Functional Programming

Midterm Exam

Friday, November 8 2019

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 return statements, mutable state (vars) and 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: For-comprehensions (10 points)

As seen in class, a `Generator[T]` can be used to generate values of type `T`:

```
trait Generator[+T] {
  /** Return a value of type ‘T‘ */
  def generate: T

  def map[S](f: T => S): Generator[S] = ...

  def flatMap[S](f: T => Generator[S]): Generator[S] = ...
}
```

In the questions below, you can make use of the following pre-existing generators:

```
/** Generates random integers. */
val integers: Generator[Int]

/** Generates random booleans. */
val booleans: Generator[Boolean]

/** Always generate the value ‘x‘. */
def single[T](x: T): Generator[T]

/** Generates random integers greater than or equal to ‘minimum‘. */
def atLeast(minimum: Int): Generator[Int]

/** Generates random integers less than or equal to ‘maximum‘. */
def atMost(maximum: Int): Generator[Int]
```

In the questions below, your solutions must not contain “`new Generator`” or “`extends Generator`” anywhere.

Question 1a. (5 points)

Implement a generator for lists of a fixed length `n`, your solution must contain a for-comprehension.

```
/** Generates lists of length ‘n‘, the elements of the lists are random integers.
 *  If n < 0, generates empty lists.
 */
def lists(n: Int): Generator[List[Int]] =
```

Question 1b. (2 points)

Implement a generator for lists of length `<=` `limit`. your solution must contain a for-comprehension.

```
/** Generates lists of a random length between 0 and ‘limit‘ included,
 *  the elements of the lists are random integers.
 *  If limit < 0, generates empty lists.
 */
def listsUpTo(limit: Int): Generator[List[Int]] =
```

Question 1c. (3 points)

Implement a generator for *sorted* lists of a fixed length `n`, your solution must contain a for-comprehension. Your solution **must not** contain any call to `.sort`, `.sorted` or `.sortBy`.

```
/** Generates lists of length ‘n‘, the elements of the lists are random integers.
 *  Each generated list is sorted in ascending order and contains no element
 *  smaller than ‘minimum‘.
 *  If n < 0, generates empty lists.
 */
def sortedLists(n: Int, minimum: Int): Generator[List[Int]] =
```

Exercise 2: Structural Induction (10 points)

In this exercise, you will be working the standard `List` data structure. Your goal is to prove that the following equivalence holds for all `xs` of type `List[A]` and any function `f` of type `A => B`:

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

Axioms. You may use the following axioms:

(1) `Nil ++ ys === ys`

(2) `(x :: xs) ++ ys === x :: (xs ++ ys)`

(3) `Nil.map(f) === Nil`

(4) `(x :: xs).map(f) === f(x) :: xs.map(f)`

(5) `Nil.reverse === Nil`

(6) `(x :: xs).reverse === xs.reverse ++ (x :: Nil)`

(7) `(xs ++ ys) ++ zs === xs ++ (ys ++ zs)`

Lemma. In question 3b, you may use the following lemma (which you will prove in question 3a):

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

Note: Be *very precise* in your proof:

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

Question 3a. (5 points)

Prove (8) `(xs ++ (x :: Nil)).map(f) === xs.map(f) ++ (f(x) :: Nil)`

Question 3b. (5 points)

Prove that `xs.map(f).reverse === xs.reverse.map(f)`

Exercise 3: Variance (10 points)

Given the following classes:

- `class F[+X]`
- `class G[-X]`
- `class H[X]`

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 and C:

- `A <: B`
- `B <: C`

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;
- `X` in case T1 is neither a supertype nor a supertype of T2.

Each correct answer is worth one point. Each incorrect answer deduces half a point.

T1	`<:`, `>:` or `X`	**T2**
`F[A]`	______	`F[B]`
`G[B]`	______	`G[C]`
`H[B]`	______	`H[A]`
`F[A]`	______	`F[C]`
`B => B`	______	`A => C`
`A => C`	______	`C => A`
`C => B`	______	`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)

In this exercise you will be working with perfectly balanced trees, defined as follows:

```
trait Perfect[A]
case class Empty[A]() extends Perfect[A]
case class Layer[A](elem: A, next: Perfect[(A, A)]) extends Perfect[A]
```

Perfect trees are always perfectly balanced by construction. Unlike the traditional tree data-structure, which consists of nodes and leaves, perfect trees are made of several layers of increasing size and a single empty tree at the bottom. For instance, the perfect tree containing the numbers 1 to 7 is defined as follows:

```
val exampleTree: Perfect[Int] =
  Layer(1,                       // 1st layer, with 1 element
    Layer((2, 3),                // 2nd layer, with 2 elements
      Layer(((4, 5), (6, 7)),    // 3rd layer, with 4 elements
        Empty()                  // empty tree at the bottom
      )
    )
  )
```

Note that each layer holds twice as many elements as its parent layer, and that elements are packed into nested pairs. This structure emerges from the fact that the `next` layer of `Perfect[A]` is defined to be a `Perfect[(A, A)]`, that is, a tree with pairs of `A`-s as its elements. We call this data-structure a perfectly balanced tree because it always contains exactly $2^n - 1$ elements, where n is the number of layers.

Your task will be to implement several methods of perfectly balanced trees. Your implementations must be written directly in the body of trait `Perfect`, as opposed to being in the body of the case classes `Empty` and `Layer`.

Question 4a. (3 points)

Implement the `size` method that counts the number of elements in the tree.

```
trait Perfect[A] {
  def size: Int =









}
```

Question 4b. (3 points)

Implement the `map` method that given a function `A => B`, build a new perfectly balanced tree by applying the function to all elements of the tree.

Hint: when recursively calling `map` on the next layer, pay attention to the argument's type, it should be `(A, A) => (B, B)` and not `A => B`.

```
trait Perfect[A] {
  def map[B](f: A => B): Perfect[B] =









}
```

Question 4c. (4 points)

Implement the `toList` method that transforms a perfectly balanced tree into a list. The order of the elements in the returned list should correspond to a layer by layer traversal of the tree. In other words, calling `.toList` on the example tree defined in the introduction should return `List(1, 2, 3, 4, 5, 6, 7)`.

Hint 1: When creating or concatenating lists, make sure that all elements are of the same type.

Hint 2: Don't worry about performance, aim for the simplest solution possible. A correct but slow solution will be given a full score, no extra point will be awarded for tail-recursive solutions.

```
trait Perfect[A] {
  def toList: List[A] =









}
```

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 discalaarded. 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.