Solution à rendre avant le 25 avril 2002 à 18h dans la boîte devant le bureau INR 318

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 remplissant 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 final, c-à-d l'origine de l'écran ou de la feuille de papier.

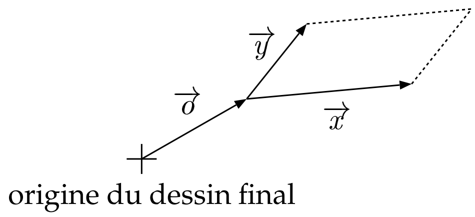


FIG. 1 – Cadre

Pour vous aider à réaliser le langage d'images, nous mettons à votre disposition un certain nombre de fichiers Scala sur la page Web du cours. `Vector.scala` contient une classe permettant de définir des vecteurs à deux composantes et d'effectuer quelques opérations sur ces vecteurs. `Frame.scala` contient la classe modélisant les cadres. `Graphics.scala` contient une classe permettant d'effectuer le rendu d'images, soit dans un fichier PostScript soit directement à l'écran. Finalement, `Painter.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 transformer le cadre qui lui est passé. 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 la figure 2 l'illustre.

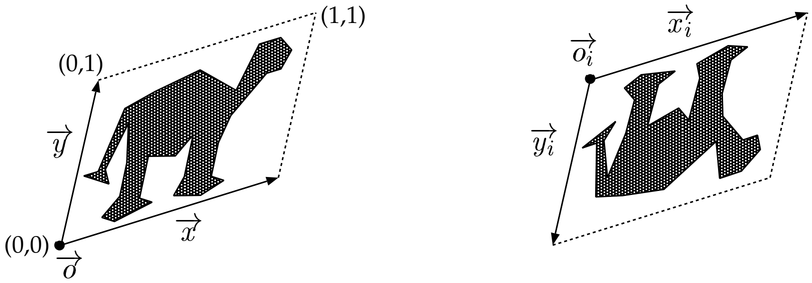


image originale   image inversée

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

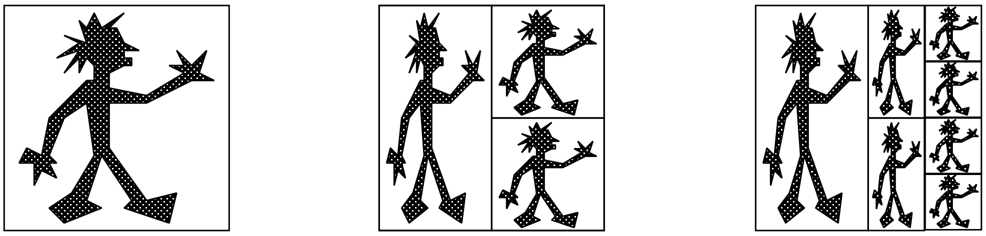
La fonction `transformPainter` permet d'effectuer de telles transformations. Elle prend en argument trois points et un peintre. Les trois points spécifient la position de l'origine et des extrémités de l'abscisse et de l'ordonnée du nouveau cadre dans le référentiel du cadre original. La fonction `flipVert`, qui inverse verticalement l'image dessinée par un peintre, s'exprime alors de la manière suivante :

```
def flipVert(p: Painter) =
  transformPainter(Vector(0.0, 1.0), Vector(1.0, 1.0), Vector(0.0, 0.0))
  (p)
```

Pour vous familiariser avec le système, lisez attentivement les fichiers fournis (à l'exception peut-être de `Screen.scala` et `PostScript.scala`), puis définissez, au moyen de la fonction `transformPainter`, la fonction `flipHoriz`. Cette fonction est similaire à `flipVert` mais inverse l'image horizontalement.

En vous inspirant de la fonction `beside` fournie, écrivez la fonction `below` qui prend en argument deux peintres et qui en retourne un nouveau plaçant les images de ces deux peintres l'une en dessus de l'autre.

Au moyen des fonctions `beside` et `below`, définissez la fonction `rightSplit` qui prend en argument un peintre et un entier n et qui retourne un nouveau peintre. Ce nouveau peintre peint $2^{n+1} - 1$ copies de l'image originale selon le schéma de la figure 3.



p ≡ `rightSplit` (p, 0)   `rightSplit` (p, 1)   `rightSplit` (p, 2)

FIG. 3 – La fonction `rightSplit`