Programming Principles

Midterm Exam

Friday, November 9th 2012

First Name: _____

Last Name: _____

Your points are *precious*, don't let them go to waste!

Your Name Work that can't be attributed to you is lost: write your name on each sheet of the exam.

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
Total	30	

Exercise 1: Grades (10 points)

In this exercise we analyze the quiz grades of students on a course. Each student is represented by his/her name - a unique `String`. Grades are represented as `Int`s from `1` to `6`. On each quiz the student receives a certain grade - this is the student's `Result` on a particular quiz:

```
type Result = (String, Int)
```

where the first element of the pair is the student's name, and the second is their grade.

A quiz is defined as a list of results for each student:

```
type Quiz = List[Result]
```

A quiz will always have grades for all the students enrolled in the course.

Finally, a course is simply a list of quizzes:

```
type Course = List[Quiz]
```

Here is an example of the course called Programming Principles, which consists of five students and three quizzes:

```
val programmingPrinciples = List(
  List(
    ("Johnny Depp", 1),
    ("Matt Damon", 3),
    ("Scott Bakula", 6),
    ("Lukas Rytz", 6),
    ("Mickey Mouse", 1)
  ),
  List(
    ("Johnny Depp", 3),
    ("Matt Damon", 2),
    ("Scott Bakula", 6),
    ("Lukas Rytz", 5),
    ("Mickey Mouse", 1)
  ),
  List(
    ("Johnny Depp", 4),
    ("Matt Damon", 4),
    ("Scott Bakula", 6),
    ("Lukas Rytz", 6),
    ("Mickey Mouse", 1)
  )
)
```

We suggest that you consider using the following collection methods for solving this exercise:

`def map[U](f: T => U): List[U]`	a new collection which contains the elements of this collection, transformed by the function `f`
`def map[P, Q] (f: (K, V) => (P, Q)): Map[P, Q]`	a new map which contains the key-value pairs of this map, transformed by the function `f`
`def filter(p: T => Boolean): List[T]`	a new collection which contains only those elements of this collection which satisfy the predicate `p`
`def filter(p: (K, V) => Boolean): Map[K, V]`	a new map which contains only those key-value pairs of this map which satisfy the predicate `p`
`def take(n: Int): List[T]`	a new collection which contains only the first `n` elements of this collection (or all if size is less than `n`)
`def takeWhile(p: T => Boolean): List[T]`	a new collection which contains the longest prefix of the elements of this collection which satisfy the predicate `p`
`def sortBy[U](f: T => U): List[T]`	a new sequence which contains the elements of this sequence in a sorted order determined by the function `f` – useful when elements of the collection are not ordered (e.g. `Tuple2`), but they can be mapped into some other element type which is ordered (e.g. `Double`)
`def groupBy[K](f: T => K): Map[K, List[T]]`	a map which contains all the elements of this collection grouped by the mapping function `f` – the elements which mapped into the same key are put into the same group
`def flatten: List[T]`	given that this collection has collections as its elements, this returns a flattened collections (e.g. `List(List(1, 2), List(3))` becomes `List(1, 2, 3)`)
`def toMap: Map[K, V]`	given that this collection contains pairs of type `(K, V)`, this returns a map with keys of type `K` and values of type `V`
`def toList: List[T]`	converts this collection into a list – if this collection is a `Map[K, V]`, then the resulting list will contain pairs `(K, V)`
`def foldLeft[U](z: U) (f: (U, T) => U): U`	folds the elements of this collection by applying `f` to all the elements
`def foldLeft[U](z: U) (f: (U, (K, V)) => U): U`	folds the key-value pairs of this map by applying `f` to all the elements
`def length: List[T]`	returns the length of this sequence
`def size: List[T]`	returns the size of this collection (same as `length`, but works for any collection type, not just sequences)

The methods listed above work for all collection types, not just lists – for example, you can replace the return type `List` with a return type `Map` in the type signatures above. However, methods `sortBy` and `length` can only be called on sequences, like `List`.

You are allowed use other methods as well, but the solution should be the most elegant if you use the suggested methods. Each part of the exercise is independent of the other parts – you do not have to solve part 1 to solve part 2. But in any part you can reuse the methods defined in the earlier parts.

Part 1: Quiz Grade Means

Write the method `quizMeans` which, given a `course`, computes the list of mean grades of all the quizzes in that course:

```
def quizMeans(course: Course): List[Float] = ???
```

A mean grade of a quiz is defined as the sum of the grades of all the students divided by the number of students.

Part 2: Student Grade Means

Write the method `studentMeans` which, given a `course`, computes the mean grades of all the students in that course:

```
def studentMeans(course: Course): Map[String, Float] = ???
```

The method should return a `Map` mapping each student to his mean grade. The mean grade of a student is defined as the sum of his grades on all the quizzes divided by the number of quizzes.

Part 3: Top Students

Write the method `topStudents` which, given a `course` and the number of top students `n`, computes the `n` top students on the course.

```
def topStudents(course: Course, n: Int): List[String] = ???
```

The top `n` students are those `n` students with the highest mean grades. You can assume that no two students have the same mean grade.

Hint: in any part of the exercise you can use one of the methods defined earlier.

Part 4: Passing Students

Write the method `passingStudents` which, given a `course`, computes the list of the students which passed the course:

```
def passingStudents(course: Course): List[String] = ???
```

The students that passed the course are those whose mean grade is greater than or equal to `3.5`.

Part 5: Grade histogram

In the final part of this exercise, you will compute the number of students per each final grade. A final grade is obtained by computing the mean grade of the student and then rounding it to the nearest integer.

The assignment from a grade to the number of students with that grade is called a **grade histogram**, and we model it with a `Histogram` datatype:

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

For example, if there were `3` students `John Rambo`, `Judge Dredd` and `Rocky Balboa`, and they have the mean grades `3.3`, `3.2` and `6.0`, respectively, then the histogram is:

```
Map(3 -> 2, 6 -> 1)
```

because there were `2` students with a mean grade which rounds to `3`, and a single student with a result of `6`.

Note that if zero students have a certain grade, then you do not have to include it in the histogram. Above, none of the students have a grade `5`, so the key `5` is not in the `Map`.

Write the method `histogram` which, given a `course`, computes the grade histogram of the course:

```
def histogram(course: Course): Histogram = ???
```

Hint: consider using the `math.round(x: Float)` method from the Scala standard library which rounds a real number to an integer number.

Exercise 2: Hamming Numbers (10 points)

In this exercise we will work with Hamming numbers. On Wikipedia there are two definitions:

- Hamming numbers are numbers that entirely divide a power of 60
- Hamming numbers have the form $2^i \cdot 3^j \cdot 5^k$, where i, j and k are ≥ 0

As an example, we verify that 48 is a Hamming number. We list the powers of 60 (on the left) and see that 48 divides 60^2 without remainder (on the right):

$$\begin{array}{rcl@{\qquad}rcl} 60^0 & = & 1 & 1 \bmod 48 & = & 1 \\ 60^1 & = & 60 & 60 \bmod 48 & = & 12 \\ 60^2 & = & 3600 & 3600 \bmod 48 & = & 0 \end{array}$$

We can easily verify that 48 also matches the second definition of Hamming numbers: $48 = 2^4 \cdot 3 = 2^4 \cdot 3^1 \cdot 5^0$, so we have $i = 4$, $j = 1$ and $k = 0$.

The goal of this exercise is to verify that the two definitions are indeed equivalent. We will implement the two definitions and compare the results. Since we are dealing with large numbers we use `BigInt` instead of `Int` in this exercise. You can work with `BigInt`s just like with ordinary integers.

The following methods of the class `Stream[T]` might be useful for solving this exercise.

`def #::[T](x: T): Stream[T]`	prepends the element `x` to this stream
`def head: T`	the first element of this stream
`def tail: Stream[T]`	this stream without its first element
`def zip[U](that: Stream[U]): Stream[(T,U)]`	a stream whose elements are pairs of the elements of the streams `this` and `that`
`def take(n: Int): Stream[T]`	a finite stream consisting of the first `n` elements of this stream
`def forall(p: T => Boolean): Boolean`	`true` if `p` holds for every element in this stream
`def exists(p: T => Boolean): Boolean`	`true` if `p` holds for some element in this stream
`def map[U](f: T => U): Stream[U]`	a stream which contains the elements of this stream, transformed by the function f
`def filter(p: T => Boolean): Stream[T]`	the stream of elements from this stream that satisfy predicate `p`

First definition

To implement the first definition, we begin with an infinte stream consisting of all powers of 60:

```
val powersOfSixty: Stream[BigInt] = ???
powersOfSixty.take(4).toList  // should give List(1, 60, 3600, 216000)
```

We also need the infinite stream that counts from 1 upwards:

```
val fromOne: Stream[BigInt] = ???
fromOne.take(3).toList  // should give List(1, 2, 3)
```

The Hamming numbers can now be obtained by filtering the stream `fromOne` and keeping only numbers which divide a power of sixty. We write a predicate function `dividesPowerOfSixty(i: BigInt): Boolean` that returns true if `i` is a divisor of 60^n for some $n \geq 0$.

However, we can obviously not look at all the powers of 60, there are infinitely many. Instead we use a conservative approximation: to find out if i divides any power of 60, the largest power of 60 we have to check is $p = 60^{\lceil \log_2(i) \rceil}$. If i doesn't divide p, then it also does not divide any larger power of 60.

```
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 = ???
```

Now you can define the first stream of hamming numbers:

```
val hamming1: Stream[BigInt] = ???
```

Second definition

The second definition of Hamming numbers will lead to a more efficient solution than the first one: instead of checking a property for each number the numbers are computed directly. An algorithmic description to compute Hamming numbers is the following:

- The number 1 is a Hamming number
- If x is a Hamming number, the numbers $2 \cdot x$, $3 \cdot x$ and $5 \cdot x$ are also Hamming numbers

If we implement this using a recursive value, we need a way to merge two streams into one. This job is done by `mergeAscending`: it merges two monotonically growing streams into one by taking the smaller of the two head elements and merging the remaining elements recursively.

If both input streams have the same head element, then one of the two should be dropped (in other words, the resulting stream should not have duplicates).

You can assume that both input streams are infinite.

```
def mergeAscending(s1: Stream[BigInt], s2: Stream[BigInt]): Stream[BigInt] = ???
mergeAscending(1 #:: 4 #:: ...,
               2 #:: 4 #:: 5 #:: ...) // should give 1 #:: 2 #:: 4 #:: 5 #:: ...
```

Note that you can merge three streams into one by using `mergeAscending` twice:

```
mergeAscending(s1, mergeAscending(s2, s3))
```

Now we are ready to implement the stream of Hamming numbers:

```
val hamming2: Stream[BigInt] = ???
```

Comparing the two streams

To see if the two streams `hamming1` and `hamming2` contain the same elements, write an expression that evaluates to `true` if the first 100 elements of the two streams are the same, and `false` otherwise:

```
val ok: Boolean = ???
```

Exercise 3: Abstract Syntax Trees (10 points)

The toy programming language Funny supports decimal numbers, e.g. `1` or `42`, and function applications, e.g. `plus(2, -2)` or `div(1, plus(2, -2))`. The language is as simple as that. Funny doesn't support conditional expressions, operators or even function definitions - it's just numbers and calls to functions.

When designing a compiler for a programming language it is common to represent programs as abstract syntax trees, recursive data structures which express code hierarchically.

In `funnyc`, the Funny compiler, abstract syntax trees are encoded using two case classes: `Number` and `Apply`, which both inherit from a common superclass `Tree`.

```
abstract class Tree
case class Number(n: Int) extends Tree
case class Apply(fun: String, args: List[Tree]) extends Tree
```

These case classes can represent the aforementioned Funny programs as follows:

Program	Abstract syntax tree
`1`	`Number(1)`
`42`	`Number(42)`
`plus(2, -2)`	`Apply("plus", List(Number(2), Number(-2)))`
`div(1, plus(2, -2))`	`Apply("div", List(Number(1), Apply("plus", List(Number(2), Number(-2)))))`

Part 1

Convert the following two Funny programs into their abstract syntax tree representation: a) `plus(1, plus(2, 3))`, b) `plus(plus(1, 2), 3)`. Are the resulting trees the same?

Part 2

Back then, when Funny only supported numbers, the compiler was easy to grasp. However with the recent addition of function calls, developers of `funnyc` started experiencing problems with debugging.

Help the developers by writing a prettyprinting function for Funny abstract syntax trees with the signature: `def prettyprint(tree: Tree): String`. The prettyprinter should convert syntax trees into equivalent Funny source code, like shown in the left column of the example table above.

Note. Use the `toString` method to convert numbers to strings. For example, `42.toString` produces `"42"`.

Part 3

Despite being perceived by some as overly complex, Funny has quickly gained traction on Twitter. New feature requests are abundant. One of those is preventing division by zero errors, which happen when the function named `div` is called with two arguments, the second of which is `0`.

Write a function `def divsByZero(tree: Tree): List[Tree]`, which given an abstract syntax tree of a Funny program, returns all divisions by zero which occur in that program.

Note. You only need to detect the most blatant division by zero mistakes, i.e. just the function applications having the form of `div(..., 0)`. You don't have to account for expressions like `div(1, plus(-2, 2))`.