératif. Écrire en Haskell la fonction récursive terminale :

```
guess :: (a -> Bool) -> (a -> a) -> a -> a
```

telle que `guess isGoodEnough improveGuess initialGuess` renvoie une approximation obtenue itérativement :
— `isGoodEnough` est le test d'arrêt qui renvoie `True` si et seulement si l'approximation est suffisante;
— `improveGuess` est la fonction d'amélioration de l'approximation à chaque itération;
— `initialGuess` est l'approximation initiale.

(i) Réécrire `squareRoot` à l'aide de `guess`.

(ii) Utiliser `guess` pour calculer une approximation de π comme zéro de la fonction sinus à l'aide de la méthode de NEWTON.

c. Soit I un intervalle de $\mathbb{R}$ et ϕ une application définie sur I telle que $\phi(I) \subset I$.
On appelle méthode du point fixe, la suite récurrente $(x_n)_{n\in\mathbb{N}}$ définie par la récurrence

$$x_0 \in I \quad \text{et} \quad \forall n \in \mathbb{N},\ x_{n+1} = \phi(x_n)$$

On admet que la suite $(x_n)_{n\in\mathbb{N}}$ converge vers un point fixe de ϕ, i.e. un élément x de I tel que $\phi(x) = x$.

(i) Utiliser `guess` pour écrire en Haskell la fonction

```
fixedPointGuess :: Double -> (Double -> Double) -> Double -> Double
```

telle que `fixedPointGuess eps phi x0` soit une approximation d'un point fixe de la fonction `phi`. `eps` est la précision requise entre deux termes consécutifs et `x0` est l'approximation initiale.

(ii) Écrire en Haskell la fonction

```
damping :: (Double -> Double) -> (Double -> Double)
```

such that `damping f` soit la fonction amortie de la fonction `f`.
La fonction amortie de f est $g : x \mapsto \dfrac{x + f(x)}{2}$.

(iii) Utiliser `approximation_point_fixe` et `amorti` pour écrire en Haskell la fonction

```
fixedPointGuess_damping :: Double -> (Double -> Double) -> Double -> Double
```

telle que `fixedPointGuess_damping eps phi x0` soit une approximation d'un point fixe par amortissement de la fonction `phi`. Les autres paramètres sont identiques à ceux de `fixedPointGuess`.

(iv) Identifier `squareRoot` comme un cas particulier de `fixedPointGuess_damping`.