Functional Programming

Midterm Exam

Friday, November 6 2015

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.

Stay Functional You are strictly forbidden to use mutable state (vars) or mutable collections in your solutions.

Some Help The last page of this exam contains an appendix which is useful for formulating your solutions. You can detach this page and keep it aside.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
4	10	
Total	40	

Exercise 1: Pairwise Differences (10 points)

In this exercise, we seek to encode lists of integers as successive differences of consecutive pairs of elements. We will then look at rebuilding the original list, given the list of differences.

Part 1: Calculating the list of differences (5 pts)

Implement the `differences` function, which takes a list of integers, and returns a list of pairwise differences of the elements of this list:

```
def differences(ls: List[Int]): List[Int] = ???
```

Calling `differences(xs)` must return a list `ys` such that:

- `ys.size == ls.size`
- If `ls.nonEmpty` then `ys.head == ls.head`
- For all `0 < i < ys.size, ys(i) == xs(i) - xs(i - 1)`

Here are some example outputs of the function:

```
scala> differences(Nil)
res0: List[Int] = Nil

scala> differences(List(1))
res1: List[Int] = List(1)

scala> differences(List(1, -2, 3, -4, 5, -6))
res2: List[Int] = List(1, -3, 5, -7, 9, -11)
```

Part 2: Rebuilding the original list (5 pts)

Implement the `rebuildList` function, which takes a list corresponding to the list of differences, and rebuilds the original list from it. In other words, for all `xs: List[Int], rebuildList(differences(xs)) == xs`:

```
def rebuildList(ls: List[Int]): List[Int] = ???
```

Here are some example outputs of the function:

```
scala> rebuildList(differences(Nil))
res3: List[Int] = Nil

scala> rebuildList(differences(List(1)))
res4: List[Int] = List(1)

scala> rebuildList(differences(List(1, -2, 3, -4, 5, -6)))
res5: List[Int] = List(1, -2, 3, -4, 5, -6)
```

Hint: The API in the appendix is your friend.

Exercise 2: Recursion on Trees (10 points)

Consider the following implementation of a `Tree`:

```
abstract class Tree
case class Node(left: Tree, elem: Int, right: Tree) extends Tree
case object Leaf extends Tree
```

Part 1: Recursively computing min and max (3 pts)

Write a recursive function computing both the minimum and maximum among the elements of a tree. The function should accept a `Node` and return a pair of integers which should respectively be the smallest, and greatest, elements of the tree. The function takes a `Node` as a parameter to ensure the tree has at least one element. **Note: In this part, you can make no assumption about the tree structure, in particular, do not assume it is a binary search tree.**

```
def computeMinMax(b: Node): (Int, Int) = ???
```

Hint: You might find the following functions useful:

- `Math.min(a, b)`
- `Math.max(a, b)`

Part 2: Determining if a tree is a binary search tree (7 pts)

We consider the problem of determining if a tree is a binary search tree (BST). Checking this property would be useful, for instance to check that a transformation on a given BST returns a BST.

Definition of BST: a tree is a BST when for **every** node, its element is greater than all the elements of the left subtree, and smaller than all the elements of the right subtree.

Implement a function which determines if a tree is a BST or not. You are free to write helper functions if you need them.

```
def isBinarySearchTree(b: Tree): Boolean = ???
```

Hint: you might find the values `Int.MinValue` and `Int.MaxValue`, which are the smallest, and respectively greatest, values in the `Int` range, useful.

Exercise 3: Subtyping (10 points)

Given the following hierarchy of classes:

```
class Pair[+A, +B]
class Iterable[+A]
class Map[A, +B] extends Iterable[Pair[A, B]]
```

Consider also the following typing relationships for `W`, `V`, `X` and `Y`:

- `V <: W`
- `Y <: X`

Fill in the subtyping relation between the types below. **Justify your answers**.

T1	?	T2
`Map[V, Y]`		`Map[W, X]`
`Iterable[Pair[V, Y]] => V`		`Map[V, Y] => V`
`Map[Map[V, Y], Y]`		`Map[Iterable[Pair[V, X]], X]`
`(Y => Y) => V`		`(X => Y) => W`
`W => (Y => X)`		`W => (V => Y)`

Exercise 4: Structural Induction (10 points)

Consider the following implementation of a binary `Tree` of `Int`s:

```
abstract class Tree
case class Node(left: Tree, elem: Int, right: Tree) extends Tree
case object Leaf extends Tree
```

which we intend to use as a *binary search tree*. Recall from the exercise sessions that a binary search tree is a representation for a set of `Int`s, in which elements of the left subtree are strictly smaller than `elem`, and elements of the right subtree are strictly greater than `elem`.

More formally, let us define `S(t)`, the mathematical set of elements represented by `t: Tree`, as:

```
S(Leaf) = {}
S(Node(l, e, r)) = S(l) U S(r) U {e}
```

(Note that the function `S` is well-defined for all `Tree`s; not only for valid *binary search* `Tree`s.)

We can now formally define the binary search tree property `Bst(t)` as a boolean function of a tree:

```
Bst(Leaf) = true
Bst(Node(l, e, r)) =
  Bst(l) &&
  Bst(r) &&
  (for all k in S(l) : k < e) &&
  (for all k in S(r) : k > e)
```

In the exercise sessions, you have implemented a function `add(t, v)`, which takes a binary search tree as input, and returns a new binary search tree with all the elements of `t` in it, plus `v`. Formally, we can write the *specification* of `add` as follows:

```
For all t: Tree such that Bst(t) and all v: Int:
  Bst(add(t, v)) &&
  S(add(t, v)) === S(t) U {v}
```

Here is an implementation of `add`:

```
def add(t: Tree, v: Int): Tree = t match {
  case Leaf                    => Node(Leaf, v, Leaf)
  case Node(l, e, r) if v == e => t
  case Node(l, e, r) if v < e  => Node(add(l, v), e, r)
  case Node(l, e, r) if v > e  => Node(l, e, add(r, v))
}
```

Your task, in this question, is to *prove* that the above implementation of `add` *satisfies* its specification.

From the implementation, we derive the following *axioms*:

```
(1)  add(Leaf,          v) === Node(Leaf, v, Leaf)
(2)  add(Node(l, e, r), v) === Node(l, v, r)           if v == e
(3)  add(Node(l, e, r), v) === Node(add(l, v), e, r)   if v < e
(4)  add(Node(l, e, r), v) === Node(l, e, add(r, v))   if v > e
```

Part 1: the set represented by the new tree satisfies the specification (8 pts)

Prove by *structural induction* the second part of the specification, i.e.,

```
For all t: Tree such that Bst(t) and all v: Int:
  S(add(t, v)) === S(t) U {v}
```

At *each* step, precisely indicate which “lemma” allows you to do the rewriting. For example, an axiom (and which one), the definition of `S(t)`, etc. Failing to do so will be penalized.

Note: do *not* prove the case `v > e`. It is a copy-paste of the case `v < e`, so we do not ask you to write it.

Part 2: the new tree is still a binary search tree (2 pts)

Prove by *structural induction* the first part of the specification, i.e.,

```
For all t: Tree such that Bst(t) and all v: Int:
  Bst(add(t, v))
```

Similarly, precisely indicate which lemma you use at each step, and do not prove the case `v > e`.

Appendix: Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful, on `List[A]`:

- `xs ++ (ys: List[A]): List[A]`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A`, or `xs(n: Int): A`: returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]`: returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean): List[A]`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
- `xs.flatMap[B](f: A => List[B]): List[B]`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going left to right.
- `xs.foldRight[B](z: B)(op: (A, B) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going right to left.
- `xs.map[B](f: A => B): List[B]`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
- `xs.nonEmpty: Boolean`: returns `true` if the list has at least one element, `false` otherwise.
- `xs.reverse: List[A]`: reverses the elements of the list `xs`.
- `xs.scan[B >: A](z: B)(op: (B, B) => B): List[B]`: produces a `List[B]` containing cumulative results of applying the operator `op` going left to right, with the start value `z`. The returning list contains 1 more element than the input list, the head being `z` itself.
- `xs.take(n: Int): List[A]`: returns a `List[A]` containing the first `n` elements of `xs`. If there are less than `n` elements in `xs`, returns these elements.
- `xs.zip(ys: List[B]): List[(A, B)]`: zips elements of `xs` and `ys` in a pairwise fashion. If one list is longer than the other one, remaining elements are discarded. Returns a `List[(A, B)]`.