# Functional Programming

## Midterm Solution

Friday, November 8 2019

## Exercise 1: For-comprehensions (10 points)

### Question 1a. (5 points)

```
def lists(n: Int): Generator[List[Int]] =
  if (n <= 0)
    single(Nil)
  else
    for {
      head <- integers
      tail <- lists(n - 1)
    } yield head :: tail
```

### Question 1b. (2 points)

```
def listsUpTo(limit: Int): Generator[List[Int]] =
  for {
    n <- atMost(limit)
    list <- lists(n)
  } yield list
```

### Question 1c. (3 points)

```
def sortedLists(n: Int, minimum: Int): Generator[List[Int]] =
  if (n <= 0)
    single(Nil)
  else
    for {
      head <- atLeast(minimum)
      tail <- sortedLists(n - 1, head)
    } yield head :: tail
```

## Exercise 2: Structural Induction (10 points)

### Question 2a.

**`(xs ++ (x :: Nil)).map(f) === xs.map(f) ++ (f(x) :: Nil)`**

By structural induction on `xs`.

When `xs === Nil`:

```
(Nil ++ (x :: Nil)).map(f)
  === (x :: Nil).map(f) (by ++ (1))
  === f(x) :: Nil.map(f) (by map (4))
  === f(x) :: Nil (by map (3))
  === Nil ++ (f(x) :: Nil) (by ++ (1))
  === Nil.map(f) ++ (f(x) :: Nil) (by map (3))
```

When `xs === y :: ys`:

(Inductive hypothesis: **`(ys ++ (x :: Nil)).map(f) === ys.map(f) ++ (f(x) :: Nil)`**)

```
((y :: ys) ++ (x :: Nil)).map(f)
  === (y :: (ys ++ (x :: Nil))).map(f) (by ++ (2))
  === f(y) :: (ys ++ (x :: Nil)).map(f) (by map (4))
  === f(y) :: (ys.map(f) ++ (f(x) :: Nil)) (by inductive hypothesis)
  === (f(y) :: ys.map(f)) ++ (f(x) :: Nil) (by ++ (2))
  === (y :: ys).map(f) ++ (f(x) :: Nil) (by map (4))
```

### Question 2b.

**`xs.map(f).reverse === xs.reverse.map(f)`**

By structural induction on `xs`.

When `xs === Nil`:

```
Nil.reverse.map(f)
  === Nil.map(f) (by reverse (5))
  === Nil (by map (3))
  === Nil.reverse (by reverse (5))
  === Nil.map(f).reverse (by map (3))
```

When `xs === y :: ys`:

(Inductive hypothesis: **`ys.map(f).reverse === ys.reverse.map(f)`**)

```
(y :: ys).reverse.map(f)
  === (ys.reverse ++ (y :: Nil)).map(f) (by reverse (6))
  === ys.reverse.map(f) ++ (f(y) :: Nil) (by lemma)
  === ys.map(f).reverse ++ (f(y) :: Nil) (by hypothesis)
  === (f(y) :: ys.map(f)).reverse (by reverse (6))
  === (y :: ys).map(f).reverse (by map (4))
```

### Grading scheme

Grading scheme for both questions:

- 1 point for base case
- 4 points for the inductive case
- a small error (missing parenthesis) in the inductive case is -1 point
- a small error in the base case is -0.5 points
- misusing IH, badly misusing an axiom or inventing own axioms is -3 or -2 points, depending on how complete the rest of the solution is
- partial solutions are worth 0.5 and up to 2 points for the base and inductive case.

Incorrectly specifying the inductive hypothesis did *not* subtract from the grade, although correctly doing so added to points for the partial solution.

### Comments

The most common error was missing parenthesis in the inductive part of the second question. By axiom 4, `(y :: ys).map(f) ++ (f(x) :: Nil)` is equivalent to `(f(y) :: ys.map(f)) ++ (f(x) :: Nil)` and not to `f(y) :: (ys.map(f) ++ (f(x) :: Nil))`. The parenthesis here are crucial for determining which expression we are dealing with.

Read the inductive hypothesis for both exercises closely. For example, take a look at the IH for the second case: when `xs === y :: ys`, the IH is **`ys.map(f).reverse === ys.reverse.map(f)`**. Note that the IH only works for `ys`, not for `xs`. If IH could be applied to `xs`, the resulting proof would be circular: we would be assuming that **`xs.map(f).reverse === xs.reverse.map(f)`**, which is the precise thing we are trying to prove.

Additionally: structural induction is inherently tied to the *structure* (of `List`-s in this case). A `List` is defined to be either `Nil` or `y :: ys` for some `y` and `ys`, and therefore the inductive case *must* start by assuming that `xs === y :: ys`. While it seems reasonable to instead assume that `xs === ys ++ (y :: Nil)`, this does not directly follow from the definition of `List`. What one could do is prove that `y :: ys === zs ++ (z :: Nil)` for some `zs` and `z`, and only then use that to prove the inductive case.

## Exercise 3: Variance (10 points)

- `F[A] <: F[B]`
- `G[B] >: G[C]`
- `H[B] X H[A]`
- `F[A] <: F[C]`
- `B => B <: A => C`
- `A => C >: C => A`
- `C => B X A => A`
- `F[A] => B >: F[B] => A`
- `G[A] => F[A] <: G[B] => F[B]`
- `(A => C) => (C => A) <: (B => B) => (A => C)`

# Exercise 4: Pattern matching and recursion (10 points)

### Question 4a.

```
def size: Int =
  this match {
    case Empty() => 0
    case Layer(_, next) => 1 + 2 * next.size
  }
```

### Question 4b.

```
def map[B](f: A => B): Perfect[B] =
  this match {
    case Empty() => Empty()
    case Layer(elem, next) => Layer(f(elem), next.map { case (a, b) => (f(a), f(b)) })
  }
```

### Question 4c.

```
def toList: List[A] =
  this match {
    case Empty() => Nil
    case Layer(elem, next) => elem :: next.toList.flatMap { case (a, b) => a :: b :: Nil }
  }
```