Functional Programming

Final Exam

Friday, December 21 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 can not 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 pages of this exam contains an appendix which is useful for formulating your solutions. You can detach these pages and keep them aside.

Using APIs Unless otherwise noted, you are always allowed to use methods of the Scala API that you know, even if they are not listed in the appendix. However, for Lisp, you may not use non-operator functions that are not listed in the appendix, unless you define them.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
4	10	
Total	40	

Exercise 1: Pure Functional Programming (10 points)

Our new Java intern has written some imperative code. We would prefer a purely functional solution using pattern matching. Your task is to re-implement the intern code using functional programming.

```
def flatMap[T](list: List[T], f: T => List[T]): List[T] = {
  var in = list
  var out: List[T] = Nil
  while (in.nonEmpty) {
    out = out ::: f(in.head)
    in = in.tail
  }
  out
}
```

This solution is undesirable for several reasons:

- it uses `var`-s and a `while` loop so it is not in the spirit of functional programming
- it has worse than linear complexity in the output list because of the expression `out ::: f(in.head)`

Write a new implementation of the `flatMap` method that produces the same result but satisfies the following properties:

- runs in time $O(n)$ where n is the size of the result
- no imperative constructs such as `var` and `while`
- use pattern matching instead of methods such as `nonEmpty`, `head`, `tail`
- it may define and implement additional methods
- any recursive method you implement must be tail-recursive
- it may only use the following methods on `List`: `::` and `:::`. Pattern matches are also allowed.

A correct solution will give 7 points and a correct tail recursive solution will give 10 points. Only implement one solution.

```scala
def flatMap[T](list: List[T], f: T => List[T]): List[T] =
```

Exercise 2: State (10 points)

Intuitively, an expression is *referentially transparent* if it always returns the same value, no matter the global state of the program, and thus can be replaced by that value without changing the result of the program.

More formally, an expression `e` is said to be referentially transparent if, in any program `P` where `e` is bound to some variable `x`, we can substitute each occurrence of `x` with `e` and obtain an equivalent program.

For example, given `def f(n:Int) = n+1`, we can say that `f(0)` is referentially transparent but `f(readInt)` is not, because in a program `P` such as:

```
P   = (n: Int) => { val x = e; println(x * x)},
```

if `e` is `f(0)` we can replace `x` by `e = f(0)` while retaining the same semantics, as in:

```
P'  = (n: Int) => { val x = f(0); println(f(0) * f(0))},
```

but if `e` is `f(readInt)` the following program would have a different semantics:

```
P'' = (n: Int) => { val x = f(readInt); println(f(readInt) * f(readInt))}
```

because `readInt` reads an integer on the standard input, an observable effect that is now duplicated.

Now, consider the following definitions:

```
class Counter {
  var count = 0
  def inc = count += 1
  def get = count
}
def f1(x: Int): Int = x * x
def f2(x: Int, y: Int): (Int,Int) = {
  var quotient = x / y
  var reminder = x % y
  (quotient, reminder)
}
def f3(xs: Seq[T], op: (Int,T) => Int): T = {
  var acc, i = 0
  while (i < xs.length) {
    acc = op(acc, xs(i))
    i += 1
  }
  acc
}
def f4(): Unit = ()
def f5(): Unit = println("hello world!")
def f6(c: Counter): Counter = { c.inc; c }
def f7(c: Counter): Int = c.get
def f8(n: Int): Counter => Unit = c => for (i <- 1 to n) c.inc
def f9[A](f: Int => A, x: Int): A = f(x)
def f10(f: Int => Int): Int => Int = {
  var cache: Option[(Int, Int)] = None
  x => cache match {
    case Some((arg, value)) if arg == x => value
    case _ =>
      val r = f(x)
      cache = Some((x, r))
      r
```

```
  }}
val f11: Int => Int = { val c0 = new Counter; f10(n => c0.get + n) }
```

Given arbitrary referentially-transparent expressions `n: Int`, `m: Int`, `xs: Seq[Int]`, and `c: Counter`, are the following expressions referentially transparent? Tick either the checkbox `Y` (for *yes*) or `N` (for *no*). A correct answer grants 0.5 point, an incorrect one detracts 0.25 point, and a lack of answer does nothing.

1. `Y[ ] N[ ] f1(n)`
2. `Y[ ] N[ ] f2(n, m)`
3. `Y[ ] N[ ] f3(xs, _ + _)`
4. `Y[ ] N[ ] f3(xs, _ + c.get + _)`
5. `Y[ ] N[ ] f4()`
6. `Y[ ] N[ ] f5()`
7. `Y[ ] N[ ] f6(c)`
8. `Y[ ] N[ ] f6(new Counter)`
9. `Y[ ] N[ ] f6(new Counter).get`
10. `Y[ ] N[ ] f7(c)`
11. `Y[ ] N[ ] f8(n)(c)`
12. `Y[ ] N[ ] f8(n)`
13. `Y[ ] N[ ] f8(c.get)`
14. `Y[ ] N[ ] f9((x:Int)=> (), c.get)`
15. `Y[ ] N[ ] f9(f1, f1(c.get))`
16. `Y[ ] N[ ] f9(x => y => println(x+y), 0)`
17. `Y[ ] N[ ] f10(f1)`
18. `Y[ ] N[ ] f10(x => f6(c).get + x)`
19. `Y[ ] N[ ] f10(x => c.get + x)`
20. `Y[ ] N[ ] f11`

Exercise 3: Lambda Calculus (10 points)

Church numerals are a representation of natural numbers using only functions. In this encoding, a number `n` is represented by a higher-order function that maps any function `f` to its `n`-fold composition. For examples, in Scheme−−, `0`, `1`, `2` and `3` are represented as follows:

- `(val zero (lambda (f x) x) ...)`
- `(val one (lambda (f x) (f x)) ...)`
- `(val two (lambda (f x) (f (f x))) ...)`
- `(val three (lambda (f x) (f (f (f x)))) ...)`

Church-encoded lists are a representation of lists using only functions. In this encoding, a list is represented by a higher-order function that takes two arguments and returns the first one when the list is empty and the second one applied to head and tail when the list is non-empty:

```
(val chNil
  (lambda (m n) m)

(def (chCons h t)
  (lambda (m n) (n h t))

...))
```

Church-encoded lists are constructed with `chCons` and `chNil` in the same way that normal lists are constructed using `cons` and `nil`. For instance, the list containing `'a`, `'b` and `'c` would be defined as follows in the two encodings:

- `(val myList (  cons 'a (  cons 'b (  cons 'c nil  ))) ...)`
- `(val chList (chCons 'a (chCons 'b (chCons 'c chNil))) ...)`

To decompose normal lists in Scheme−− one needs to use the built-in **`null?`**, **`car`**, **`cdr`** functions. For example, concatenation of two normal lists could be implemented as follows:

```
(def (concat l1 l2)
  (if (null? l1)
    l2
    (cons (car l1) (concat (cdr l1) l2))))
```

With church-encoded lists, decomposition is achieved by "applying" the list to a pair of continuations, one for the empty case and another one for the non-empty case. For example, concatenation of two church-encoded lists could be implemented as follows:

```
(def (chConcat l1 l2)
  (l1
    l2
    (lambda (h t) chCons h (chConcat t l2))))
```

Exercise 3.1

Give a Scheme−− implementation of the `succ` function that takes a church numeral and returns its successor. For example, `(succ zero)` evaluates to `one`, `(succ one)` evaluates to `two`, and `(succ two)` evaluates to `three`.

Note that in this encoding numbers are represented as functions and cannot be compared with the builtin `=`.

```
(def (succ n)

)
```

Exercise 3.2

Give a Scheme−− implementation of the `size` function that takes a church-encoded list and returns its size as a church numeral. For example, `(size chNil)` evaluates to `zero`, `(size (chCons 1 chNil))` evaluates to `one`, and `(size (chCons 1 (chCons 2 chNil)))` evaluates to `two`. You are allowed to use the `succ` function defined earlier.

Note that in this encoding lists are represented as functions and cannot be compared with the builtin `=`.

```
(def (size l)

)
```

Exercise 4: Streams (10 points)

Implement a `transpose` function that takes as input a matrix represented as a stream of streams of strings and returns a transposed matrix of the same type. Assume that all the inner streams have the same size.

For example, if the input matrix is:

$$\begin{vmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \end{vmatrix}$$

the output matrix would be:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \\ a_4 & b_4 & c_4 \end{vmatrix}$$

The implementation of `transpose` must satisfy the following properties:

- it is generic, meaning that it should work for finite and infinite streams
- it does not use the built-in `transpose` function

You may define and implement additional methods and you may use functions on `Stream` defined in the appendix.

```
def transpose(src: Stream[Stream[String]]): Stream[Stream[String]] =
```

Appendix

Scala Collections API

List (containing elements of type A):

- `xs ::: (ys: List[A]): List[A]`: prepends the list `xs` to the left of `ys`, 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.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.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)]`.

Stream (containing elements of type A):

- `xs #:: (ys: => Stream[A]): Stream[A]`: Builds a new stream starting with the element `xs`, and whose future elements will be those of `ys`.

Stream (the object):

- `Stream.Empty: Stream[Nothing]`: The empty stream.
- `Stream.from(i: Int): Stream[Int]`: Creates an infinite stream of integers starting at i.

You can use the same `List` API for `Stream`, replacing `List` by `Stream`.

Scheme−− API

Syntax reference:

- (**val** name body rest): defines a value.
- (**def** (name arg1 ... argN) body rest): defines a function.
- (**lambda** (arg1 ... argN) body): defines an anonymous function.
- (f arg1 ... argN): applies a function.

Built-in functions:

- (**null?** x): returns true if the list x is empty.
- (**cons** x y): constructs a list with head x and tail y.
- **nil**: constructs an empty list.
- (**car** x): head of the list x.
- (**cdr** x): tail of the list x.
- (**nth** i x): element in position i of the list x starting from 0.
- (**if** cond then else): returns the term then if cond evaluates to true, and **else** otherwise.