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

Exercice 1. Écrire en Haskell la fonction récursive terminale

```
dichotomy :: Double -> (Double -> Double) -> Double -> Double -> Maybe Double
```

telle que `dichotomy eps f a b` soit une valeur approchée d'un zéro de `f` dans l'intervalle `[a,b]` à `eps` près en utilisant la méthode de dichotomie, ou `Nothing` si une racine n'a pas pu être trouvée.

Exercice 2. Modifier le module `ArithExpr` vu en cours afin que le type `ArithExpr` soit une instance des classes de type `Num`, `Eq` et `Ord`.

Exercice 3.

a. Écrire en Haskell la fonction

```
isLeap :: Int -> Maybe Bool
```

telle que `isLeap year` :
— soit égal à `Nothing` si `year` est inférieure à 1582 (début du calendrier grégorien) ;
— sinon, indique si `year` est bissextile ou non.

b. Écrire en Haskell la fonction

```
dayCnt :: Int -> Bool -> Maybe Int
```

telle que `dayCount month leap` soit égal :
— à `Nothing` si `month` n'est pas un numéro de mois valide ;
— au nombre de jours du `month`ème mois de l'année, `leap` indiquant si l'année est bissextile ou non.

c. **Sans utiliser de *pattern matching***, combiner les deux fonctions précédentes pour écrire en Haskell la fonction

```
dayCount :: Int -> Int -> Maybe Int
```

telle que `dayCount year month` soit égal :
— à `Nothing` si le calcul ne peut pas aboutir
— au nombre de jours du `month`ème de l'année `year`.
Tous les cas doivent être traités.

Exercice 4 (Module `BinaryTree`)**.**

a. Définir en Haskell un type `BinaryTree a` qui décrit un arbre binaire dont les nœuds sont étiquetés par des éléments de type `a`.

b. *(i)* Écrire en Haskell la fonction récursive

```
size :: BinaryTree a -> Integer
```

telle que `size bt` soit égal à la taille de l'arbre binaire `bt`, c'est-à-dire à son nombre d'étiquettes.

(ii) Écrire en Haskell la fonction récursive

```
height :: BinaryTree a -> Integer
```

telle que `height a` soit égal à la hauteur de l'arbre binaire `bt`.

(iii) Écrire en Haskell la fonction récursive

```
elemNode :: a -> BinaryTree a -> Bool
```

telle que `elemNode x bt` vérifie si l'arbre binaire `bt` contient un nœud dont l'étiquette est égale à `x`. [*Question subsidiaire.* La signature fournie est incomplète : la corriger.]

c. *(i)* Identifier `size`, `height` et `elem` comme trois cas particuliers d'une nouvelle fonction `fold` à implémenter.

(ii) Réécrire `size`, `height` et `elem` avec cette nouvelle fonction.

Exercice 5 (Module `FirstOrderPredicate`)**.**

a. Définir en Haskell un type `FirstOrderPredicate` qui décrit un prédicat logique du premier ordre composé :
— de l'unique variable logique ;
— de constantes logiques « vrai » et « faux » ;
— d'opérateurs logiques (négation, conjonction, disjonction).

b. *Sans l'implémenter,* introduire la fonction

```
(==) :: FirstOrderPredicate -> FirstOrderPredicate -> Bool
```

telle que `p1 == p2` vérifie si `p1` et `p2` sont rigoureusement identiques.

c. Écrire en Haskell la fonction récursive

```
evaluate :: Bool -> FirstOrderPredicate -> Bool
```

telle que `evaluate x p` soit égal à la valeur de `p` lorsque son unique variable vaut `x`.

d. Écrire en Haskell la fonction

```
(<=>) :: FirstOrderPredicate -> FirstOrderPredicate -> Bool
```

telle que `p1 <=> p2` vérifie si `p1` et `p2` sont logiquement équivalents.

e. Écrire en Haskell la fonction récursive

```
simplify :: FirstOrderPredicate -> FirstOrderPredicate
```

telle que `simplify p` soit une proposition plus « simple » que `p` tout en lui restant logiquement équivalente.