Functional Programming

Midterm Exam

Friday, November 11 2016

First Name: _____

Last Name: _____

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

Your Name Work that cannot 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 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: List functions (10 points)

You are asked to implement the following `List` functions using only the specified `List` API methods. You are also allowed to use the `reverse` method in any subquestion, should you see it fit. If you need another method of `List`, you need to reimplement it as part of your answer.

Please refer to the appendix on the last page as a reminder for the behavior of the given `List` API methods.

(a) Implement `scanLeft` using only `foldLeft`, `Nil` and `:: (cons)`.

```
def scanLeft[A, B](xs: List[A])(z: B)(op: (B, A) => B): List[B] = ???
```

(b) Implement `flatMap` using only `foldRight`, `Nil` and `:: (cons)`.

```
def flatMap[A, B](xs: List[A])(f: A => List[B]): List[B] = ???
```

Exercise 2: Subtyping (10 points)

Given the following hierarchy of classes:

```
trait Producer[+A]
trait Consumer[-A]
trait Factory[+A, -B] extends Producer[A] with Consumer[B]
```

Recall that `+` means covariance and `-` means contravariance.

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 using symbols:

- `<:` in case `T1` is a subtype of `T2`;
- `>:` in case `T1` is a supertype of `T2`;
- $\times$ in case `T1` is neither a supertype nor a supertype of `T2`.

Justify your answers.

T1	?	T2
`Producer[X]`		`Producer[Y]`
`Producer[Consumer[X]]`		`Producer[Consumer[Y]]`
`Factory[Producer[X], X]`		`Factory[Factory[Y, Y], Y]`
`Factory[Y, Y] => Producer[V]`		`Factory[Y, X] => Producer[W]`
`List[Factory[Y, Y]]`		`List[Consumer[X]]`

Exercise 3: Proof by induction (10 points)

We want to implement a function `sum(list: List[Int]): Int`, which returns the sum of the elements of a list of `Int`s. We can easily *specify* that function as follows:

```
(1)  sum(Nil) === 0
(2)  sum(x :: xs) === x + sum(xs)
```

If we naively translate this specification into a Scala implementation, we end up with a uselessly non-tail recursive function. Besides, doing the recursion ourselves is wasteful. Instead, we implement it using `foldLeft`:

```
def betterSum(list: List[Int]): Int =
  list.foldLeft(0)(add)

def add(a: Int, b: Int): Int = a + b
```

However, that implementation is not trivially correct anymore. We would like to *prove* that it is correct for all lists of integers. In other words, we want to prove that

```
list.foldLeft(0)(add) === sum(list)
```

for all lists of integers.

In addition to the specification of `sum` (1-2), you may use the following axioms:

```
(3)  Nil.foldLeft(z)(f) === z
(4)  (x :: xs).foldLeft(z)(f) === xs.foldLeft(f(z, x))(f)
(5)  add(a, b) === a + b
(6)  a + b === b + a
(7)  (a + b) + c === a + (b + c)
(8)  a + 0 === a
```

Axioms 3-5 follow from the implementations of `foldLeft` and `add`. Axioms 6-8 encode well-known properties of `Int.+`: commutativity, associativity, and neutral element.

Your task: prove the following lemma by structural induction:

```
list.foldLeft(z)(add) === z + sum(list)
```

From that lemma, we can "trivially" (with the help of axioms 6 and 8) derive that `betterSum`'s implementation is correct by substituting `0` for `z` in the lemma. You are not asked to do that last bit.

Note: Be *very* precise in your proof:

- Clearly state which axiom/lemma you use *at each step*, and when/if you use the induction hypothesis. Even using "trivial" properties of `Int.+` such as commutativity must be justified using the corresponding axiom.
- Use only 1 axiom/lemma/hypothesis at each step, and only once. Applying 2 axioms requires 2 steps.
- Underline the part of an equation on which you apply your axiom.
- Make sure to state what you want to prove, and what your induction hypothesis is, if any.

Exercise 4: Graph Reachability (10 points)

Consider the following case class definitions:

```
case class Node(id: Int)
case class Edge(from: Node, to: Node)
```

Let us represent a directed graph `G` as the list of all its edges (of type `List[Edge]`). We are interested in computing the set of all nodes reachable in **exactly** `n` steps from a set of initial nodes.

1. Write a `reachable` function with the following signature to provide this functionality:

   ```
   def reachable(n: Int, init: Set[Node], edges: List[Edge]): Set[Node]
   ```

   You can assume that `n >= 0`.

2. Given a set of nodes within the graph, use the function you defined above to compute the subset of these nodes that belong to a cycle of size 3 within the graph.

   ```
   def cycles3(nodes: Set[Node], edges: List[Edge]): Set[Node]
   ```

Appendix: Scala Standard Library Methods

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

- `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.scanLeft[B](z: B)(op: (B, A) => 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. For example, `List("A", "B", "C").scanLeft("z")(_ + _)` returns `List("z", "zA", "zAB", "zABC")`. `scanLeft` is similar to `foldLeft`, except that, instead of returning only the last value of the accumulator, it returns a list of all the intermediate values of the accumulator.
- `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)]`.
- `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.

You may also need the following functions on `Set[A]`:

- `s1 ++ (s2: Set[A]): Set[A]`: union of sets `s1` and `s2`.
- `s.filter(p: A => Boolean): Set[A]`: returns all elements from `s` that satisfy the predicate `p` as a `Set[A]`.
- `s.flatMap[B](f: A => List[B]): Set[B]`: applies `f` to every element of the set `s`, and flattens the result into a `Set[B]`. You can also use `s.flatMap[B])(f: A => Set[B]): Set[B]`.
- `s.map[B](f: A => B): Set[B]`: applies `f` to every element of the set `s` and returns a new set of type `Set[B]`.
- `s.isEmpty: Boolean`: returns `true` is the set `s` contains no elements, `false` otherwise.
- `s.nonEmpty: Boolean`: returns `true` if the set `s` has at least one element,`false` otherwise.
- `s.toList: List[A]`: returns a list of type `List[A]` that contains all elements from the set `s` is some arbitrary order. Note that the resulting list will contain no duplicates.
- `s.foldLeft[B](z: B)(op: (B, A) => B): B`: same as `s.toList.foldLeft(z)(op)`.
- `s.foldRight[B](z: B)(op: (A, B) => B): B`: same as `s.toList.foldRight(z)(op)`.