Examen final

Programmation IV

15 juin 2005

Nom : _____

Prénom : _____

Section : _____

Exercice	Points	Points obtenus
Total	0	

Exercice 1 : Nombres de Hamming (15 points)

On appelle *nombres de Hamming* la suite infinie (flot) de nombres entiers satisfaisant les propriétés suivantes :

1. La suite est strictement croissante ;
2. La suite commence par le nombre 1 ;
3. Si la suite contient le nombre `x`, alors elle contient également les nombres `2*x`, `3*x` et `5*x` ;
4. La suite ne contient *aucun* autre nombre.

Concrètement on obtient :

```
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, ..
```

Partie 1

(*6 points*) Écrivez une fonction `merge` qui fusionne deux flots (*streams*) `xs` et `ys` croissants en un nouveau flot lui-même croissant (en éliminant les éventuels doublons) :

```
def merge(xs: Stream[Int], ys: Stream[Int]): Stream[Int] = {
  // à compléter..









}
```

Partie 2

(*9 points*) Définissez la valeur `hamming` qui représente la suite infinie des nombres de Hamming en utilisant la fonction `merge` de la partie 1 :

```
val hamming: Stream[Int] =
  Stream.cons(1, // à compléter..









  );
```

Indication : la valeur de `hamming` se définit de manière récursive.

Exercice 2 : Preuve par induction structurelle (10 points)

On définit les termes au moyen des deux classes suivantes :

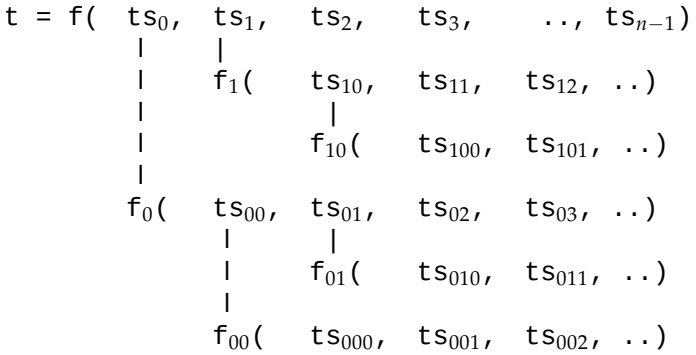
```
abstract class Term;
case class Constr(f: Symbol, ts: List[Term]) extends Term;
```

La fonction `at` permet d'accéder aux différents sous-termes d'un terme au moyen d'une liste d'indices (chemin) indiquant leur position.

```
def at(t: Term, pos: List[Int]): Term = Pair(t, pos) match {
  case Pair(_, Nil) => t                                  // 1ère clause
  case Pair(Constr(f, ts), i::rest) => at(ts(i), rest)  // 2ème clause
}
```

Remarque : Pour rappel l'expression `ts(i)` dans le code ci-dessus permet d'accéder au i-ème élément de la liste `ts`.

Pour le terme `t` ci-dessous :



on obtient par exemple :

```
at(t, List(1, 0))    = ts10
at(t, List(0, 1, 1)) = ts011
```

Étant donnés la fonction `at` définie plus haut et les deux lemmes :

```
       Nil ::: ys   =              ys   // lemme 1
(x :: xs) ::: ys   = x :: (xs ::: ys)  // lemme 2
```

démontrez l'égalité suivante :

```
at(at(t, u), v) = at(t, u ::: v)
```

par *induction structurelle* sur `u`.

L'affirmation ci-dessus signifie qu'un noeud de l'arbre - un terme est en fait un arbre - atteint par un chemin en deux parties `u` et `v` peut aussi être atteint par un seul chemin codé `u:::v`.

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

Exercice 3 : Parcourir un arbre binaire (15 points)

Partie 1

(*7 points*) Les méthodes `foldLeft` et `foldRight` de la classe `List` permettent de combiner les éléments d'une liste au moyen d'une fonction donnée.

De façon similaire la méthode `fold` de la classe `Tree` ci-dessous permet de combiner les noeuds d'un arbre au moyen d'une fonction donnée. Complétez la méthode `fold` :

```
abstract class Tree[+A] {
  def fold[B](z: B)(f: (A, B, B) => B): B = this match {
    // à compléter..









  }
}
case object Empty extends Tree[All];
case class Node[A](x: A, left: Tree[A], right: Tree[A]) extends Tree[A];
```

Par exemple la somme des éléments de l'arbre `t` est égale à `22` :

```
val t = Node(5, Node(2, Empty, Empty),
                Node(8, Node(7, Empty, Empty), Empty));
System.out.println(t.fold(0)((x, l, r) => x + l + r));
```

Partie 2

(*8 points*) Écrivez une fonction `collect` qui utilise `fold` (cf. partie 1) pour construire une liste contenant les éléments d'un arbre binaire pris dans l'ordre préfixe (*preorder*), infixe (*inorder*) et suffixe (*postorder*) :

```
val PREORDER = -1; val INORDER = 0; val POSTORDER = 1;

def collect[A](t: Tree[A], order: Int): List[A] = {
  // à compléter..









}
```

La fonction `collect` appliquée à l'arbre `t` retourne par exemple :

```
collect(t, PREORDER)    ==>   List(5,2,8,7)
collect(t, INORDER)     ==>   List(2,5,7,8)
collect(t, POSTORDER)   ==>   List(2,7,8,5)
```

Exercice 4 : Lisp et Prolog (10 points)

Partie 1

(*5 points*) Définissez l'analogue en Lisp de la fonction Scala `flatMap` suivante :

```
def flatMap[A, B](xs: List[A]; f: A => List[B]): List[B] = xs match {
  case Nil => Nil
  case head :: tail => f(head) ::: flatMap(tail, f)
}
```

Partie 2

(*5 points*) Définissez en Prolog un prédicat `insert(X, Xs, Ys)` qui exprime que :

1. `Ys` est une liste de nombres qui contient comme éléments `X` et les éléments de la liste `Xs`,
2. `Ys` est triée par ordre croissant (elle peut contenir des doublons).

Vous pouvez supposer que
- la liste `Xs` est triée,
- il existe un prédicat `leq(X, Y)` qui est vrai ssi `X` est plus petit ou égal à `Y`.

Exercice 5 : Neurones impulsionnels (10 points)

Nous nous proposons dans cet exercice de simuler un réseau de neurones impulsionnels. Voici comment cela fonctionne.

Chaque neurone est connecté à d'autres neurones à qui il peut envoyer des impulsions. Lorsqu'un neurone reçoit une impulsion (*spike*), son degré d'excitation va augmenter de la valeur de l'intensité de l'impulsion. Lorsque le degré d'excitation d'un neurone dépasse un seuil (*threshold*) propre au neurone, le neurone va émettre une impulsion à tous les neurones auxquels il est connecté. Son excitation va aussi tomber à zéro. Toutes les impulsions produites par un neurone ont la même intensité propre au neurone.

Complétez les méthodes `receiveSpike` et `connect` du code ci-dessous pour qu'il se comporte comme décrit ci-dessus. Il n'est pas nécessaire de traiter le cas où le réseau de neurones comporterait un cycle.

```
class Neuron(name: String, threshold: Int, outputIntensity: Int) {
  var outputNeurons: List[Neuron] = Nil;
  var excitationLevel: Int = 0;

  /** Appelé lorsqu'une impulsion arrive sur le neurone. */
  def receiveSpike(intensity: Int): Unit = {
    // à compléter









  }

  /** Connecte un neurone à soi-même */
  def connect(neuron: Neuron): Unit = {
    // à compléter
```

```
  }
}
```

La classe `Neuron` doit pouvoir être utilisée comme dans l'exemple suivant.

```
object Brain with Application {
  val out = new Neuron("Out", 4, 10);
  val middle = new Neuron("Middle", 5, 6);
  val start = new Neuron("Start", 0, 4);
  start.connect(middle);
  start.connect(middle);
  middle.connect(out);
  start.receiveSpike(10);
}
```

Dans cet exemple, le neurone `start` recevra une impulsion extérieure d'intensité 10, le neurone `middle` deux impulsions d'intensité 4 venant du neurone `start` et le neurone `out` une impulsion d'intensité 6 venant du neurone `middle`.