Functional Programming

Midterm Exam

Friday, November 10 2017

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

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 cannot 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: K-Largest Elements (10 points)

Part I

Implement a function that inserts a given element elem into a sorted (in ascending order) list list. The resulting list should also be sorted in ascending order. Implement the function recursively.

```
def insert(elem: Int, list: List[Int]): List[Int] = ???
```

Part II

Implement a function that takes as arguments a non-negative integer k and a ~~sorted (in ascending order)~~ list list , then returns the k-largest elements as a sorted list in ascending order.

For instance:

```
findKLargestElements(4)(List(5, 2, 7, 1, 3, 5)) == List(3, 5, 5, 7)
findKLargestElements(5)(List(3, 1, 2)) == List(1, 2, 3)
findKLargestElements(0)(List(1, 2, 3)) == Nil
```

You are required to use foldLeft in your solution. Your solution should be in $O(n \cdot k)$. You can make use of the insert function from last part, even if you have not implemented it. You may also make use of functions presented in the appendix.

```
def findKLargestElements(k: Int)(list: List[Int]): List[Int] = ???
```

Exercise 2: For comprehensions (10 points)

In a game of Othello (also known as Reversi in French-speaking countries), when a player puts a token on a square of the board, we have to look in all directions around that square to find which squares should be "flipped" (i.e., be stolen from the opponent). We implement this in a method computeFlips, taking the position of square, and returning a list of all the squares that should be flipped:

```
final case class Square(x: Int, y: Int)

def computeFlips(square: Square): List[Square] = {
  List(-1, 0, 1).flatMap { i =>
    List(-1, 0, 1).filter { j =>
      i != 0 || j != 0
    }.flatMap { j =>
      computeFlipsInDirection(square, i, j)
    }
  }
}

def computeFlipsInDirection(square: Square,
    dirX: Int, dirY: Int): List[Square] = {
  // omitted
}
```

Rewrite the method computeFlips to use one for comprehension instead of maps, flatMaps and filter s. The resulting for comprehension should of course have the same result as the expression above for all value of square. However, it is not necessary that it *desugars* exactly to the expression above.

Exercise 3: Variance (10 points)

Given the following hierarchy of classes:

```
class Chan[-P, +R]
class FixedChan[P, +R] extends Chan[P, R]
```

Recall that + means covariance, - means contravariance and no +/- means invariance (i.e. neither covariance nor contravariance).

Consider also the following typing relationships for A, B, X and Y:

- A <: B
- X <: Y

Fill in the subtyping relation between the types below using symbols:

- <: in case T1 is a subtype of T2;
- >: in case T1 is a supertype of T2;
- × in case T1 is neither a supertype nor a supertype of T2.

Wrong answers will incur negative points. Enter your solution only when you are sure.

T1	?	T2
`A => Y`		`A => X`
`A => Y`		`B => X`
`FixedChan[A, X]`		`FixedChan[A, Y]`
`FixedChan[A, X]`		`FixedChan[B, X]`
`FixedChan[A, X]`		`Chan[A, Y]`
`Chan[A, Y]`		`FixedChan[B, X]`

Exercise 4: Structural Induction (10 points)

For this exercise, you will be working on binary trees, defined as follows:

```
sealed abstract class Tree[+A] {
 def toList: List[A] = ... // Implementation not shown
 def treeMap[B](f: A => B): Tree[B] = ... // Implementation not shown
}
case class Node[+A](left: Tree[A], value: A, right: Tree[A]) extends Tree[A]
case object Leaf extends Tree[Nothing]
```

Property

Your goal is to prove that the following property holds for any tree of type Tree[A] and any function f of type A => B:

```
tree.treeMap(f).toList === tree.toList.map(f)
```

Axioms and Theorems

To prove the property, you may use the following axioms, derived from the definitions of toList, treeMap and map:

1) Leaf.toList === Nil
2) Node(left, x, right).toList === left.toList ++ (x :: right.toList)
3) Leaf.treeMap(f) === Leaf
4) Node(left, x, right).treeMap(f) === Node(left.treeMap(f), f(x), right.treeMap(f))
5) Nil.map(f) === Nil
6) (x :: xs).map(f) === f(x) :: xs.map(f)

In addition, you may make use of the following theorem (without proving it):

7) (xs ++ ys).map(f) === xs.map(f) ++ ys.map(f)

Note: Be *very precise* in your proof:

- Clearly state which axiom, theorem or hypothesis you use at each step.
- Use only 1 axiom, theorem or hypothesis at each step, and only once.
- Underline the part of an equation on which you apply your axiom, theorem or hypothesis.
- Make sure to state what you want to prove, and what your induction hypotheses are, if any.

Appendix: Scala Standard Library Methods

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

- `xs.head: A`: returns the first element of the list. Throws an exception if the list is empty.
- `xs.tail: List[A]`: returns the list `xs` without its first element. Throws an exception if the list is empty.
- `x :: (xs: List[A]): List[A]`: prepends the element x to the left of xs, returning a 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.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.size: Int`: returns the number of elements in the list.
- `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)].
- `xs.toSet: Set[A]`: returns a set of type Set[A] that contains all elements from the list xs. Note that the resulting set will contain no duplicates and may therefore be smaller than the original list.