# Programmation avancée

Examen final

vendredi 23 décembre 2011

Nom : _____

Prénom : _____

Ne pas commencer avant l'annonce, SVP.

Lisez attentivement le suivant pour ne pas gaspiller vos points *précieux* !

**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. En effet, nous ne pensons pas que tous les exercices ont la même difficulté, même s'ils ont le même nombre de points. Si vous êtes bloqués sur un exercice, mettez-le à coté pour d'abord travailler sur les autres.

**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 | 15 | |
| 2 | 8 | |
| 3 | 12 | |
| **Total** | 35 | |

## Exercice 1 : Cordes (15 points)

Une corde (“rope” en anglais) est une structure de données qui permet de manipuler efficacement de très grandes chaînes de caractères. C’est un arbre binaire qui consiste en deux types de noeuds : `Leaf` et `Node`.


Un `Leaf` contient une chaîne de caractères, un `Node` a deux références vers des noeuds enfants. Chaque noeud, soit `Leaf` ou `Node`, a un champ `length` qui représente la somme des tailles de toutes les chaînes de caractères dans le sous-arbre. Par exemple, la chaîne de caractères `"Scala is way cool!"` peut être stockée dans une corde de la manière suivante :

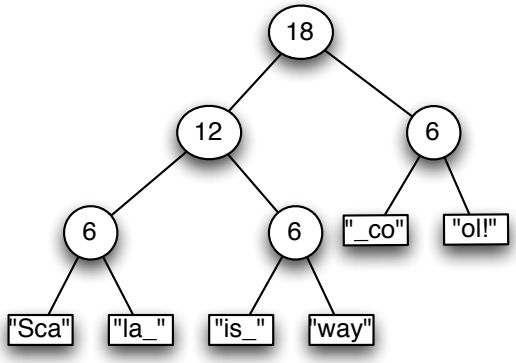



FIGURE 1 – Corde 1

La même chaîne de caractères peut être représentée par des cordes différentes. La corde suivante est une alternative pour le même exemple :

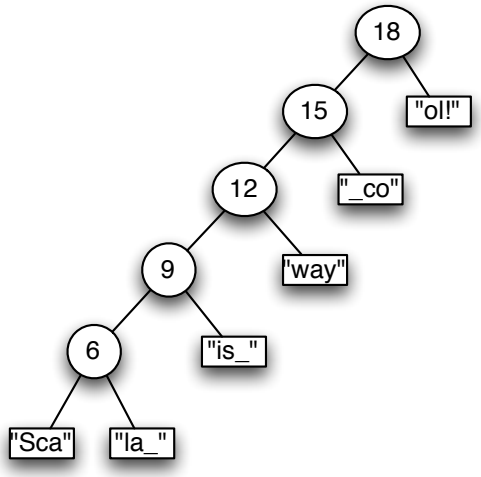



FIGURE 2 – Corde 2

### Implémentation (10 Points)

Le code suivant donne une implémentation partielle des cordes en Scala.

```
abstract class Rope {
  def length: Int
  def concat(other: Rope): Rope = [...]     // Partie 1
  def charAt(i: Int): Char
  def collectLeaves: List[Leaf]
  def balance: Rope = [...]                 // Partie 5
}
object Rope {
  def binTree(l: List[Leaf]): Rope = [...]  // Partie 4
}
case class Leaf(s: String) extends Rope {
  def length = s.length
  def charAt(i: Int): Char = [...]          // Partie 2
  def collectLeaves: List[Leaf] = [...]     // Partie 3
}
case class Node(l: Rope, r: Rope) extends Rope {
  def length = l.length + r.length
  def charAt(i: Int): Char = [...]          // Partie 2
  def collectLeaves: List[Leaf] = [...]     // Partie 3
}
```

Vos tâches pour cet exercice sont les suivantes :

**Partie 1 (1 point)** Implémentez la méthode `concat` *dans la classe abstraite* `Rope`. Elle prend une corde `other` en argument et retourne une nouvelle corde représentant la concaténation des deux cordes **`this`** et `other`.

**Partie 2 (2 points)** En utilisant la méthode `length`, et la méthode `charAt` de la classe `String`, implémentez la méthode `charAt` *dans les deux sous-classes* `Leaf` et `Rope`. Elle retourne le caractère de la corde à la position `i`. Si `i` est une position invalide, le comportément n'est pas spécifié.

**Partie 3 (2 points)** Dans *les deux sous-classes*, implémentez la méthode `collectLeaves` qui retourne une liste de feuilles `List[Leaf]` contenant tous les feuilles de la corde, dans l'ordre correct.

**Partie 4 (4 points)** On définit la *hauter d'une corde* comme le maximum nombre de noeuds sur un chemin depuis la racine jusqu'à un `Leaf`. La hauteur de la corde Figure 1 est 3, la hauteur de dans la Figure 2 est 5.

Une corde est équilibrée si ses sous-cordes sont équilibrées, et la différence des hauters de ses deux sous-cordes est au plus 1. L'exemple dans Figure 1 est équilibré, la corde dans Figure 2

ne l'est pas.

Implémentez la méthode auxiliaire `binTree` dans l'objet `Rope` qui prend comme argument une liste de feuilles `List[Leaf]` et retourne une corde équilibrée contenant ces `Leaf`.

*Conseil* : Utilisez la récursion : dans une codre équilibrée, chaque sous-corde d'un `Node` contient la moitié des `Leaf`. Il n'est pas nécessaire de connaître ou calculer des hauteurs. La corde résultante a la forme d'une pyramide.

**Partie 5 (1 point)** Implémentez, dans la classe abstraite `Rope`, la méthode `balance` qui retourne une version équilibrée de la corde **`this`**, en utilisant les méthodes `collectLeaves` et `binTree`.

## Preuve par induction (5 points)

Dans cette partie, vous allez utiliser une preuve par induction pour vérifier que l'implémentation suivante de la méthode `reverse` est correcte :

```
abstract class Rope {
  def reverse: Rope = this match {
    case Node(l, r) => Node(r.reverse, l.reverse) // (reverse-N)
    case Leaf(s)    => Leaf(s.reverse)            // (reverse-L)
  }

  override def toString: String = this match {
    case Node(l, r) => l.toString + r.toString    // (toString-N)
    case Leaf(s)    => s                          // (toString-L)
  }
}
```

Prouvez que la propriété suivante est correcte pour chaque corde `rope` :

```
rope.reverse.toString == rope.toString.reverse
```

Vous pouvez utiliser la loi suivante, où `x` et `y` sont des chaînes de caractères :

```
(x + y).reverse == y.reverse + x.reverse     // (reverse-string)
```

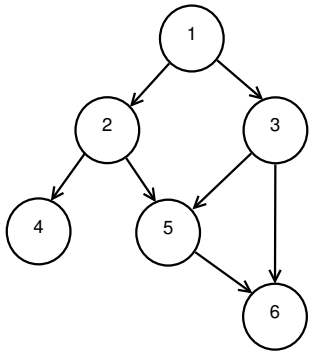
**Partie 6** La preuve est développée par une induction sur la structure de la corde. Dans cette partie, prouvez le cas de base. À chaque étape de la preuve, indiquez la loi appliquée, par exemple `(reverse-N)` ou `(string-reverse)`.

**Partie 7** Prouvez l'étape d'induction.

– Spécifiez précisement la / les hypothèse(s) d'induction.
– Prouvez le cas d'induction. De nouveau, indiquez à chaque étape la loi appliquée.

## Exercice 2 : Graphe orienté acyclique en Prolog (8 points)

Nous définissons les graphes orientés acycliques en Prolog de la manière suivante : Chaque sommet (noeud) du graphe est représenté par un nombre entier. Les arcs (liens) d'un graphe sont définis comme des faits en Prolog. Nous utilisons le prédicat `e(x, y)` pour représenter un arc de `x` vers `y`.




Le graphe ci-dessus est donc représenté par les faits suivants en Prolog :

```
e(1,2).      e(1,3).
e(2,4).      e(2,5).
e(3,5).      e(3,6).
e(5,6).
```

Dans cet exercice, vous pouvez assumer qu'il n'existe pas de cycles dans les graphes.

Dans votre solution, utilisez le point-virgule (;) comme opérateur de disjonction, donc comme "ou" logique : $A$ ou $B$ s'écrit $A$ ; $B$.

**Partie 1 (2 points)** Définissez un prédicat `neighbor(X, Y)` qui spécifie le voisinage de deux sommets : il est vrai s'il existe un arc qui relie directement `X` et `Y`.

Dans l'exemple, nous avons :

```
prolog> ?neighbor(3, 1)
true
prolog> ?neighbor(2, 6)
false
```

**Partie 2 (2 points)** Définissez le prédicat `reachable(A, B)` qui est vrai si le sommet `B` est atteignable depuis `A` en suivant les arcs du graphe. Notez que chaque sommet est atteignable depuis lui-même.

Cet exercice est indépendant de la partie 1. Exemples :

```
prolog> ?reachable(1, 1)
true
prolog> ?reachable(1, 4)
true
prolog> ?reachable(3, 4)
false
```

**Partie 3 (4 points)** Définissez un dernier prédicat `path(X, Y, Route)` qui décrit le chemin reliant deux sommets. La variable `Route` est la liste de sommets du chemin entre `X` et `Y`, avec le début et la fin inclus.

Cet exercice est indépendant des deux premiers. Exemples :

```
prolog> ?path(1, 6, Route)
[Route = [1, 3, 5, 6]]
[Route = [1, 3, 6]]
[Route = [1, 2, 5, 6]]
prolog> ?path(2, 2, [2])
true
```

There's nothing to see here. Please move along.

## Exercice 3 : Refactoriser des boucles While (12 points)

Comme première tâche chez votre nouveau employeur MegaCorp Inc., vous êtes chargés de refactoriser du vieux code Scala. Malhereusement, le programmeur qui a écrit ce code n'a pas suivi notre cours et n'a jamais entendu parler de programmation fonctionnelle - c'est pour ça que la position est maintenant la vôtre :-)

En effet, il a utilisé des boucles **`while`** et des variables partout ! Il a aussi nommé ses fonctions auxiliaires `fun1`, `fun2`, etc.

Pour déblayer ce chaos, vous allez re-écrire les expressions dans la colonne gauche de la table suivante (1.5 points par exercice). Au lieu des méthodes `fun`, utilisez une ou plusieurs des méthodes de traitement de listes communs : `map`, `filter`, etc. Une liste de méthodes disponibles est donnée à la fin.

| **Vieux Code** | **Nouveau Code** |
|---|---|
| `fun1(xs)` | |
| `fun2(xs)(x => 2 * x)` | |
| `fun3(xs)(_ + 7)` | |
| `fun4(xs,ys)` | |
| `fun5(xs)(_ % 2 == 1))` | |
| `fun6(xs, 3)` | |
| `fun2(xs)(ys => fun1(ys))` | |
| `fun5(xs)(p)._1` | |

La page suivante montre les différentes méthodes `fun`. Il ne vous faut pas les modifier, seulement comprendre ce qu'ils font.

```
def fun1[T](xs0: List[T]) = {
  var xs: List[T] = xs0
  var ys: List[T] = Nil
  while (!xs.isEmpty) {
    ys = xs.head :: ys
    xs = xs.tail
  }
  ys
}
```

```
def fun2[T,U](xs0: List[T])(f: T=>U) = {
  var xs: List[T] = xs0
  var ys: List[U] = Nil
  while (!xs.isEmpty) {
    ys = f(xs.head) :: ys
    xs = xs.tail
  }
  ys.reverse
}
```

```
def fun3[T](xs0: List[T])(f: T=>Int) = {
  var xs: List[T] = xs0
  var y: Int = 0
  while (!xs.isEmpty) {
    y = y + f(xs.head)
    xs = xs.tail
  }
  y
}
```

```
def fun4[T,U](xs0:List[T],ys0: List[U])={
  var xs: List[T] = xs0
  var ys: List[U] = ys0
  var zs: List[(T,U)] = Nil
  while (!xs.isEmpty && !ys.isEmpty) {
    zs = (xs.head,ys.head) :: zs
    xs = xs.tail
    ys = ys.tail
  }
  zs
}
```

```
def fun5[T](xs0: List[T])(p: T=>Boolean) = {
  var xs: List[T] = xs0
  var ys: List[T] = Nil
  var zs: List[T] = Nil
  while (!xs.isEmpty) {
    if (p(xs.head))
      ys = xs.head :: ys
    else
      zs = xs.head :: zs
    xs = xs.tail
  }
  (ys.reverse, zs.reverse)
}
```

```
def fun6[T](xs0: List[T], n0: Int) = {
  var n: Int = n0
  var xs: List[T] = xs0
  var ys: List[T] = Nil
  while (!xs.isEmpty && n > 0) {
    ys = xs.head :: ys
    xs = xs.tail
    n = n - 1
  }
  ys
}
```

(formulaire sur la page suivante -->)

## Formulaire

Les méthodes suivantes sont disponibles pour les réponses de cet exercice :

```
abstract class List[T] {
  def map[U](f: T => U): List[U]
  def flatMap[U](f: T => List[U]): List[U]
  def reduceRight(f: (T, T) => T): T
  def reduceLeft(f: (T, T) => T): T
  def foldRight[U](x: U)(f: (T, U) => U): U
  def foldLeft[U](x: U)(f: (U, T) => U): U

  def take(n: Int): List[T]
  def drop(n: Int): List[T]
  def splitAt(n: Int): (List[T], List[T])
  def reverse: List[T]
  def sortWith(lt: (T, T) => Boolean): List[T]
  def zip(xs: List[U]): List[(T, U)]

  def filter(p: T => Boolean): List[T]
  def forall(p: T => Boolean): Boolean
  def exists(p: T => Boolean): Boolean
  def partition(p: T => Boolean): (List[T], List[T])
}
```

- `map` applique une fonction à chaque élément
- `flatMap` applique une fonction à chaque élément et retourne la concaténation de toutes les listes résultantes
- Les méthodes `reduce` appliquent une opération sur les éléments par paires
- Les méthodes `fold` appliquent une opération sur chaque élément et un accumulateur

- `take` retourne les `n` premiers éléments
- `drop` retourne la liste sans les `n` premiers éléments
- `splitAt` retourne la liste des n premiers éléments et la liste sans les n premiers éléments
- `reverse` retourne l'inverse de la liste
- `sortWith` trie la liste par rapport à la fonction `lt`
- `zip` crée une liste de couples à partir de deux listes

- `filter` retourne les éléments qui satisfient le prédicat `p`
- `forall` et `exists` testent si le prédicat `p` est vrai pur chaque / pour au moins un élément
- `partition` regroupe les éléments de la liste qui satisfont le prédicat `p`, et ceux qui ne le satisfont pas