

## Première partie : Flots

### Exercice 1

Définissez le flot (infini) de la suite de Fibonacci. Pour mémoire, les deux premiers éléments de cette suite sont 0 et 1 ; les éléments suivants sont obtenus en sommant les deux précédents.

Pour définir ce flot, commencez par écrire la fonction `addStream`, qui prend deux flots d'entiers en argument et retourne un nouveau flot dont les éléments sont la somme des éléments correspondants de ces deux flots.

```
def addStream(s1: Stream[Int], s2: Stream[Int]): Stream[Int]
```

Au moyen de cette fonction, la définition du flot de Fibonacci prend une ligne.

### Exercice 2

$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots$  est une suite qui converge vers  $\ln(2)$ .

1. Définissez un flot formé des éléments de cette suite, en utilisant le même principe que celui vu au cours pour la suite convergeant vers  $\pi$ .
2. Appliquez la technique d'accélération d'Euler pour obtenir un flot qui converge plus rapidement que le précédent.
3. Appliquez ensuite la technique des tableaux pour obtenir une série convergeant encore plus rapidement.

Au bout de combien d'itérations arrivez-vous à un résultat jusqu'à la 10<sup>e</sup> décimale, sachant que  $\ln(2) \approx 0.693147180559945309$  ? Estimez le nombre d'itérations qui auraient été nécessaires pour obtenir cette précision avec la série non-accélérée, et estimez ainsi l'accélération totale fournie par l'utilisation des tableaux et de la technique d'Euler.

### Exercice 3

Nous nous intéressons maintenant au flot des pairs d'entiers  $(i, j)$  avec  $i, j \geq 0$ .

1. Définissez `from` qui retourne le flot de tous les entiers successifs commençants par  $n$ .

```
def from(n: Int): Stream[Int] = ...
```

2. Définissez `fromTo` qui retourne le flot de tous les entiers  $n$  pour  $l \leq n \leq u$ .

```
def fromTo(l: Int, u: Int): Stream[Int] = ...
```

3. Définissez `intPairs` qui retourne le flot de toutes les pairs d'entiers  $(i, j)$  pour  $i, j \geq 0$  en utilisant des compréhensions *for*.

```
def intPairs: Stream[(Int, Int)] = ...
```

4. Définissez une fonction équivalente à la fonction précédente *sans utiliser de compréhension *for**

## Seconde partie : résolution de contraintes

Vous devez améliorer le système de résolution de contraintes présenté au cours. Ce système est composé de *quantités* et de *contraintes*.

- Une quantité contient une valeur, qui peut être définie ou non, et relie un certain nombre de contraintes. Les quantités sont les arêtes d'un réseau de contraintes.
- Une contrainte lie plusieurs quantités par une équation simple. Les contraintes sont les nœuds d'un réseau de contraintes.

Une contrainte d'addition, par exemple, modélise une équation de la forme  $a + b = c$  où  $a$ ,  $b$  et  $c$  sont des quantités.

### Exercice 1

Vous trouverez sur la page Web du cours une implantation partielle du système de contraintes que vous complétez.

- Définissez une contrainte de multiplication modélisant une équation de la forme  $a \times b = c$ ; cette contrainte doit *savoir* que  $0 \times x = x \times 0 = 0$  quel que soit  $x$ , donc même si  $x$  est indéfini,
- Définissez une contrainte d'égalité modélisant une équation du type  $a = b$ .

Pour chacune de ces nouvelles contraintes, ajoutez un opérateur à la classe `Quantity` qui permette l'écriture agréable de contraintes, en vous inspirant de l'opérateur `+` déjà défini. Nommez l'opérateur pour les contraintes de multiplication `*`, et celui pour les contraintes d'égalité `==`.

### Exercice 2

Est-il possible d'implanter une équation d'élevation au carré de la forme  $a^2 = b$  au moyen de votre contrainte de multiplication? Si oui, implantez-la, si non, expliquez pourquoi et donnez une solution directe qui n'utilise pas d'autres contraintes.