Examen intermédiaire

Programmation IV

17 mai 2006

Nom : _____

Prénom : _____

Section : _____

Exercice	Points	Points obtenus
1	15	
2	15	
3	15	
4	25	
Total	70	

Exercice 1 : Manipulation de listes (15 points)

Partie 1

Un *suffixe* d'une liste *xs* est une liste de longueur n ($0 \leq n \leq$ `xs.length`) qui contient les n derniers éléments de *xs*. Par example, $List()$ et $List(3,5)$ sont des suffixes de $List(2,3,5)$. Complétez la définition de la fonction récursive `suffixes` pour qu'elle retourne tous les suffixes d'une liste par ordre *décroissant* de leur longueur, comme dans l'exemple suivant.

```
suffixes(List(2, 3, 5)) =
  List(List(2, 3, 5), List(3, 5), List(5), List())
```

Voici le prototype de la fonction `suffixes` que vous devez écrire.

```
def suffixes[A](xs: List[A]): List[List[A]] = xs match {
  case List()   => ...
  case y :: ys  => ...
}
```

Partie 2

Un *préfixe* d'une liste *xs* est une liste de longueur n ($0 \leq n \leq$ `xs.length`) qui contient les n premiers éléments de *xs*. Par exemple, $List()$ et $List(2,3)$ sont des préfixes de $List(2,3,5)$. Complétez la définition de la fonction récursive `prefixes` pour qu'elle retourne tous les préfixes d'une liste par ordre *croissant* de leur longueur, comme dans l'exemple suivant.

```
prefixes(List(2, 3, 5)) =
   List(List(), List(2), List(2, 3), List(2, 3, 5))
```

Voici le prototype de la fonction `prefixes` que vous devez écrire.

```
def prefixes[A](xs: List[A]): List[List[A]] = xs match {
  case List()   => ...
  case y :: ys  => ...
}
```

Exercice 2 : Preuve inductive (15 points)

Prouvez, par induction structurelle sur la variable `xs`, l'égalité suivante.

```
rev(xs ::: ys) = rev(ys) ::: rev(xs)
```

Justifiez chaque étape en vous référant uniquement aux lemmes et définitions suivants :

```
        Nil ::: ys = ys                          // app1
  (x :: xs) ::: ys = x :: (xs ::: ys)            // app2

(xs ::: ys) ::: zs = xs ::: (ys ::: zs)          // assoc
     (xs ::: Nil) = xs                           // neutre

def rev[A](l: List[A]): List[A] = l match {
  case Nil         => Nil                        // rev1
  case x :: xs     => rev(xs) ::: List(x)        // rev2
}
```

Exercice 3 : Ni trop cher, ni trop lourd (15 points)

Dans un restaurant, on cherche à générer automatiquement la liste de tous les menus ayant une valeur énergétique inférieure à un certain seuil *et* un prix limité.
Chaque plat est une instance de la classe `FoodItem`.

```
case class FoodItem (
  name:     String,
  price:    Double,
  calories: Int,
  kind:     Kind)
```

Un plat est composé :
- d'un nom, indiqué par le champ `name`
- d'un prix, indiqué par le champ `price`
- d'une valeur énergétique, indiquée par le champ `calories`
- d'un type, indiqué par le champ `kind`.

Le type d'un plat est une des instances pré-définie de la classe `Kind`.

```
class Kind
val Starter    = new Kind()
val MainCourse = new Kind()
val Desert     = new Kind()
```

Un menu est composé de trois plats de types distincts : une entrée, un plat principal et un dessert. Il est membre du type `Menu`.

```
type Menu = Triple[FoodItem, FoodItem, FoodItem]
```

Partie 1

Écrivez en Scala une fonction `genMenus` qui, étant donné
– une liste des plats disponibles et
– pour un menu, le nombre maximum de calories et le prix maximum,
génère la liste de tous les menus satisfaisant ces contraintes. Vous devez utiliser la construction **`for`**.
Voici le prototype de la fonction `genMenus` que vous devez écrire.

```
def genMenus(food: List[FoodItem],
             maxCalories: Int,
             maxPrice: Double) : List[Menu] = ...
```

Partie 2

Écrivez encore une fois cette fonction *sans utiliser* la construction **`for`** (en traduisant votre solution pour la partie 1).

Exercice 4 : Queues fonctionnelles (25 points)

Une queue fonctionnelle q est une structure séquentielle de données. Elle est dotée des trois opérations de base suivantes.
– `head(q)` retourne le premier élément de la queue.
– `tail(q)` retourne une queue contenant tous les éléments de `q` à l'exception du premier.
– `append(q, x)` retourne une queue contenant tous les éléments de `q`, suivi de l'élément `x`.
On peut facilement implanter les queues fonctionnelles à l'aide de listes.

```
type Queue[A] = List[A]
def head[A](q: Queue[A]): A = q.head
def tail[A](q: Queue[A]): Queue[A] = q.tail
def append[A](q: Queue[A], x: A): Queue[A] = q ::: List(x)
```

Partie 1

Quelle est la complexité (en terme de N) de l'exécution d'un programme qui ajoute N éléments à une queue initialement vide, puis exécute alternativement N fois l'opération `head` et N fois l'opération `tail` ?

Partie 2

On souhaite implanter les queues fonctionnelles de manière plus efficace en représentant une queue par une paire des listes `Pair(xs,ys)` où
– `xs` contient le préfixe de la queue,
– `ys` contient le reste de la queue, en ordre inverse.
Dans ce cas, la liste des éléments s'obtient grâce à l'expression suivante.

```
xs ::: ys.reverse
```

L'opération `append` est implantée comme suit.

```
type Queue[A] = Pair[List[A], List[A]]
def append[A](A: Queue[A], x: A): Queue[A] = q match {
  case Pair(xs,ys) => Pair(xs, x::ys)
}
```

Implantez les opérations `head` et `tail` (avec la même signature que précédemment).

Partie 3

Quelle est la complexité du programme décrit dans la partie 1 avec l'implémentation obtenue dans la partie 2 ?