# Functional Programming

Midterm Exam

Friday, November 7 2014

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.

**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 | 5 | |
| 2 | 5 | |
| 3 | 5 | |
| 4 | 5 | |
| 5 | 5 | |
| **Total** | 25 | |

# Exercise 1: Merging sorted lists (5 points)

The **mergesort** algorithm for sorting lists consists of two steps:

a) Split a given list in two, and recursively call the sort algorithm on both sub-lists.
b) Merge the two resulting **sorted** lists into a single, sorted list, in linear time.

In this exercise, we seek to implement the merge step of the sorting algorithm. This function should work on generic lists.

## Part 1: Starting recursive (2 points)

Implement the `merge` function. It takes two sorted lists, and a comparator function, and returns a single list, which contains the elements of the two input lists, sorted. The two input lists are also sorted according to the comparator function. The implementation of `merge` should be recursive (*not tail recursive*), and use pattern matching on lists:

```
def merge[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = ???
```

The `cmp` function takes two arguments, and returns `true` if the first argument is less than or equal to the second argument.

As an example,

```
merge(List(1, 4, 7, 10, 11), List(2, 3, 5, 7, 12, 16))((x, y) => x <= y)
```

should return

```
List(1, 2, 3, 4, 5, 7, 7, 10, 11, 12, 16)
```

## Part 2: Going tail-recursive (3 points)

Implement a function `merge2`, which is the same as `merge`, except it is tail-recursive.

```
def merge2[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = ???
```

***Note:*** Both `merge` and `merge1` are expected to have a linear running time.

## Exercise 2: Streams (5 points)

In this exercise, we seek to implement the following infinite streams:

1. (2 points) Define the function `iterate`, which takes two parameters, an initial value $x : T$ and a function $f : T \Rightarrow T$, and returns a stream. The first element of the stream is $x : T$, and the $n$-th element is the result of applying $f$ to the $(n-1)$-th element. The stream returned by `iterate` has the following structure: $x, f(x), f(f(x)), f(f(f(x))), \ldots, f^n(x), \ldots$.

   ```
   def iterate[T](x: T)(f: T => T): Stream[T] =
   ```

2. (3 points) Define the function `iterated` which takes a function $f : T \Rightarrow T$ as parameter and returns a stream. The first element of the stream is the identity function $id$, and the $n$-th element is the function composition of $f$ with the $(n-1)$-th element. The stream returned by `iterated` has the following structure: $id, f, f \circ f, f \circ f \circ f, \ldots, f^n, \ldots$.

   ```
   def iterated[T](f: T => T): Stream[T => T] =
   ```

## Exercise 3: Structural Induction (5 points)

We define the function `indexWhere` as follows:

```
def indexWhere[A](xs: List[A], f: A => Boolean): Int = {
  indexWhereAcc(xs, f, 0)
}

@tailrec
def indexWhereAcc[A](xs: List[A], f: A => Boolean, acc: Int): Int = xs match {
  case Nil     => acc
  case x :: xr =>
    if (f(x)) acc
    else indexWhereAcc(xr, f, acc+1)
}
```

Intuitively, `indexWhere` returns the index of the first element of `xs` that satisfies `f`, or the length of `xs` if there is no such element. Formally, its specification is as follows:

1. `indexWhere(Nil, f) === 0`
2. `indexWhere(x :: xr, f) === 0` if `f(x)` is true
3. `indexWhere(x :: xr, f) === 1 + indexWhere(xr, f)` if `f(x)` is false

Prove that `indexWhere` implements its specification using these three steps:

1. Write axioms for `indexWhereAcc` based on its cases
2. Prove the following lemma by structural induction:
   `indexWhereAcc(xs, f, n) === indexWhereAcc(xs, f, 0) + n`
3. Use this lemma to prove, by structural induction, the three results about `indexWhere`.

***Note:*** Be very precise in your proofs. Clearly state which axiom/lemma you use, and when/if you use the induction hypothesis.

# Exercise 4: Subtyping (5 points)

## Part 1: Union of Sets (2 points)

Given the following hierarchy of classes:

```
trait Fruit
class Apple extends Fruit
class Peach extends Fruit
```

And an abstract polymorphic set type:

```
trait Set[+A]
```

We seek to add a method `union` to the trait `Set` such that in the following expression:

```
val fruits = Set(new Apple).union(Set(new Peach))
```

the value `fruits` is of type `Set[Fruit]`. Provide a type signature for the `union` method. Note: It is sufficient to provide the type signature, you do not need to implement the body of `union`.

## Part 2: Function Conformance (3 points)

Given the following hierarchy of classes:

```
class A
class B extends A

class C
class D extends C
```

For each of the following definitions, say whether the expression on the right conforms to the declared type. In other words, do the declarations type check? Justify your answers. Note: there can be more than one conforming and/or non-conforming definitions.

1. `val x1: B => D = (b: A) => new D`
2. `val x2: A => C => D = (a: A) => (b: D) => new C`
3. `val x3: (D => B) => A = (db: D => A) => new B`

## Exercise 5: Flattening (5 points)

A member of your hacking team tried to implement a recursive data structure containing nested lists of integers. Unfortunately, they did not know much about types, case classes and inheritance, so they just nested the lists without any principle.

A sample value returned by their code looks something like this:

```
val ls: List[Any] = List(5, List(84, 12), List(3, List(-4, 7)))
```

After some frustration and some careful consideration, you conveniently decide you care only about the order of the integer values in the list, and not its exact structure. So you need to implement a function `flatten` that takes a list `l` as above, and returns a list of integers, in the order they appear in `l`:

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

For example, for `ls` above,

```
flatten(ls) == List(5, 84, 12, 3, -4, 7)
```

If the list contains something that is neither a list nor an integer, your code should throw a `MatchError`.

## Appendix: Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful:

- on `List[A]`:
  - `xs.map(f: A => B)`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
  - `xs.filter(p: A => Boolean)`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
  - `xs.flatMap(f: A => List[B])`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
  - `xs.reverse`: reverses the elements of the list `xs`.
  - `xs ++ (ys: List[A])`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- on `Stream[A]`:
  - `Stream.cons(a: A, as: Stream[A])`: constructs a stream where the first element is `a`, and the tail is the stream `as`.
  - `xs.map(f: A => B)`: applies `f` to every element of the stream `xs` and returns a new stream of type `Stream[B]`.
  - `xs.filter(p: A => Boolean)`: returns all elements from `xs` that satisfy the predicate `p` as a `Stream[A]`.
  - `xs.flatMap(f: A => Stream[B])`: applies `f` to every element of the stream `xs`, and flattens the result into a `Stream[B]`.
- on `Function1[A, B]`:
  - `f andThen (g: B => C)`: composes `f` and `g`: `(f andThen g)(a: A) == g(f(a))`
- `final class MatchError(obj: Any) extends RuntimeException`: the type of error thrown whenever an object `obj` does not match any pattern in a pattern match expression.