Programming Principles

Midterm Solution

Friday, November 9th 2012

Exercise 1: Grades (10 points)

```
type Result = (String, Int)
type Quiz = List[Result]
type Course = List[Quiz]
```

Part 1: Quiz Grade Means

```
def quizMeans(course: Course): List[Float] = course map { quiz =>
  val sum = quiz.foldLeft(0.f) {
    (acc: Float, result: (String, Int)) => acc + result._2
    // or shorter: _ + _._2
  }
  sum / quiz.length
}
```

Part 2: Student Grade Means

```
def studentMeans(course: Course): Map[String, Float] = course.flatten groupBy(_._1) map {
  case (student, namegrades) =>
    val sum = namegrades.foldLeft(0.f) {
      (acc: Float, result: (String, Int)) => acc + result._2
      // or: _ + _._2
    }
    (student, sum / namegrades.length)
}
```

Part 3: Top Students

```
def topStudents(course: Course, n: Int): List[String] =
  studentMeans(course).toList.sortBy(-_._2).map(_._1).take(n)
```

Part 4: Passing Students

```
def passingStudents(course: Course): List[String] =
  studentMeans(course).toList.filter(_._2 >= 3.5).map(_._1)

// or: studentMeans(course).toList.sortBy(-_._2).takeWhile(_._2 >= 3.0).map(_._1)
```

Part 5: Grade histogram

```
type Histogram = Map[Int, Int]

def histogram(course: Course): Histogram = {
  val studentsPerGrade = studentMeans(course).groupBy {
    case (name, mean) => math.round(mean)
  }
  studentsPerGrade map {
    case (grade, students) => (grade, students.size)
  }
}
```

Exercise 2: Hamming Numbers (10 points)

First definition

```
lazy val powersOfSixty: Stream[BigInt] = 1 #:: powersOfSixty.map(_ * 60)
lazy val fromOne: Stream[BigInt] = 1 #:: fromOne.map(_ + 1)


def log2(d: Double): Double = math.log(d) / math.log(2)
def maxExponent(i: BigInt): Int = math.ceil(log2(i.doubleValue)).toInt


def dividesPowerOfSixty(i: BigInt): Boolean =
  powersOfSixty.take(maxExponent(i) + 1).exists(_ % i == 0)


lazy val hamming1: Stream[BigInt] = fromOne.filter(dividesPowerOfSixty)
```

Second definition

```
def mergeAscending(s1: Stream[BigInt], s2: Stream[BigInt]): Stream[BigInt] = {
  if (s1.head < s2.head) s1.head #:: mergeAscending(s1.tail, s2) else
  if (s2.head < s1.head) s2.head #:: mergeAscending(s1, s2.tail) else
  mergeAscending(s1, s2.tail) // drop duplicates
}


lazy val hamming2: Stream[BigInt] =
  1 #:: mergeAscending(hamming2 map (_ * 2),
                       mergeAscending(hamming2 map (_ * 3),
                                      hamming2 map (_ * 5)))
```

Comparing the two streams

```
val ok: Boolean = hamming1 zip hamming2 take 100 forall {case (a, b) => a == b}
```

Exercise 3: Abstract Syntax Trees (10 points)

Part 1

```
plus(1, plus(2, 3)) --->
  Apply("plus", List(Number(1), Apply("plus", List(Number(2), Number(3)))))

plus(plus(1, 2), 3) --->
  Apply("plus", List(Apply("plus", List(Number(1), Number(2))), Number(3)))
```

The two trees are not the same.

Part 2

```
def prettyprint(tree: Tree): String = tree match {
  case Number(n) => n.toString
  case Apply(fun, args) => fun + "(" + mkString(args map prettyprint, ", ") + ")"
}

def mkString(ss: List[String], sep: String): String = ss match {
  case Nil => ""
  case s :: Nil => s
  case hd :: tl => hd + sep + mkString(tl, sep)
}
```

Part 3

```
def divsByZero(tree: Tree): List[Tree] = tree match {
  case Number(_) => Nil
  case Apply("div", List(x, Number(0))) => divsByZero(x) :+ tree
  case Apply(_, args) => flatten(args map divsByZero)
}

def flatten(trees: List[List[Tree]]): List[Tree] = trees match {
  case Nil => Nil
  case hd :: tl => hd ++ flatten(tl)
}
```