Première partie : ensembles purement fonctionnels

Dans ce premier mini projet vous devez définir certaines fonctions applicables sur des ensembles.
Dans le but de vous habituer au concept de fonction d'ordre supérieur, nous choisissons de représenter les ensembles par leur fonction caractéristique. En d'autres termes, un ensemble d'entiers est une fonction qui, pour un entier en argument, retourne une valeur booléenne indiquant si l'entier est présent dans l'ensemble. On définit un alias de type représentant ceci.

```
type Set = Int => Boolean
```

Partant de cette représentation, on définit la fonction testant la présence d'une valeur dans un ensemble de la manière suivante.

```
def contains(s: Set, elem: Int): Boolean = s(elem)
```

Fonctions de base sur les ensembles

Intéressons-nous tout d'abord aux fonctions de base sur les ensembles.

1. Définissez une fonction qui créée un ensemble constitué d'une seule valeur entière. Sa signature est la suivante.

   ```
   def set(elem: Int): Set
   ```

   Maintenant que l'on sait créer des ensembles à un seul élément, on veut définir une fonction permettant de construire de plus grands ensembles à partir de plus petits.
2. Définissez les fonctions `union`, `intersect`, et `diff`, qui, pour deux ensembles d'entiers, retournent respectivement leur union, leur intersection et leur différence. `diff(s, t)` retourne l'ensemble contenant tous les éléments de l'ensemble `s` qui ne sont pas dans l'ensemble `t`. Ces fonctions ont les signatures suivantes.

   ```
   def union(s: Set, t: Set): Set
   def intersect(s: Set, t: Set): Set
   def diff(s: Set, t: Set): Set
   ```
3. Définissez la fonction `filter` qui sélectionne uniquement les éléments d'un ensemble acceptés par un prédicat donné `p`. Les éléments filtrés sont retournés comme un nouvel ensemble. Votre solution doit être indépendante de l'implémentation du type `Set` et vous n'avez pas le droit d'utiliser les trois fonctions définies dans le point 2 ci-dessus. Sa signature est la suivante.

   ```
   def filter(s: Set, p: Int => Boolean): Set
   ```

Requêtes et application sur les ensembles

Dans cette partie, nous nous intéresserons aux fonctions utilisées pour faire des requêtes sur les éléments d'un ensemble. La première fonction teste si un prédicat donné est vrai pour tous les éléments d'un ensemble. Cette fonction `forall` à la signature suivante.

```
def forall(s: Set, p: Int => Boolean): Boolean
```

On remarquera qu'il n'existe pas de façon directe de trouver quels éléments sont contenus dans un ensemble. `contains` permet seulement de savoir si un élément donné est inclus. Ainsi, si l'on souhaite appliquer quelque chose à tous les éléments d'un ensemble, il faut itérer sur tous les entiers, testant chaque fois s'il est contenu dans l'ensemble pour, le cas échéant, en faire quelque chose. On considérera qu'un entier x dispose de la propriété $-1000 \leq x \leq 1000$.

1. Implantez `forall` à l'aide de la récursion linéaire. Utilisez pour cela une fonction auxiliaire nichée dans `forall`. Sa structure sera la suivante (remplacez les " ??").

   ```
   def forall(s: Set, p: Int => Boolean): Boolean = {
     def iter(a: Int): Boolean = {
       if (??) ??
       else if (??) ??
       else iter(??)
     }
     iter(??)
   }
   ```

2. A l'aide de `forall`, implantez une fonction `exists` qui teste si un ensemble contient au moins un élément pour lequel un prédicat donné est vrai. Notez au passage que les fonctions `forall` et `exists` se comportent comme les quantificateurs $\forall$ et $\exists$ dans la logique de premier-ordre.

   ```
   def exists(s: Set, p: Int => Boolean): Boolean
   ```

3. Pour finir, écrivez la fonction `map` qui, pour un ensemble donné, transforme cet ensemble en un autre en appliquant à chacun des ses éléments une fonction donnée. `map` à la signature suivante.

   ```
   def map(s: Set, f: Int => Int): Set
   ```

   Vous remarquerez que la fonction `f` en argument ne doit pas être appliquée directement. On veut plutôt obtenir une représentation fonctionnelle de cette "description constructive" ...

Seconde partie : ensemble orientés-objet

Les exercices qui suivent portent tous sur les ensembles d'entiers vus au cours (classe `IntSet` et ses sous-classes).

Intersection I

Ajoutez une fonction `intersect` aux ensembles d'entiers, qui calcule l'intersection de deux ensembles.

Conseil : commencez par ajouter une fonction auxiliaire `intersect0` qui prend un ensemble accumulateur comme second argument. Cet accumulateur contient le résultat courant de l'intersection. Il joue un rôle similaire à celui de l'argument `result` de la fonction `iter` utilisée dans la définition de `sum` (voir cours, p. 18, semaine 2).

```
def intersect(that: IntSet): IntSet
def intersect0(that: IntSet, accu: IntSet): IntSet
```

La définition de `intersect` en termes de `intersect0` est ensuite triviale.

Filtrage

Ajoutez une fonction `filter` aux ensembles d'entiers, qui filtre un ensemble au moyen d'un prédicat. `filter` prend en argument une fonction, le prédicat, qui reçoit un entier et qui retourne un booléen. `filter` retourne ensuite le sous-ensemble de tous les entiers de l'ensemble d'origine pour lesquels le prédicat est vrai. Par exemple, l'appel suivant :

```
s.filter(x => x > 0)
```

appliqué à l'ensemble $\{-10, 5, 21, -1, 0, 3\}$ retourne l'ensemble $\{5, 21, 3\}$, c'est-à-dire le sous-ensemble des positifs.

Conseil : utilisez une technique similaire à celle utilisée pour l'exercice précédent.

Intersection II

Écrivez une nouvelle version de la fonction `intersect` au moyen de la fonction `filter` définie à l'exercice prédécent.