| Programmation IV | Série 7 | 3 mai 2006 |
|---|---|---|

# Introduction

Pour cette série, nous allons construire un petit langage graphique, permettant de créer des images composées d'un ensemble d'images primitives. Par « langage » on entend un ensemble de fonctions Scala permettant de travailler avec des images.

Les deux abstractions de base de notre langage d'images sont le *cadre* et le *peintre*. Un cadre est un simple parallélogramme à l'intérieur duquel une image peut être dessinée. Un peintre est une fonction qui prend en argument un cadre et qui dessine une image à l'intérieur de ce cadre. L'image dessinée par un peintre est spécifique au peintre, et ne peut être modifiée.

Le langage d'images permet de *composer* plusieurs peintres pour en obtenir un nouveau. Par exemple, il est possible de prendre deux peintres et d'en obtenir un nouveau qui place les images des deux peintres côte-à-côte. Le fait que la composition de deux peintres soit elle-même un peintre est une caractéristique fondamentale de notre langage, et permet la création d'images relativement complexes à partir de constructions très simples.

Étant donné que les peintres sont des fonctions, la composition de peintres se fait simplement en composant des fonctions. Par exemple, l'opération plaçant deux peintres côte-à-côte n'est rien d'autre qu'une fonction qui prend en argument deux fonctions — les deux peintres à composer — et qui retourne une nouvelle fonction — le peintre résultat.

Un cadre (fig. 1) est spécifié par trois vecteurs : le vecteur origine $\overrightarrow{o}$, le vecteur abscisse $\overrightarrow{x}$ et le vecteur ordonnée $\overrightarrow{y}$. Le vecteur origine spécifie la position de l'origine du cadre par rapport à l'origine du dessin englobant.

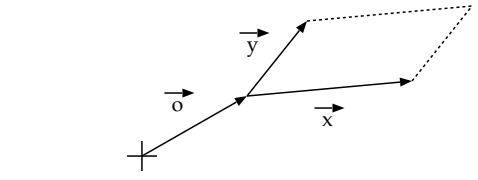


FIG. 1 – Cadre

Pour vous aider à réaliser le langage d'images, nous mettons à votre disposition un fichier Scala sur la page Web du cours. La classe `Vector` permet de définir des vecteurs à deux composantes et d'effectuer quelques opérations sur ces vecteurs. `Frame` est une classe modélisant les cadres. `Canvas` est une classe abstraite permettant d'effectuer le rendu d'images, soit dans un fichier PostScript, soit directement à l'écran.

Finalement, `Painter-partial.scala` contient les définitions relatives aux peintres et un exemple de création de dessin, que vous pouvez directement visualiser. C'est ce dernier fichier que vous devez modifier.

Nous pouvons maintenant définir des fonctions manipulant des peintres. Une manière très générale de transformer un peintre est de le faire peindre dans un nouveau cadre défini en fonction de celui qu'il reçoit en argument. Par exemple, pour obtenir un peintre qui dessine la même image qu'un autre mais inversée verticalement, il suffit de lui passer un cadre inversé, comme l'illustre la figure 2.

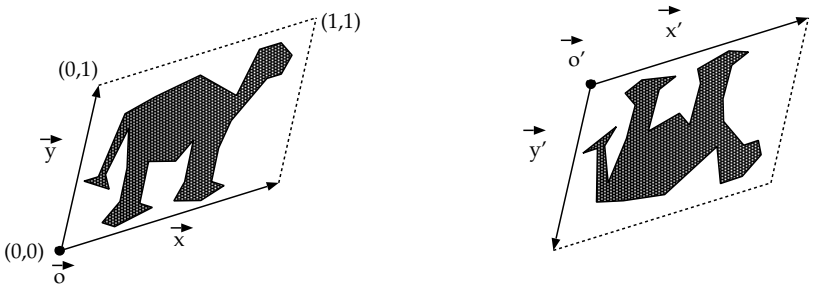


FIG. 2 – Inversion verticale d'une image par inversion du cadre

L'opérateur `/` dans la classe `Frame` permet de définir un cadre relativement à un autre. Si `frame2` est un cadre exprimé dans le référentiel d'un autre cadre `frame1`, alors `frame1 / frame2` retourne le cadre `frame2` exprimé dans le même référentiel que `frame1`. La figure 3 illustre le fonctionnement de cet opérateur : `frame1` est exprimé dans le référentiel $(\overrightarrow{o}, \overrightarrow{x}, \overrightarrow{y})$, tandis que `frame2` est exprimé dans le référentiel défini par `frame1`, à savoir $(\overrightarrow{o'}, \overrightarrow{x'}, \overrightarrow{y'})$. Le résultat de l'opération `frame1 / frame2` est le cadre `frame2` exprimé dans le référentiel $(\overrightarrow{o}, \overrightarrow{x}, \overrightarrow{y})$.

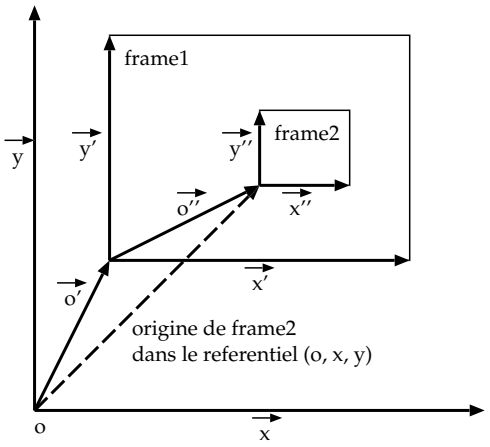


FIG. 3 – L'opérateur /

Par exemple, la fonction `flipVert`, qui inverse verticalement l'image dessinée par un peintre, peut s'exprimer de la manière suivante :

```
def flipVert(p: Painter): Painter = {
  frame: Frame => p(frame / (new Frame(new Vector(0.0, 1.0),
                                       new Vector(1.0, 0.0),
                                       new Vector(0.0, - 1.0))))
}
```

Pour vous familiariser avec le système, lisez attentivement le fichiers fournis.

# Partie 1

Écrivez la fonction `beside` qui prend en argument deux peintres et en retourne un nouveau qui place le premier à gauche du second. Vous pouvez tester cette fonction en essayant de dessiner deux triangles l'un à côté de l'autre, au moyen du peintre `triangle` fourni.

# Partie 2

Écrivez la fonction `below` qui prend en argument deux peintres et en retourne un nouveau qui place le premier en dessous du second.

# Partie 3

Au moyen de ces deux fonctions et des peintres fournis, définissez une fonction de dessin du *triangle de Sierpinski*. Cette fonction prend en argument un entier $n$ et retourne un peintre dessinant le triangle de Sierpinski de niveau $n$. Ce triangle est défini de manière récursive ainsi : le triangle de Sierpinski de niveau $0$ est un triangle rectangle isocèle dont les sommets sont l'origine et les points $(1, 0)$ et $(0, 1)$. C'est exactement ce que dessine le peintre `triangle` fourni. Le triangle de Sierpinski de niveau $n > 0$ est composé de trois triangles de niveau $n - 1$ arrangés de manière à former un plus grand triangle, comme illustré à la figure 4.

# Partie 4 (optionnelle)

S'il vous reste encore du temps, définissez une fonction qui prend en argument un entier $n$ et retourne un peintre dessinant le *flocon de von Koch* de niveau $n$. Le flocon de von Koch est défini récursivement, d'une manière similaire au triangle de Sierpinski : le flocon de von Koch de niveau $0$ est un simple segment unitaire et horizontal ; le flocon

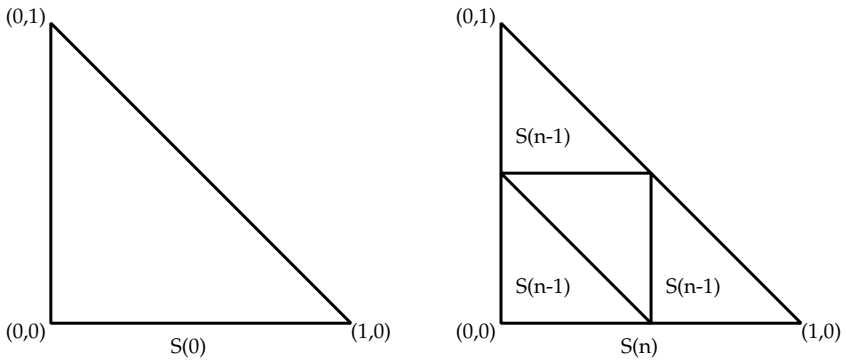


FIG. 4 – Triangle de Sierpinski de niveau $0$ et de niveau $n$

de von Koch de niveau $n > 0$ est composé de quatre flocons de niveau $n - 1$ arrangés de la manière présentée à la figure 5.

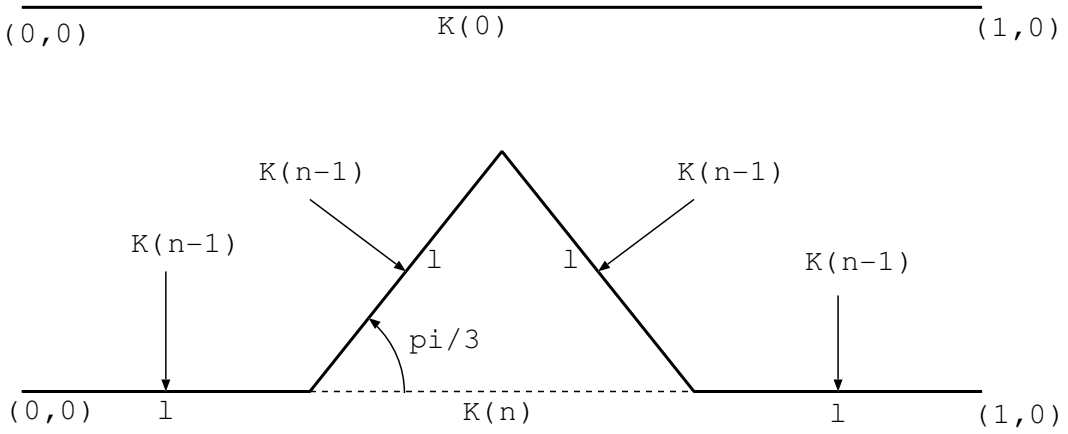


FIG. 5 – Flocons de von Koch de niveau $0$ et $n$