Examen intermédiaire

Programmation IV

11 mai 2005

Nom : _____

Prénom : _____

Section : _____

Exercice	Points	Points obtenus
1	10	
2	15	
3	10	
4	15	
Total	50	

Exercice 1 : Manipulation de listes (10 points)

La méthode `zip` permet de combiner deux listes en une liste de paires. Soient `n` la taille de `xs` et `m` la taille de `ys`, la taille de `xs.zip(ys)` est alors égale à `minimum(n, m)`.

La méthode `zip` de la classe `List` est définie comme suit :

```
abstract class List[A] {
  //...
  def zip[B](that: List[B]): List[Pair[A,B]] =
    if (this.isEmpty || that.isEmpty) Nil
    else Pair(this.head, that.head) :: this.tail.zip(that.tail);
}
```

Partie 1

Écrivez une fonction plus générale `zipWith` qui combine deux listes `xs` et `ys` au moyen d'une fonction `f` donnée :

```
  def zipWith[A,B,C](xs: List[A], ys: List[B], f: (A,B) => C): List[C] =
    // à compléter..
```

Partie 2

Écrivez une fonction `zipAll` qui combine deux listes en une liste de paires. Soient `n` la taille de `xs` et `m` la taille de `ys`, la taille de `zipAll(xs, ys, x, y)` est alors égale à `maximum(n, m)`. Les éléments manquants dans l'une des deux listes sont donnés par les valeurs `x` et `y`.

```
  def zipAll[A,B](xs: List[A], ys: List[B], x: A, y: B): List[Pair[A,B]] =
    // à compléter..
```

Exercice 2 : Preuve par induction structurelle (15 points)

Soient les deux méthodes `append` (concaténation de deux listes) et `mapFun` (application d'une fonction aux éléments d'une liste) suivantes :

```
def append[A](xs: List[A], ys: List[A]): List[A] = xs match {
  case Nil => ys                                  // 1ère clause
  case head :: tail => head :: append(tail, ys)   // 2ème clause
}

def mapFun[A,B](xs: List[A], f: A => B): List[B] = xs match {
  case Nil => Nil                                 // 1ère clause
  case head :: tail => f(head) :: mapFun(tail, f) // 2ème clause
}
```

démontrez l'égalité

```
mapFun(append(xs, ys), f) = append(mapFun(xs, f), mapFun(ys, f))
```

par *induction structurelle* sur `xs`.

Remarque : justifiez *chaque* étape de votre raisonnement comme présenté dans le cours, c.-à.-d. en indiquant par exemple "(selon 1ère clause de `mapFun`)".

Exercice 3 : Triangle de Pascal (10 points)

On souhaite implémenter une fonction `pascal` qui retourne une liste de listes d'entiers contenant les éléments du *triangle de Pascal* de taille `n` (`n` positif) :

```
type Row = List[Int];
type Triangle = List[Row];

def nextRow(xs: Row): Row = // à compléter (voir partie 1)

def pascal(n: int): Triangle = // à compléter (voir partie 2)
```

Par exemple `pascal(7)` retourne le résultat suivant :

```
List(List(1,7,21,35,35,21,7,1),
     List(1,6,15,20,15,6,1),
     List(1,5,10,10,5,1),
     List(1,4,6,4,1),    // ...
     List(1,3,3,1),      // row 3
     List(1,2,1),        // row 2
     List(1,1),          // row 1
     List(1))            // row 0
```

Partie 1

Écrivez tout d'abord une fonction `nextRow` qui calcule la ligne qui suit une ligne donnée row_{i-1} de la façon suivante :

	x_0	x_1	x_2	..	x_n	0	(left shifted row_{i-1})
$+$	0	x_0	x_1	..	x_{n-1}	x_n	(right shifted row_{i-1})
$=$	x_0	$x_1 + x_0$	$x_2 + x_1$	..	$x_n + x_{n-1}$	x_n	(row_i)

Par exemple la ligne row_3 s'obtient ainsi :

	1	2	1	0
$+$	0	1	2	1
$=$	1	3	3	1

Partie 2

Écrivez ensuite la fonction `pascal` en utilisant la fonction `nextRow` définie plus haut.

Indication : il est possible d'écrire une solution récursive terminale en utilisant une fonction auxiliaire, mais une solution différente est également acceptée.

Exercice 4 : Abstraction de données (15 points)

On souhaite réaliser un paquetage offrant les opérations arithmétiques telles que l'addition, la soustraction et le produit scalaire sur les polynômes ainsi que leur évaluation pour une valeur donnée `x` de type `double`.

Voici un exemple d'utilisation de la classe à implanter :

```
val p = new Polynomial(List(2, 3));  // p(x) = 2 + 3*x
val q = new Polynomial(List(1, 3));  // q(x) = 1 + 3*x
val r = p + q;                       // addition
val s = p - q;                       // soustraction
val u = q * 2;                       // produit scalaire
System.out.println(p(2.0));          // évaluation (<=> p.apply(2.0))
```

Complétez la classe `Polynomial` ci-dessous sachant que le paramètre `xs` contient les coefficients c_0, c_1, .., c_n où c_n correspond au coefficient de degré n :

```
class Polynomial(xs: List[double]) extends Function[double, double] {
  val coeffs: List[double] = xs;

  def +(that: Polynomial): Polynomial = // à compléter..




  def -(that: Polynomial): Polynomial = // à compléter..




  def *(n: double): Polynomial = // à compléter..




  def apply(x: double): double = // à compléter..




}
```