Programmation avancée

Examen final

jeudi 18 décembre 2008

Nom : _____

Prénom : _____

Section : _____

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. Les exercices en ont le même nombre mais n'ont probablement pas la même difficulté.

Votre attention La donnée de chaque exercice est précisément formulée. Si vous ne la comprenez pas, vous perdrez des points.

Nos contraintes Certains exercices ont des *contraintes supplémentaires* pour obtenir le maximum de points. Si vous n'y arrivez pas, essayez sans contraintes: il y a des points aussi pour ce genre de solutions.

Exercice	**Points**	**Points obtenus**
1	10	
2	10	
3	10	
4	10	
Total	40	

Exercice 1 : Jeux de mots : c'est `for` (10 points)

Considérons trois fonctions qui prennent en argument un dictionnaire sous forme d'une liste de mots, où un mot est représenté par une liste de caractères : (`List[List[Char]]`) :

1. `palindromes` retourne la liste de tous les palindromes du dictionnaire. Un palindrome est un mot dont l'ordre des caractères reste le même qu'on le lise de gauche à droite ou de droite à gauche. "kayak" est une palindrome. `palindromes` a la signature suivante.

   ```
   def palindromes(xs: List[List[Char]]): List[List[Char]] = ...
   ```

2. `commence` retourne toutes les paires de mots qui commencent par le meme caractère. Si le dictionnaire contient les mots "ara", "beta", et "ado", les paires de mots qui commencent par le même caractère sont "ara"–"ado" et "ado"–"ara". Vous pouvez assumer que le dictionnaire ne contient pas de duplicatats. `commence` a la signature suivante.

   ```
   def commence(xs: List[List[Char]]):
     List[Pair[List[Char], List[Char]]] = ...
   ```

Votre tâche pour cet exercice

1. Vous implantez `palindromes` en utilisant des compréhensions `for`.
2. Puis en utilisant des fonctions d'ordre supérieur.
3. Vous implantez `commence` en utilisant des fonctions d'ordre supérieur.

Contraintes supplémentaires

– Votre solution est implantée de *façon fonctionnelle*, c'est-à-dire sans utiliser de variables.

Exercice 2 : Aidez le CERN, encore une fois (10 points)

Les données du *BlackTrack* font le bonheur des physiciens du CERN. Il est toutefois apparu que certains trous noirs sont un peu trop volumineux : au cours de leur existence, ils détruisent chacun un petit morceau de terre genevoise. Les autorités n'y sont pas fondamentalement opposées, mais il est quand même question de dédommagement.

Un contrat à été signé, qui prévoit un paiement pour chaque trou noir dont la capacité de destruction est d'au moins 73% plus grande que la moyenne de celle des trous précédents. Le cas du premier trou n'a pas été réglé.

Le *BlackTrack* à été modifié pour émettre un flot dont chaque élément représente la capacité de destruction (en mottes par seconde) d'un trou noir. Le n ème élément du flot correspond à celle du n ème trou.

Tous les flots dans cet exercice sont de la forme `Stream[Double]`.

Votre tâche pour cet exercice

1. Vous implantez la fonction `moyenne` qui prend en argument le flot émis par le *BlackTrack* et qui retourne un flot dont l'éléments à la position n est la capacité de destruction moyenne du premier au n ème trou. `moyenne` a la structure suivante.

```
def moyenne(trous: Stream[Double]): Stream[Double] = {
  def moyenneImpl(trous: Stream[Doule],
                  n: Int,
                  moyenneAvant: Double): Stream[Doule] = ...
  Stream.cons(trous.head, moyenneImpl(...))
}
```

2. Vous implantez la fonction `paimentTrou` qui prend en argument le flot émis par le *BlackTrack* ainsi que le résultat de `moyenne` et qui retourne un flot contenant la capacité de destruction uniquement des trous noirs pour lesquels le CERN devra payer. La méthode `zip` sur les flots peut vous aider pour cela.

Contraintes supplémentaires

- Votre solution est implantée de *façon fonctionnelle*, c'est-à-dire sans utiliser de variables.
- Votre solution utilise l'*évaluation retardée* ou paresseuse, c'est-à-dire qu'elle permet la lecture de l'élément de tête du flot retourné sans que cela n'entraîne la lecture de l'entier des flots en arguments.

Exercice 3 : La Preuve par Induction (10 points)

L'égalité suivante définit la propriété d'*idempotence* des listes.

```
xs filter p filter p == xs filter p
```

On considère une implantation de `filter` qui permet de poser les égalités suivantes.

```
     Nil filter p  ==  Nil
(x :: xs) filter p  ==  if (p(x)) x :: (xs filter p)
                        else xs filter p
```

Votre tâche pour cet exercice

1. Vous prouvez que le filtrage des listes avec l'implantation considérée est idempotent. Pour cela, vous montrez que l'égalité définissant l'idempotence tient pour tout type `A`, pour toute liste `xs` de type `List[A]`, et pour tout prédicat `p` de type `A => Boolean`.

Exercice 4 : Tri rapide en Scala et en Prolog (10 points)

Le tri rapide (*quicksort*) est une méthode de tri récursive. Elle consiste à choisir un élément *pivot* et placer dans une liste tous les éléments qui lui sont inférieurs, dans une autre tous ceux qui lui sont supérieurs. On parle de "partitionnement". Le processus s'applique récursivement aux listes des éléments inférieurs et supérieurs. Les deux listes et le pivot sont ensuite concaténés. Considérerez un algorithme qui prend toujours l'élément de tête comme pivot.

Pour la partie Prolog, vous pouvez utiliser les prédicats suivants.

- `append(Xs, Ys, Zs)` : "la liste `Zs` est la concaténation de `Xs` et `Ys`"
- `lt(X, Y)` : "le nombre `X` est plus petit que `Y`"
- `ge(X, Y)` : "le nombre `X` est plus grand que ou égal à `Y`"

Votre tâche en Scala

1. Vous implantez `partition` qui prend comme paramètres une liste d'entiers et le pivot, et qui retourne deux listes. La première est celle des éléments plus petit que le pivot, la seconde celle des éléments plus grand ou égaux. `partition` a la signature suivante.

```
def partition(lss: List[Int], pivot: Int):
  Pair[List[Int], List[Int]] = ...
```

2. Vous implantez `quicksort` qui a la structure suivante.

```
def quicksort(lss: List[Int]): List[Int] = lss match {
    case Nil => ...
    case pivot :: ls => ...
  }
```

Votre tâche en Prolog

1. Le prédicat `partition(Xs, P, Ys, Zs)` se lit "`Ys` contient les éléments de `Xs` plus petit que `P`, `Zs` les éléments de `Xs` plus grands que ou égaux à `P`". Complétez la définition suivante.

```
partition([], P, [] []).
partition([X|Xs], P, [X|Ys], Zs) :- ...
partition([X|Xs], P, Ys, [X|Zs]) :- ...
```

2. Le prédicat `quicksort(Xs, Ys)` se lit "`Ys` contient les éléments de `Xs` ordonnés du plus petit au plus grand". Complétez la définition suivante.

```
quicksort([], []).
quicksort([X|Xs], Ys) :- ...
```