Programmation avancée

Examen final

vendredi 24 décembre 2010

Nom : _____

Prénom : _____

Vos points sont *précieux*, ne les gaspillez pas !

Votre nom Le travail qui ne peut pas vous être attribué est perdu: écrivez votre nom sur chaque feuille que vous rendez.

Votre temps Tous les points ne sont pas égaux. Commencez par résoudre les problèmes dont le rapport points/difficulté vous paraît le plus favorable.

Votre attention La donnée de chaque exercice est précisément formulée, et parfois subtile. Si vous ne la comprenez pas, vous ne pourrez pas en tirer tous les points.

Exercice	**Points**	**Points obtenus**
1	10	
2	4	
3	10	
4	6	
Total	30	

Exercice 1 : Systèmes de Lindenmayer (10 points)

Un *L-System* est un système de réécriture qui décrit comment obtenir une séquence infinie de chaînes de caractères à partir d'une *chaîne initiale* et d'un ensemble de *règles*. Chaque règle décrit la récriture d'un caractère unique en une chaîne.

Par exemple, le L-System composé de la chaîne initiale `A` et des deux règles (1) `A` $\rightarrow$ `B-A-B` et (2) `B` $\rightarrow$ `A+B+A` produit une séquence infinie de chaînes dont les trois premiers éléments sont :

1 :	`A`	*(chaîne initiale)*
2 :	`B-A-B`	*(par application de la règle 1)*
3 :	`A+B+A-B-A-B-A+B+A`	*(par application des règles 1 et 2)*

Comme cet exemple l'illustre, toutes les règles applicables sont appliquées en parallèle à chaque étape. De plus, les caractères pour lesquels il n'existe pas de règle (`-` et `+` dans cet exemple) sont laissés intacts.

Pour représenter de tels systèmes en Scala, on définit tout d'abord le type `LString` des chaînes, que l'on représente par des listes de caractères :

```
type LString = List[Char]
```

Ensuite, on définit le type des règles, que l'on représente par une paire composée d'un caractère (le caractère à récrire) et d'une chaîne (le résultat de la récriture) :

```
type LRule = (Char, LString)
```

Partie 1 (8 points) Ecrivez la fonction `lStep` qui, étant donné une chaîne initiale et une liste de règles, récrit la chaîne en appliquant les règles. Cette fonction a le profil suivant :

```
def lStep(start: LString, rules: List[LRule]): LString = ...
```

Suggestion : aidez-vous de la méthode `find` de la classe `List`, qui est décrite dans le formulaire à la fin de cet énoncé.

Appliquée au deuxième élément et aux règles de l'exemple décrit plus haut :

```
lStep(List('B','-','A','-','B'),
      List(('A', List('B','-','A','-','B')),
           ('B', List('A','+','B','+','A'))))
```

cette fonction retourne le troisième élément :

```
List('A','+','B,'+','A','-','B','-','A','-','B','-','A',
     '+','B,'+','A')
```

Partie 2 (2 points) En vous aidant de `lStep`, définissez la fonction `lStream` qui, étant donné une chaîne de départ et une liste de règles, retourne le flot infini des chaînes successives produites par le L-System. Cette fonction a le profil suivant :

```
def lStream(start: LString, rules: List[LRule]): Stream[LString] =
  ...
```

Exercice 2 : Filtrage de motifs (4 points)

Soient les deux fonctions suivantes :

```
def f(list: Any) = list match {
  case a :: b :: Nil ⇒ true
  case _             ⇒ false
}

def g(list: Any) = list match {
  case (a :: b) :: c ⇒ true
  case _             ⇒ false
}
```

Pour les différentes valeurs de `list` indiquées dans le tableau suivant, cochez les cases pour lesquelles l'appel à `f` ou à `g` retourne **`true`**.

`val` `list =`	`f(list) ==` **`true`**	`g(list) ==` **`true`**
`1 :: 2 :: 3 :: Nil`	☐	☐
`List(1 :: 2 :: 3 :: Nil)`	☐	☐
`(1 :: Nil) :: (2 :: Nil) :: Nil`	☐	☐
`List(1, 2, 3) :: List(3, 4, 5)`	☐	☐
`Nil :: Nil`	☐	☐
`(1 :: Nil) :: Nil`	☐	☐
`List(1, 2) :: List(3)`	☐	☐
`List(1, 2) :: List(3, 4) :: Nil`	☐	☐

Exercice 3 : Quadtrees (10 points)

Un *quadtree* est un arbre représentant un ensemble d'éléments positionnés dans le plan. Dans le cadre de cet exercice, les quadtrees représentent des ensembles de villes positionnées géographiquement.

La figure 1 montre un ensemble de quatre villes de Suisse. Elle présente également les quadrants délimités par la ville de Fribourg : le premier, nommé NW (pour *north-west*), contient la ville de Neuchâtel ; les deux suivants (NE pour *north-east* et SE pour *south-east*) sont vides ; le dernier (SW pour *south-west*) contient les deux dernières villes, Yverdon et Lausanne.

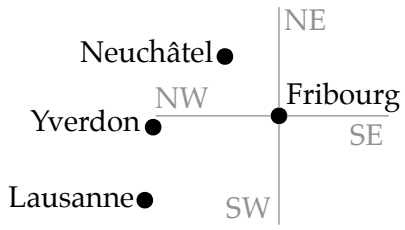

FIGURE 1 – Un ensemble de quatre villes de Suisse

La figure 2 montre une représentation possible de cet ensemble sous forme d'un quadtree. La ville de Fribourg et sa position (578461, 183802) sont à la racine de l'arbre. Chacun des quatre fils de Fribourg est un quadtree contenant les villes qui se trouvent dans un quadrant donné. Par exemple, le quatrième fils (SW) est un quadtree contenant Lausanne et sa position à la racine. Les fils de Lausanne sont eux-aussi des quadtrees, dont un seul n'est pas vide : NE, qui contient Yverdon car cette ville est dans le quadrant nord-est de Lausanne.

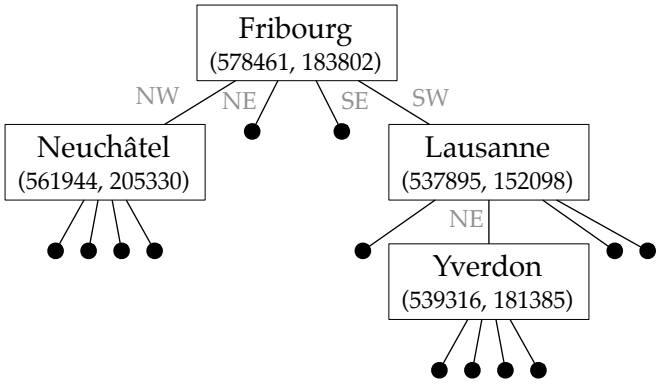

FIGURE 2 – Un quadtree pour les villes de la figure 1

(suite au verso)

Pour modéliser ces quadtrees en Scala, on commence par définir le type des points et le type des villes, ces dernières étant simplement représentées par leur nom :

```
case class Point(x: Int, y: Int)
type City = String
```

Les quadtrees quant à eux sont représentés par une classe mère abstraite `QuadTree` et deux sous-classes concrètes : `Empty` — un simple objet — représente les feuilles, tandis que `Fork` représente les nœuds composés d'une ville, de sa position et de quatre quadtree fils.

```
abstract class QuadTree {
  def insert(p: Point, c: City): QuadTree
  def lookup(p: Point): Option[City]
  def foreach(f: (Point, City) ⇒ Unit): Unit
}
case object Empty extends QuadTree { /* ... partie 1 */ }
case class Fork(pos: Point, city: City,
                nw: QuadTree, ne: QuadTree,
                se: QuadTree, sw: QuadTree) extends QuadTree {
  private def inNW(p: Point) = p.x <  pos.x && p.y >= pos.y
  private def inNE(p: Point) = p.x >= pos.x && p.y >  pos.y
  private def inSE(p: Point) = p.x >  pos.x && p.y <= pos.y
  private def inSW(p: Point) = p.x <= pos.x && p.y <  pos.y

  /* ... partie 2 */
}
```

Trois méthodes sont déclarées dans la classe mère :

1. `insert(p, c)` produit un nouvel arbre contenant les mêmes villes que celui sur lequel l'opération est appliquée, avec en plus la ville `c` à la position `p`. Si l'arbre contenait déjà une ville à la position `p`, celle-ci est remplacée par `c`.
2. `lookup(p)` retourne la ville se trouvant à la position `p` ou `None` s'il n'y en a aucune. (Le type `Option` est décrit dans le formulaire à la fin de cet énoncé.)
3. `foreach(f)` applique la fonction `f` à tous les couples (ville, position) de l'arbre.

Partie 1 (3 points) Complétez l'objet `Empty` en définissant ses trois méthodes `insert`, `lookup` et `foreach`.

Partie 2 (7 points) Complétez la classe `Fork` en définissant ses trois méthodes `insert`, `lookup` et `foreach`.

Vous avez bien entendu le droit de vous aider des méthodes privées `inNW`, `inNE`, `inSE` et `inSW` qu'elle contient. Ces méthodes testent l'appartenance d'un point à un quadrant.

Exercice 4 : Scheme−− (6 points)

Soit la fonction Scala suivante :

```
def f[T](l: List[T], p: T ⇒ Boolean): (List[T], List[T]) =
  l match {
    case h :: t if p(h) ⇒
      val tt = f(t, p)
      (h :: tt._1, tt._2)
    case l ⇒
      (List(), l)
  }
```

Partie 1 (2 points) Décrivez, en une phrase, ce que fait cette fonction. Donnez une description de haut niveau, ne vous contentez pas de paraphraser le code !

Partie 2 (4 points) Donnez la version Scheme−− de cette fonction. Pour cela, représentez les paires de valeur au moyen de cellules `cons` : utilisez la fonction `cons` pour créer une paire, la fonction `car` pour extraire sa première composante et la fonction `cdr` pour extraire sa seconde composante.

Formulaire

Ce formulaire décrit des versions simplifiée de quelques classes de la bibliothèque standard Scala que vous pourriez trouver utile pour cet examen.

Tuple2

Le type `Tuple2` (ou `Pair`, qui est un alias) représente les paires de valeurs.

```
case class Tuple2[T,U](_1: T, _2: U)
```

On utilise généralement la notation parenthésée pour les types des tuples, le type noté `(T, U)` étant équivalent au type `Tuple2[T,U]` (pour tous types `T` et `U`).

List

Le type `List[T]` représente les listes de valeurs de type `T`.

La méthode `map` applique la fonction reçue en argument à tous les éléments de la liste et retourne la liste des résultats.

La méthode `flatMap` applique la fonction reçue en argument à tous les éléments de la liste et retourne la concaténation de toutes les listes résultantes.

La méthode `find` retourne le premier élément de la liste satisfaisant le prédicat donné, s'il y en a un. Cet élément est retourné sous la forme d'une valeur optionnelle (voir la classe `Option` ci-dessous).

```
abstract class List[T] {
  def map[U](f: T ⇒ U): List[U]
  def flatMap[U](f: T ⇒ List[U]): List[U]
  def find(p: T ⇒ Boolean): Option[T]
}
```

(Pour simplifier, les deux sous-classes concrètes de `List` ne sont pas présentées.)

Option

Le type `Option[T]` représente les valeurs optionnelles de type `T`.

```
trait Option[+T]
case class Some[+T](v: T) extends Option[T]
case object None extends Option[Nothing]
```