Programmation avancée

Examen final

jeudi 17 décembre 2009

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. En effet, nous ne pensons pas que tous les exercices ont la même difficulté, même s'ils ont le même nombre de points.

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	10	
3	10	
4	10	
Total	40	

Exercice 1 : Monotonie (10 points)

Étant donné une liste non-vide d'entiers, on cherche à calculer la longueur de la plus longue sous-liste croissante. Par exemple, si la liste est `List(1, 2, -1, 2, 3, 4, 3, 4, 2, 10)`, la plus longue sous-liste croissante est `List(-1, 2, 3, 4)` et sa longueur est 4.

Toute sous-liste peut être obtenue comme le *préfixe d'un suffixe* de la liste originale. Ainsi, on peut calculer les sous-listes croissantes comme le plus long préfixe croissant de chaque suffixe.

Pour mémoire, un préfixe (respectivement suffixe) d'une liste A est une liste qui, étant donné un $k \geq 0$, contient les k premiers (respectivement derniers) éléments de A dans le même ordre. `List(1, 2, -1, 2, 3, 4, 3, 4, 2, 10)`, `List(4, 3, 4, 2, 10)`, `List(10)`, ou `List()` sont quelques suffixes de la liste donnée en exemple.

Votre tâche pour cet exercice

1. Écrivez la fonction `longestMonoPrefix` qui retourne la longueur du plus long préfixe croissant d'une liste non-vide.

   ```
   def longestMonoPrefix(xsss: List[(Int, Int)]): Int
   ```

   Afin de tester la croissance des éléments, il faut comparer les éléments de la liste avec leur successeur immédiat. Pour faciliter cela, la fonction `longestMonoPrefix` travaille sur une liste `xsss` de paires faisants correspondre un élément avec son successeur. Cette liste s'obtient, pour une list `xs`, par l'expression "`xs zip xs.tail`". Remarquez que cette liste a une longueur inférieure de un à la liste originale et peut donc être vide.

2. Écrivez une fonction qui calcule tous les suffixes d'une liste `xs` non-vide donnée. La signature de cette fonction est la suivante :

   ```
   def suffixes[T](xs: List[T]): List[List[T]]
   ```

3. À l'aide des deux fonctions précédentes, écrivez une fonction qui calcule la longueur de la plus longue sous-liste croissante d'une liste `xs` non-vide donnée. La signature de cette fonction est la suivante :

   ```
   def maxMonoSub(xs: List[Int]): Int
   ```

Indication Votre solution peut utiliser la fonction qui retourne le plus grand élément d'une liste d'entiers et dont la signature est la suivante :

```
def max(xs: List[Int]): Int = ...
```

Exercice 2 : La bourse ou la vie (10 points)

La bourse des matières premières *PolyNex* modernise son système informatique en utilisant Scala. Le système contient déjà les entités suivantes.

`trait Matiere`
Une matière première qui peut être vendu ou achetée.

`trait Courtier`
Un courtier, qui peut vendre et acheter des marchandises.

`case class Offre(matiere: Matiere, courtier: Courtier)`
Une offre pour une matière par un courtier. Elle peut être une offre de vente ou d'achat suivant son contexte. On admet que toutes les offres se font sur les mêmes quantités standards.

`type OffreS = Stream[Offre]`
Un flux infini d'offres. En fonction de son contexte, le flux contient uniquement des offres de ventes ou uniquement des offres d'achats.

`case class Transaction(matiere: Matiere, acheteur: Courtier, vendeur: Courtier)`
Une transaction. La bourse génère une transaction pour une offre de vente et une offre d'achat. La transaction correspond à la mise en relation du courtier ayant fait l'offre de vente (le vendeur) et celui ayant fait l'offre d'achat (acheteur), dans le but d'acheter une quantité standard d'une matière. La génération d'une transaction clôt l'offre de vente et l'offre d'achat. Une offre clôturée ne peut pas faire partie d'une autre transaction.

Votre tâche pour cet exercice

1. Écrivez une fonction qui trouve dans un flux d'offres la première offre pour une matière donnée. La fonction retourne le courtier qui fait l'offre ainsi qu'un nouveau flux sans l'offre trouvée. La signature de cette fonction est la suivante :

   ```
   def trouveOffre(matiere: Matiere, offres: OffreS): (Courtier, OffreS)
   ```

2. Écrivez une fonction qui, pour un flux d'offres d'achats et pour un flux d'offres de ventes, retourne un flux de transactions qui permet de clôturer toutes les offres. Le deux flux d'offres sont infinis et contiennent une infinité d'offres pour chaque matière. La signature de cette fonction est la suivante :

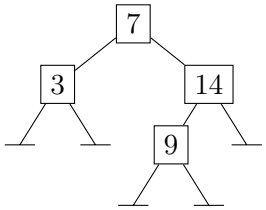
   ```
   def transactions(achats: OffreS, ventes: OffreS): Stream[Transaction]
   ```

Exercice 3 : Prolog est en ordre (10 points)

Les ensembles de nombres peuvent être représentés, en Prolog, sous forme d'arbre binaire à l'aide de deux prédicats récursifs (similaires aux classes `Empty` et `NonEmpty` du premier projet) :

- `node(X, L, R)` représente un noeud de l'arbre. Dans ce prédicat, `X` est une valeur (nombre) attachée au noeud ; `L` et `R` sont les sous-arbres gauche et droite, respectivement.
- `leaf` représente les feuilles de l'arbre et ne contient aucune valeur.

L'expression `node(7,node(3, leaf, leaf), node(14, node(9, leaf, leaf), leaf))` représente l'ensemble $\{3, 7, 14, 9\}$ et correspond à l'arbre suivant :



Votre tâche pour cet exercice

1. Définissez le prédicat `tree_sum(T, S)`. Dans ce prédicat, `T` est un ensemble de nombres de la forme décrite au dessus ; `S` est la somme des nombres de cet ensemble.
2. Définissez le prédicat `tree_flat(T, L)`. Dans ce prédicat, `T` est un ensemble de nombres ; `S` est une liste qui contient tous les éléments de l'ensemble en ordre de parcours *infixe*.

   Pour mémoire, un parcours infixe sur un arbre binaire est un parcours qui, pour un noeud de l'arbre, visite d'abord l'arbre gauche, puis la valeur du noeud, puis l'arbre droite. Un parcours infixe sur l'arbre ci-dessus visitera les noeuds dans l'ordre $3, 7, 9, 14$.

Indication Votre solution peut utiliser le prédicat `append` vu en cours, ainsi que le prédicat `add(X, Y, Z)` dans lequel X, Y et Z sont des nombres tels que $X + Y = Z$.

Exercice 4 : Remplacements en chaîne (10 points)

On cherche à écrire un interpréteur pour des expressions qui manipulent les chaînes de caractères. Ces expressions sont représentées par la classe :

```
abstract class StringExpr {}
```

Dans un premier temps, seules deux formes d'expressions existent : une chaîne constante et une expression de remplacement. Cette dernière remplace dans une expression toutes les occurrences d'une chaîne (s'il y en a) par une autre.

Les expressions peuvent être nichées les unes dans les autres. Elles permettent, par exemple, l'expression décrite comme : «Remplacer "bar" par "foo" dans la chaine résultante du remplacement de "baz" par "bar" dans la chaîne constante "bazbar".» L'interprétation de cette expression donne «foofoo».

Votre tâche pour cet exercice

1. Sans rien ajouter à la signature de `StringExpr`, concevez un ensemble de *classes cas* pour représenter les expressions qui manipulent les chaînes de caractères.
2. Écrivez la fonction qui interprète ces expressions et qui a la signature suivante.
   ```
   def eval(expr: StringExpr): String
   ```
3. Ajoutez une forme d'expression qui supprime toutes les occurrences d'une chaîne (s'il y en a) dans une expression.
4. Écrivez une nouvelle fonction qui transforme une expression contenant des suppressions en une *expression équivalente* pouvant être interprétée par la fonction `eval` originale.

Indication La classe `String` contient la méthode suivante, que votre solution peut utiliser :

```
class String {
  /** Remplace chaque occurrence de 'occ' par 'rempl' dans cette chaine. */
  def replaceAll(occ: String, rempl: String): String = ...
}
```