Functional Programming

Midterm Exam

Wednesday, November 7 2018

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: List functions (10 points)

(a) Implement a function that takes a list `ls` as argument, and returns a list of all the suffixes of `ls`. That is, given a list `List(a,b,c,...)` it returns `List(List(a,b,c,...), List(b,c,...), List(c,...), List(...), ..., List())`. Implement the function recursively using only `Nil` (empty), `::` (cons) and pattern matching.

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

(b) Implement a function that takes a lists `ls` as argument and returns the length of the longest contiguous sequence of repeated elements in that list. For this second question, you are required to use `foldLeft` in your solution, and your solution should *not* be recursive.

For example:

```
longest(List(1, 2, 2, 5, 5, 5, 1, 1, 1)) == 3
```

```
def longest[A](ls: List[A]): Int = ???
```

Exercise 2: For-comprehensions (10 points)

You are given three classes (Student, Exam and Course which are defined below) and the method `generatePassedExams`, which from a given list of students and a list of courses, generates a list of students and all their successfully passed courses together with the corresponding grade. A course is considered as successfully passed if the grade for that course is greater than 2.

```
case class Student(name: String, exams: List[Exam])

case class Exam(courseId: String, grade: Double)

case class Course(id: String, name: String)

def generatePassedExams(
  students: List[Student], courses: List[Course]): List[(String, String, Double)] = {
    for {
      s <- students
      e <- s.exams
      if e.grade > 2
      c <- courses
      if e.courseId == c.id
    } yield (s.name, c.name, e.grade)
}
```

Your task is to rewrite the method `generatePassedExams` to use `map`, `flatMap` and `filter` instead of the for-comprehension. The resulting method should of course have the same result as the for-comprehension above.

Exercise 3: Variance (10 points)

Given the following hierarchy of classes:

```
class Writer[-D]
class Packet[+E]
```

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

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

- X <: Y
- Y <: Z

Part 1 (4 points)

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
`Packet[X]`		`Packet[Z]`
`Writer[X]`		`Writer[Y]`
`Writer[Packet[X]]`		`Writer[Packet[Y]]`
`Packet[Y => Y]`		`Packet[Z => X]`

Part 2 (6 points)

In the following implementations, which method signatures are valid? Wrong answers will incur negative points.

Mark all answers with an X.

```
class Writer[-D] {
  def getLast: D = ???                          Valid [ ]   Invalid [ ]
  def append(x: D): D = ???                     Valid [ ]   Invalid [ ]
  def write(x: D): Writer[D] = ???              Valid [ ]   Invalid [ ]
}

class Packet[+E] {
  def contains(x: E): Boolean = ???             Valid [ ]   Invalid [ ]
  def getLast: E = ???                          Valid [ ]   Invalid [ ]
  def toList: List[E] = ???                     Valid [ ]   Invalid [ ]
}
```

Exercise 4: Structural Induction (10 points)

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

```
sealed trait Expr
case class Lit(i: Int) extends Expr
case class Plus(l: Expr, r: Expr) extends Expr
```

Expression trees correspond to arithmetic expression made of integer literals and addition. For example the following expression:

```
((1 + 2) + 3)
```

Is represented as:

```
val e: Expr = Plus(Plus(Lit(1), Lit(2)), Lit(3))
```

Expressions can be evaluated to integers using the following recursive function:

```
def eval(e: Expr): Int =
  e match {
    case Lit(i)     => i
    case Plus(l, r) => eval(l) + eval(r)
  }
```

Consider the following `flip` function, that transforms expression trees by swapping the left and right-hand side of `Plus` nodes:

```
def flip(e: Expr): Expr =
  e match {
    case Lit(i)     => Lit(i)
    case Plus(l, r) => Plus(flip(r), flip(l))
  }

println(flip(e)) // Plus(Lit(3), Plus(Lit(2), Lit(1)))
```

Your goal in this exercise is to prove that the following property holds for any `expr` of type `Expr`:

```
eval(expr) === eval(flip(expr))
```

You can assume commutativity of addition on integer, that is, for every `i: Int` and `j: Int`, `i + j ===j + i`.

Note: Be *very precise* in your proof. Make sure to go one step at a time and clearly state what property you are using in each step of your proof.

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.