CY TECH	**Cycle ingénieur 2ème année** **TD nº 4 – Listes**	
	Matière : **Programmation fonctionnelle**	*Date* : **2023 – 2024**
		Durée : **6 heures**
		Nombre de pages : **4**

Fonctions à prédicat

Exercice 1.

a. Réécrire de manière non récursive `all` en Haskell à l'aide de `any`.

b. Réécrire de manière non récursive `elem` en Haskell à l'aide de `any`.

Exercice 2.

a. Écrire de manière non récursive en Haskell la fonction

```
uppers :: [Char] -> [Char]
```

telle que `uppers cs` soit la liste de tous les caractères de `cs` qui sont en majuscule.

b. Écrire de manière non récursive en Haskell la fonction

```
includes :: Eq a => [a] -> [a] -> Bool
```

telle que `includes l1 l2` vérifie si `l1` contient tous les éléments de `l2`.

Exercice 3. Un document est modélisé par son titre et une liste de mots-clés.

a. Définir en Haskell le type `Document` correspondant.

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

```
searchByKeyword :: String -> [Document] -> [Document]
```

telle que `searchByKeyword kw docs` soit la liste des documents parmi `docs` dont `kw` est un mot-clé.

(ii) Écrire de manière non récursive en Haskell la fonction

```
searchByKewords :: [String] -> [Document] -> [Document]
```

telle que `searchByKeywords kws docs` soit la liste des documents parmi `docs` pour lesquels tous les éléments de `kws` sont des mots-clé.

c. Écrire de manière non récursive en Haskell la fonction

```
relevance :: [String] -> Document -> Int
```

telle que `relevance kws doc` soit égale au nombre d'éléments de `kws` qui sont des mots-clé de `doc`.

Fonctions de transformation, fonctions d'agrégation

Exercice 4.

a. Écrire de manière non récursive en Haskell en utilisant `foldl` ou `foldr` la fonction

```
sum :: [Integer] -> Integer
```

telle que `sum l` soit égal à la somme des éléments de `l`.

b. Écrire de manière non récursive en Haskell en utilisant `foldl` ou `foldr` la fonction

```
max :: Ord a => [a] -> Maybe a
```

telle que `max l` soit égal :

— à `Nothing` si `l` est vide ;

— au plus grand élément de `l` selon l'ordre induit par `Ord`.

c. Écrire de manière non récursive en Haskell en utilisant `foldl` ou `foldr` la fonction

```
maxIndex :: Ord a => [a] -> Maybe Int
```

telle que `maxIndex l` soit égal :

— à `Nothing` si `l` est vide ;

— à l'indice de la première occurrence du plus grand élément de `l` selon l'ordre induit par `Ord`.

Exercice 5. Un(e) élève est modélisé par son nom, son prénom et une liste de notes.
Le bulletin d'un(e) élève sur l'année est modélisé par son nom complet ainsi que sa moyenne générale.

a. Définir en Haskell les types `Pupil` et `ReportCard` correspondants.

b. Écrire de manière non récursive en Haskell la fonction

```
reports :: [Pupil] -> [ReportCard]
```

telle que `reports ps` soit la liste des résultats de l'année à partir de la liste `ps` des élèves.

Exercice 6.

a. En utilisant uniquement `foldl` ou `foldr`, réécrire de manière non récursive en Haskell les fonctions :

(i) `any` *(ii)* `find` *(iii)* `filter` *(iv)* `map`

b. *(i)* Réécrire de manière récursive en Haskell les fonctions `any` et `find`.

(ii) Laquelle des deux versions est la plus efficace ? Pourquoi ?

c. *(i)* Réécrire de manière récursive en Haskell la fonction `partition`, c'est-à-dire **sans utiliser `filter`**.

(ii) Cette version est-elle plus efficace que la version proposée dans le cours utilisant `filter` ? Pourquoi ?

Exercice 7.

a. Écrire de manière non récursive en Haskell la fonction

```
prefixes :: [a] -> [[a]]
```

telle que `prefixes l` soit la liste des préfixes de `l` dans l'ordre de leur longueur. Par exemple :

```
prefixes [1 ; 2 ; 3] = [[]; [1]; [1; 2]; [1; 2; 3]]
```

b. Écrire de manière non récursive en Haskell la fonction

```
join :: [a] -> [b] -> [(a, b)]
```

telle que `join l1 l2` soit la liste de tous les couples tels que :

— la première composante est un élément de `l1`;

— la deuxième composante est un élément de `l2`.

```
join [1 ; 2 ; 3] [true ; false] = [(1, true); (1, false);
                                   (2, true); (2, false);
                                   (3, true); (3, false)]
```

ATTENTION. `join` $\neq$ `zip` !

Exercice 8.

L'administration système de CY Tech veut gérer les utilisateurs de ses ressources informatiques :

— pour chaque utilisateur, on stocke son nom, son prénom et son age;

— pour tout étudiant, on stocke également son année d'étude;

— pour tout enseignant, on stocke également son département de rattachement et son ancienneté (en années).

a. Définir en Haskell les types `Student` et `Teacher` correspondants.

b. Définir en Haskell les requêtes suivantes :

(i) chercher tous les étudiants d'une année donnée et les trier;

(ii) chercher tous les enseignants d'un département donné et les trier;

(iii) chercher tous les enseignants avec au moins dix ans d'ancienneté et les trier;

(iv) chercher tous les couples étudiant/enseignant tels que l'étudiant soit au moins aussi vieux que l'enseignant.

Ces requêtes ne doivent jamais lever d'exception.

c. La gestion des études souhaite également utiliser ce modèle et y ajouter la gestion des cours. Pour chaque cours, on stocke :

— l'enseignant qui réalise le cours;

— l'année de formation concernée par le cours;

— la salle où le cours est effectué : une salle est identifiée par son nom, le bâtiment où elle se trouve et sa capacité en nombre d'étudiants;

— l'instant de début du cours;

— la durée du cours (en minutes).

(i) Définir les types `Classroom` et `Lecture` correspondant.

(ii) Définir en Haskell les requêtes suivantes :

— chercher tous les cours d'un enseignant donnée et les trier par instant de début ;

— chercher tous les cours d'un étudiant donné et les trier par instant de début ;

— chercher tous les cours dont la salle a une capacité strictement inférieure à l'effectif étudiant concerné par ce cours ;

— chercher tous les conflits dans `lectures` (on suppose que les créneaux sont tous de même durée et débutent à des heures prédéfinies sans chevauchement).

Ces requêtes ne doivent jamais lever d'exception.

Listes infinies

Exercice 9. On cherche à généraliser et simplifier l'écriture des méthodes de l'exercice 4 du TD n° 2.

a. Réécrire les fonctions `fixed` et `whilst` de la question b de l'exercice 4 du TD n° 2 à l'aide de `iterate`.

b. Écrire en Haskell la fonction

```
recurrence :: (Integer -> a -> a) -> a -> [a]
```

telle que `recurrence f u0` calcule la liste infinie des valeurs (dans l'ordre de leur indice n) de la suite récurrente $(u_n)_{n\in\mathbb{N}}$ définie par

$$u_0 = \texttt{u0} \qquad \text{et} \qquad \forall n \in \mathbb{N},\ u_{n+1} = \texttt{f}\ n\ u_n$$